Principled Scavenging*

Stefan Monnier Bratin Saha Zhong Shao

Department of Computer Science
Yale University
New Haven, CT 06520-8285
{monnier, saha, shao}@cs.yale.edu

ABSTRACT

Proof-carrying code and typed assembly languages aim to minimize the trusted computing base by directly certifying the actual machine code. Unfortunately, these systems cannot get rid of the dependency on a trusted garbage collector. Indeed, constructing a provably type-safe garbage collector is one of the major open problems in the area of certifying compilation.

Building on an idea by Wang and Appel, we present a series of new techniques for writing type-safe stop-and-copy garbage collectors. We show how to use intensional type analysis to capture the contract between the mutator and the collector, and how the same method can be applied to support forwarding pointers and generations. Unlike Wang and Appel (which requires whole-program analysis), our new framework directly supports higher-order funtions and is compatible with separate compilation; our collectors are written in provably type-safe languages with rigorous semantics and fully formalized soundness proofs.

1. INTRODUCTION

The correctness of most type-safe systems relies critically on the correctness of an underlying garbage collector (GC). This also holds for Proof-Carrying Code (PCC) [14] and Typed Assembly Languages (TAL) [13]—both of which aim to minimize the trusted computing base by directly certifying the actual machine code. Unfortunately, these systems cannot get rid of the dependency on a trusted garbage collector. Indeed, constructing a verifiably type-safe garbage collector is widely considered as one of the major open problems in the area of certifying compilation [12, 3].

Recently, Wang and Appel [25] proposed to tackle the problem by building a tracing garbage collector on top of a region-based calculus. Our work builds on theirs but makes the following new significant contributions:

- We show how to use intensional type analysis (ITA) [21, 8] to accurately describe the contract between the mutator and the collector and how the same framework can be applied to construct various different type-safe GCs.
- Using ITA to typecheck GC may seem to be an obvious idea to some people, however, none of the previous work [23, 17, 21] have succeeded in getting it to work. Indeed, Wang and Appel [25] subsequently gave up on using ITA. We show why the problem is nontrivial (see Section 2.2) and how to modify the basic ITA framework to solve the problem.
- Wang and Appel's collector [25] relies on whole-program analysis and code duplication to support higher-order and polymorphic languages; this breaks separate compilation and is impractical. We show how to use runtime type analysis to write our GC as a library (thus no code duplication) and how to directly support higher-order polymorphic functions.
- We expose in detail how to implement and certify efficient forwarding pointers. Making them type-safe is surprisingly subtle (see Section 7). Wang and Appel [25] also claim to support forwarding pointers but their scheme is less efficient and it is unclear whether it is sound.
- We also show how to handle generations with a simple extension of our base calculus.
- A garbage collector is type-safe only if it is written in a provably type-safe language. We have complete type-soundness proofs for all our calculi (see the companion technical report [11]).Wang and Appel's collectors [25, 24], on the other hand, are not fully formalized.

Although our paper is theoretical in nature, we believe it will be of great interests to the general audience, especially those who are looking to apply new language theory to solve important practical problems such as mobile-code safety and certifying compilation. We have started implementing our type-safe garbage collector in the FLINT system [18], however, making the implementation realistic still involves solving several problems (e.g., breadth-first copying, remembered sets, and data structures with cycles, which we still cannot support satisfactorily). Thus implementation issues are beyond the scope of this paper. Nevertheless, we believe our current contributions constitute a significant step towards the goal of providing a practical type-safe garbage collector.

*This research was sponsored in part by the Defense Advanced Research Projects Agency ISO under the title "Scaling Proof-Carrying Code to Production Compilers and Security Policies," ARPA Order No. H559, issued under Contract No. F30602-99-1-0519, and in part by NSF Grants CCR-9901011 and CCR-0081590. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

2. MOTIVATION AND APPROACH

Why do we want a type-safe garbage collector?

The explosive growth of the Internet has induced newfound interest in mobile computation as well as security. Increasingly, applications are being developed at remote sites and then downloaded for execution. A robust mobile code system must allow code from potentially untrusted sources to be executed. At the same time, the system must detect and prevent the execution of malicious code.

The safety of such a system depends not only on the properties of the code being downloaded, but also on the security of the host system itself, or more specifically, its trusted computing base (TCB).

Proof-carrying code and typed assembly languages have been proposed to reduce the size of this TCB by bundling the untrusted code with a mechanically checkable proof of safety, where the safety is usually defined as type-safety. Such systems only need to trust their verifier and runtime system rather than their whole compiler suite.

But all these certifying-compiler projects (e.g., PCC, TAL) still crucially rely on the correctness of a tracing garbage collector for their safety. Recently, both Crary [3] and Morrisett [12] have characterized type-safe garbage collection as one of the major open problems in the area of certifying compilation.

A type-safe GC is not only desirable for reducing the size of the TCB but also for making it possible to ship custom-tailored GC along with mobile code, or to choose between many more GC variants without risking the integrity of the system. Writing GC inside a type-safe language itself also makes it possible to achieve principled interoperation between garbage collection and other memory-management mechanisms (e.g., those based on malloc-free and regions). Indeed, one major software-engineering benefit is that a type-safe GC must make explicit the contract between the collector and the mutator and it must make sure that it is always respected. Without typechecking, such rules can prove difficult to implement correctly and bugs can be very difficult to find.

2.1 The problem

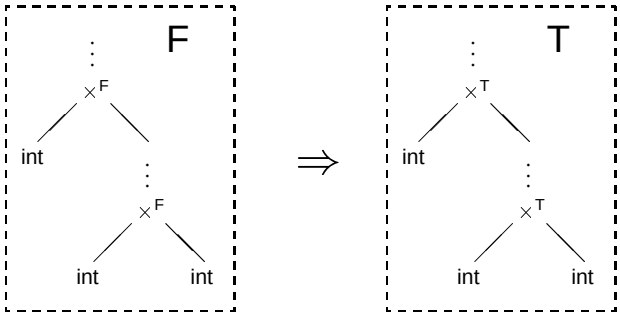
Recently, Wang and Appel [23] proposed to tackle the problem by layering a stop-and-copy tracing garbage collector [26] on top of a region based calculus, thus providing both type safety and completely automatic memory management.

A region calculus [19] annotates the type of every heap object with the region in which it is allocated (such as $\sigma_1 \times^\rho \sigma_2$ where ρ is the region), thus allowing to safely reclaim memory by freeing any region that does not appear in any of the currently live types.

The basic idea in building a type-safe GC is to concentrate on type-safety rather than correctness. Rather than try to prove that the *copy* function faithfully copies all the heap, we just need to show that it has a type looking somewhat like $\forall\alpha.(\alpha \rightarrow (\alpha[\mathsf{T}/\mathsf{F}]))$ where $(\alpha[\mathsf{T}/\mathsf{F}])$ stands for the type α where the region annotation T is substituted for F (see Fig. 1). Assuming we have such a function and we don't keep any reference to the region F, the region calculus will allow us to safely reclaim F.

Clearly, there is no correctness guarantee in sight since the *copy* function might return a completely different value or might not faithfully reproduce the original graph, but it ensures type-safe execution of the whole mutator-collector system and even offers a form of type-preservation guarantee.

The main problem is clearly to write this *copy* function which needs to trace through arbitrary heap structures at runtime. Therefore, the language needs to support some form of runtime type information in order to do the actual *copy*.



$$copy : \forall\mathsf{F}.\forall\mathsf{T}.\forall\alpha.(\alpha \rightarrow \alpha[\mathsf{T}/\mathsf{F}])$$

$$GC = \Lambda\mathsf{F}.\Lambda\alpha.\lambda(x:\alpha, k:\forall\rho.\alpha[\rho/\mathsf{F}] \rightarrow 0)$$
$$\quad \text{let region } \mathsf{T} \text{ in}$$
$$\quad \text{let } y = copy[\mathsf{F}][\mathsf{T}][\alpha](x) \text{ in}$$
$$\quad \text{only } \mathsf{T} \text{ in } k[\mathsf{T}](y)$$

Figure 1: Stop-and-Copy from region F to region T. GC is written in continuation passing style (CPS). It takes the current region, the heap and a continuation and begins by allocating a new region T with "let region T in e". It then copies the heap into this new region and frees the old region implicitly with "only T in e" which tells that all regions but T can be reclaimed. This way of freeing regions was introduced by Wang and Appel to circumvent problems linked to aliasing of regions.

In their followup paper [25], Wang and Appel suggest to circumvent the problem of runtime type information using a mix of monomorphization and defunctionalization (a form of closure conversion due to Tolmach [20] which reduces the language to a monomorphic first-order calculus). However, this approach suffers from several major drawbacks:

- It can introduce a significant code size increase and forces the use of separate specialized *GC* and *copy* functions for each type appearing in the program. Instead of the promised flexibility to choose among various GC variants, this approach locks you into a single 100% tailor-made collector.
- Monomorphization is not applicable in the presence of recursive polymorphism or existential packages, so their type-safe GC cannot handle languages with polymorphic recursion or abstract types.
- Monomorphization and defunctionalization all rely on whole program analysis which are clearly imcompatible with separate compilation.
- Finally, although their type-safe GC does properly formalize the interaction between the mutator and the collector, the formalization is hidden inside the compiler and hence does not allow to bring out open the overly intimate relationship between the GC and the compiler.

They also try to preserve sharing using forwarding pointers. The rough sketch of the solution they propose is similar to the one we developed (which was done independently). It relies mostly on a form of *cast* which allows covariant subtyping of references. Making sure that this cast is sufficiently constrained to be safe is difficult. Their informal presentation is incomplete and possibly incorrect, and leaves many important questions unanswered.

2.2 Our solution

We want to do away with any form of whole program analysis so as to make the mutator and the collector independent in order to reap the promised benefits of more flexibility and clearer interaction between mutator and GC.

In this paper, we present a different approach for writing the *copy* function, relying on runtime type analysis. The substitution present in the return type of *copy* as well as the need to observe types at runtime leads one very naturally to use intensional type analysis (ITA) [21, 8]. In fact, an earlier paper of Wang and Appel [23] was titled "safe garbage collection = regions + intensional type analysis" but they subsequently gave up on using ITA and opted for the solution mentioned above [25]. Saha et al. [17, 21] also tried to use ITA to write the *copy* function, but their attempt is missing crucial details and didn't really work either.

2.2.1 Intensional type analysis

A full description of ITA is outside the scope of this paper, but in short, ITA provides a typecase term to examine at runtime the actual type with which a type variable was instantiated (using some form of runtime type descriptor) and a typerec type to write type functions that do the equivalent of a *fold* to map one type into another. You can thus define a type function that replaces all occurrences of Int with Bool. One important property is that typerec functions only analyse type constructors and linger when faced with a type variable until that variable is instantiated.

2.2.2 A case for symmetry

So what is the problem? It seems that ITA provides us with just the right tools. We can for example write a simple typerec function $\mathsf{S}_{\mathsf{T},\mathsf{F}}(\sigma)$ which substitutes region annotations T for F by recursively analyzing the type σ. At the term level, in the body of *copy*, we can similarly use typecase to determine the actual type of the object we are copying.

But that means that the type grows each time we go through the GC, from σ to $\mathsf{S}_{\mathsf{T},\mathsf{F}}(\sigma)$ to $\mathsf{S}_{\rho,\mathsf{T}}(\mathsf{S}_{\mathsf{T},\mathsf{F}}(\sigma))\ldots$. This may seem unimportant since S should be reduced away anyway. But $\mathsf{S}_{\rho,\mathsf{T}}(\alpha)$ cannot be reduced further until α is instantiated: $\exists\alpha.\mathsf{S}_{\mathsf{T},\mathsf{F}}(\alpha)$ is a normal form. So the accumulation of S operators is a real problem, since $\mathsf{S}_{\rho,\mathsf{F}}(\alpha)$ is not equal to $\mathsf{S}_{\rho,\mathsf{T}}(\mathsf{S}_{\mathsf{T},\mathsf{F}}(\alpha))$.

We could arrange for $\mathsf{S}_{\rho,\mathsf{T}}(\mathsf{S}_{\mathsf{T},\mathsf{F}}(\sigma))$ to reduce to $\mathsf{S}_{\rho,\mathsf{F}}(\sigma)$. But then all types become $\mathsf{S}_{\rho,\mathsf{F}}(\sigma)$ (where F is the "initial region") except before the first collection. Also it is very ad-hoc and only works if S obeys this kind of idempotency.

A better approach is to ensure that the input and output types of *copy* are symmetric. We first redefine $\mathsf{S}_\rho(\sigma)$ which simply substitutes ρ for any region annotation (why bother with an initial region) and then redefine *copy* to have type $\forall\mathsf{F}.\forall\mathsf{T}.\forall\alpha.(\mathsf{S}_\mathsf{F}(\alpha)\to\mathsf{S}_\mathsf{T}(\alpha))$ which gets us rid of the special case before the first collection and does not require any special reduction rule for S since *GC* does not increase the size of the type any more.

2.2.3 A case for tags

The above solution looks good until we try to copy existential packages $\exists\alpha\in\Delta.\sigma$, used to encode closures. The Δ annotation is used to bound the set of regions that can appear in the witness type hidden under the type variable α.

Opening an existential package of type $\exists\alpha\in\{\mathsf{F}\}.\mathsf{S}_\mathsf{F}(\alpha)$, gives us the value σ of α (i.e. the witness type) and a value of type $\mathsf{S}_\mathsf{F}(\sigma)$. Recursively applying *copy* to that value will return a new value of type $\mathsf{S}_\mathsf{T}(\sigma)$, but how can we construct the new existential package of type $\exists\alpha\in\{\mathsf{T}\}.\mathsf{S}_\mathsf{T}(\alpha)$? Reusing σ for the witness type will not do since σ is not constrained to $\{\mathsf{T}\}$ but to $\{\mathsf{F}\}$. A witness of $\mathsf{S}_\mathsf{T}(\sigma)$ cannot work either; the only correctly typed package we can produce is $\langle\alpha=\mathsf{S}_\mathsf{T}(\sigma), v:\alpha\rangle$ which has type $\exists\alpha\in\{\mathsf{T}\}.\alpha$.

Clearly, we are again trying to push a new S onto the type rather than replacing an S with another. So we can again arrange for $\mathsf{S}_\mathsf{T}(\mathsf{S}_\mathsf{F}(\sigma))$ to reduce to $\mathsf{S}_\mathsf{T}(\sigma)$, but we really do not want to tie our hands with such an ad-hoc and restrictive scheme.

Instead, we can pay a bit more attention to what we do and observe that $\mathsf{S}_\rho(\sigma)$ makes region annotations on σ completely useless, so instead of trying to get those annotations right only to see them substituted we can simply define a parallel set of non-annotated types τ (that we will call *tags*). Since tags have no region annotations, we can hide them in tag variables without any Δ constraint, which side-steps the problem when copying existentials conveniently. Note that contrary to common practice, our tags are not attached to their corresponding objects but are managed completely independently.

Such a split between types and tags is not a new concept since it was already used in the work on intensional type analysis where tags were called *constructors* [8, 5]. But here, tags take on more significance since they correspond to a source-level notion of type and will be mapped to *different* actual types with different type functions M (formerly S) which are used to encapsulate all the constraints that mutator data has to satisfy in order for the collector to do its job. As you will see in sections 7 and 8 we will use a non-trivial M mapping to force the mutator to provide space for forwarding pointers and to enforce the invariant that references do not point from the old generation to the new.

3. SOURCE LANGUAGE λ_{CLOS}

For simplicity of the presentation, the source language we propose to compile and garbage collect is the simply typed λ-calculus.

In order to be able to use our region calculus, we need to convert the source program into a continuation passing style form (CPS). And we also need to close our code to make all data manipulation explicit, so we turn all closures into existential packages.

We will not go into the details of how to do the CPS conversion [7] and the closure conversion using existentials [10, 9].

The language used after CPS conversion and closure conversion is the language λ_{CLOS} shown below.

$$
\begin{array}{lll}
(\textit{types}) & \tau ::= \mathsf{Int} \mid t \mid \tau_1\times\tau_2 \mid \tau\to 0 \mid \exists t.\tau \\
(\textit{values}) & v ::= n \mid f \mid x \mid (v_1,v_2) \mid \langle t=\tau_1, v:\tau_2\rangle \\
(\textit{terms}) & e ::= \mathsf{let}\ x=v\ \mathsf{in}\ e \mid \mathsf{let}\ x=\pi_i v\ \mathsf{in}\ e \\
 & \qquad \mid v_1(v_2) \mid \mathsf{open}\ v\ \mathsf{as}\ \langle t,x\rangle\ \mathsf{in}\ e \mid \mathsf{halt}\ v \\
(\textit{programs}) & p ::= \mathsf{letrec}\ \overrightarrow{f=\lambda(x:\tau).e}\ \mathsf{in}\ e
\end{array}
$$

Since functions are in CPS, they never return, which we represent with the arbitrary return type 0, often pronounced *void*. (v_1, v_2) builds a pair while $\pi_i v$ selects its i^{th} element. To represent closures, the language includes existential packages of type $\exists t.\tau_2$ and constructed by $\langle t=\tau_1, v:\tau_2\rangle$. The construct

$$\mathsf{open}\ v\ \mathsf{as}\ \langle t,x\rangle\ \mathsf{in}\ e$$

takes an existential package v, binds the witness type to t and the value to x, and then executes e. The complete program

$$\mathsf{letrec}\ \overrightarrow{f_i=\lambda(x_i:\tau_i).e_i}\ \mathsf{in}\ e$$

consists of a list of mutually recursive closed function declarations followed by the main term to be executed.

$$
\begin{array}{lll}
(\textit{regions}) & \rho & ::= \nu \mid r \\
(\textit{kinds}) & \kappa & ::= \Omega \mid \Omega \longrightarrow \Omega \\
(\textit{tags}) & \tau & ::= t \mid \mathsf{Int} \mid \tau_1 \times \tau_2 \mid \tau \to 0 \mid \exists t.\tau \\
 & & \quad \mid \lambda t.\tau \mid \tau_1 \tau_2 \\
(\textit{types}) & \sigma & ::= \mathsf{int} \mid \sigma_1 \times \sigma_2 \mid \forall[\vec{t}][\vec{r}](\vec{\sigma}) \to 0 \mid \exists t.\sigma \\
 & & \quad \mid \sigma \ \mathsf{at}\ \rho \mid \mathsf{M}_\rho(\tau) \\
 & & \\
(\textit{values}) & v & ::= n \mid x \mid \nu.\ell \mid (v_1, v_2) \mid \langle t = \tau, v : \sigma \rangle \\
 & & \quad \mid \lambda[\vec{t}][\vec{r}](\overrightarrow{x:\sigma}).e \\
(\textit{operations}) & op & ::= v \mid \pi_i v \mid \mathsf{put}[\rho]v \mid \mathsf{get}\ v \\
(\textit{terms}) & e & ::= v[\vec{\tau}][\vec{\rho}](\vec{v}) \mid \mathsf{let}\ x = op\ \mathsf{in}\ e \mid \mathsf{halt}\ v \\
 & & \quad \mid \mathsf{ifgc}\ \rho\ e_1\ e_2 \mid \mathsf{open}\ v\ \mathsf{as}\ \langle t, x \rangle\ \mathsf{in}\ e \\
 & & \quad \mid \mathsf{let\ region}\ r\ \mathsf{in}\ e \mid \mathsf{only}\ \Delta\ \mathsf{in}\ e \\
 & & \quad \mid \mathsf{typecase}\ \tau\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\to}; t_1 t_2.e_{\times}; t_e.e_{\exists})
\end{array}
$$

Figure 2: Syntax of λ_{GC}

4. TARGET LANGUAGE λ_{GC}

We translate λ_{CLOS} programs into our target language λ_{GC}. The target language is also used to write the garbage collector. Figure 2 gives the syntax of λ_{GC}; the semantics are presented in Section 6). λ_{GC} extends λ_{CLOS} with regions [19] and (fully reflexive) intensional type analysis [21]. Functions are also fully closed and use CPS but they can additionally be polymorphic over tags and regions.

4.1 Regions

Our region calculus uses *reference* values $\nu.\ell$ of type $(\sigma\ \mathsf{at}\ \nu)$. ν is the region in which the object is allocated and ℓ is the actual location within that region. Object allocation and memory accesses are made explicit with put and get. In order to trigger GC, ifgc allows us to check whether a region is full. Ensuring timely collection using heap limit checks or other mechanisms is outside the scope of this paper, so put never fails, even if the region is "full".

Region allocation and reclamation is done with let region and only. Deallocation of a region is implicit since only lists the regions that should be kept. This neatly works around aliasing problems, at the cost of a more expensive deallocation operation (only needs to go through the list of all regions to find which ones need to be reclaimed). In our case, we have very few regions and deallocate them only occasionally, so it is a good tradeoff.

4.2 Functions and code

Since programs in λ_{GC} are completely closed, we can separate code from data. The memory configuration enforces this by having a separate dedicated region cd for all the code blocks. A value $\lambda[\vec{t}][\vec{r}](\overrightarrow{x:\sigma}).e$ is only an array of instructions (which can contain references to other values in cd) and needs to be put into a region to get a function pointer before one can call it. In practice, functions are placed into the cd region when translating code from λ_{CLOS} and never directly appear in λ_{GC} code.

The indirection provided by memory references allows us to do away with letrec. For convenience, we will use $\mathsf{fix} f.e$ in examples, but in reality, e will be placed at the location ℓ in the cd region and all occurrences of f will be replaced by $\mathsf{cd}.\ell$.

4.3 Intensional type analysis

As explained earlier, we have split the notion of type into two. Tags represent the runtime type descriptors and map very directly to source-level types without any region annotations. The only difference between λ_{CLOS} types and λ_{GC} tags is the addition of tag functions $\lambda t.\tau$ and tag applications $\tau_1\tau_2$, which are needed for type analysis of existentials [21]. To do the actual analysis of tags, terms include a *refining* typecase construct, i.e. a more refined tag is substituted for τ in each arm of the typecase. Finally, instead of a full-blown typerec construct we only provide a hard-coded M, to keep the presentation simpler. $\mathsf{M}_\rho(\tau)$ is the type corresponding to the tag τ complemented with region annotations ρ:

$$
\begin{array}{ll}
\mathsf{M}_\rho(\mathsf{Int}) & \Longrightarrow \mathsf{int} \\
\mathsf{M}_\rho(\tau_1 \times \tau_2) & \Longrightarrow (\mathsf{M}_\rho(\tau_1) \times \mathsf{M}_\rho(\tau_2))\ \mathsf{at}\ \rho \\
\mathsf{M}_\rho(\exists t.\tau) & \Longrightarrow (\exists t.\mathsf{M}_\rho(\tau))\ \mathsf{at}\ \rho \\
\mathsf{M}_\rho(\tau \to 0) & \Longrightarrow \forall[][r](\mathsf{M}_r(\tau)) \to 0\ \mathsf{at}\ \mathsf{cd}
\end{array}
$$

This definition of M forces the mutator to maintain the invariant that all objects are allocated in the same region, which is all our garbage collector requires.

$\boxed{F \vdash \lambda_{\mathsf{CLOS}} \Rightarrow \lambda_{\mathsf{GC}}}$

$$\frac{}{F \vdash_v n \Rightarrow n} \qquad \frac{}{F \vdash_v f \Rightarrow \mathsf{cd}.F(f)} \qquad \frac{}{F \vdash_v x \Rightarrow x}$$

$$\frac{F \vdash_v v_1 \Rightarrow v_1' \quad F \vdash_v v_2 \Rightarrow v_2'}{F \vdash_v (v_1, v_2) \Rightarrow \mathsf{put}[r](v_1', v_2')}$$

$$\frac{F \vdash_v v \Rightarrow v'}{F \vdash_v \langle t = \tau_1, v : \tau_2 \rangle \Rightarrow \mathsf{put}[r]\langle t = \tau_1, v' : \mathsf{M}_r(\tau_2) \rangle}$$

$$\frac{F \vdash_v v_1 \Rightarrow v_1' \quad F \vdash_v v_2 \Rightarrow v_2'}{F \vdash_e v_1(v_2) \Rightarrow v_1'[][r](v_2')} \qquad \frac{F \vdash_v v \Rightarrow v'}{F \vdash_e \mathsf{halt}\ v \Rightarrow \mathsf{halt}\ v'}$$

$$\frac{F \vdash_e e \Rightarrow e' \quad F \vdash_v v \Rightarrow v'}{F \vdash_e \mathsf{open}\ v\ \mathsf{as}\ \langle t, x \rangle\ \mathsf{in}\ e \Rightarrow \mathsf{open}\ (\mathsf{get}\ v')\ \mathsf{as}\ \langle t, x \rangle\ \mathsf{in}\ e'}$$

$$\frac{F \vdash_e e \Rightarrow e' \quad F \vdash_v v \Rightarrow v'}{F \vdash_e \mathsf{let}\ x = v\ \mathsf{in}\ e \Rightarrow \mathsf{let}\ x = v'\ \mathsf{in}\ e'}$$

$$\frac{F \vdash_e e \Rightarrow e' \quad F \vdash_v v \Rightarrow v'}{F \vdash_e \mathsf{let}\ x = \pi_i v\ \mathsf{in}\ e \Rightarrow \mathsf{let}\ x = \pi_i(\mathsf{get}\ v')\ \mathsf{in}\ e'}$$

$$\frac{F \vdash_e e \Rightarrow e' \quad \ell = F(f)}{\begin{array}{l}\vdash_f (f = \lambda(x:\tau).e) \\ \quad \Rightarrow\ \lambda[][r](x : \mathsf{M}_r(\tau)).\mathsf{ifgc}\ r\ (gc[\tau][r](\mathsf{cd}.\ell, x))\ e'\end{array}}$$

$$\frac{\begin{array}{c}F = \{f_1 \mapsto \ell_1, \ldots, f_n \mapsto \ell_n\} \\ F \vdash_f f_i = \lambda(x_i:\tau_i).e_i \Rightarrow f_i' \qquad F \vdash_e e \Rightarrow e'\end{array}}{\begin{array}{l}\vdash_p \mathsf{letrec}\ \overrightarrow{f_i = \lambda(x_i:\tau_i).e_i}\ \mathsf{in}\ e \\ \quad \Rightarrow\ (\{\mathsf{cd} \mapsto \{\ell_1 \mapsto f_1', \ldots\}\}, \mathsf{let\ region}\ r\ \mathsf{in}\ e')\end{array}}$$

Figure 3: Translation of λ_{CLOS} terms.

5. TRANSLATING λ_{CLOS} TO λ_{GC}

The translation of terms from λ_{CLOS} to λ_{GC} shown in Fig. 3 is mostly directed by the type translation M_ρ presented earlier: each

```
fix gc[t][r1](f:∀[][r](Mr(t)) → 0, x:Mr1(t)).
  let region r2 in
  let y = copy[t][r1, r2](x) in
  only {r2} in f[][r2](y)

fix copy[t][r1, r2](x:Mr1(t)) : Mr2(t).
  typecase t of
    Int    ⇒ x
    →      ⇒ x
    t1 × t2 ⇒ let x1 = copy[t1][r1, r2](π1(get x)) in
              let x2 = copy[t2][r1, r2](π2(get x)) in
              put[r2](x1, x2)
    ∃te    ⇒ open (get x) as ⟨t, y⟩ in
              let z = copy[tet][r1, r2](y) in
              put[r2]⟨t=t, z:Mr2(tet)⟩
```

Figure 4: The garbage collector proper.

function takes the current region as an argument and begins by checking if a garbage collection is necessary. All operations on data are slightly rewritten to account for the need to allocate them in the region or to fetch them from the region. For example a λ_{CLOS} function like:

```
fix swap(x:Int × Int).
  let x1 = π1x in let x2 = π2x in let x' = (x2, x1) in halt 0
```

will be turned into the following λ_{GC} function:

```
cd.ℓ = λ[][r](x:(int × int) at r).
         ifgc r (gc[Int × Int][r](cd.ℓ, x))
           let x = get x in
           let x1 = π1x in
           let x2 = π2x in
           let x' = put[r](x2, x1) in
           halt 0
```

The mapping between λ_{CLOS} identifiers like $swap$ and λ_{GC} location like $\mathsf{cd}.\ell$ is kept in F. The new argument r refers to the current region. It is initially created at the very beginning of the program and is changed after each garbage collection.

An important detail here is that the garbage collector receives the tag τ rather than the type σ of the argument. The GC receives the tags for analysis as they were in λ_{CLOS} rather than as they are translated in λ_{GC}. This maintains a clear distinction between the types the programmer thinks he manipulates and the underlying types they map to.

Another interesting detail is that if the region is full, the function calls the garbage collector with itself as the return function. I.e. when the collection is finished, the collector will jump back to the function which will then redo the check. We could instead call the garbage collector with another function as argument. That would save us from redoing the ifgc but would require many tiny functions which are just not worth bothering with.

The translation in Fig. 3 uses λ_{GC} in a somewhat loose way to keep the presentation concise. More specifically, it will generate terms such as $\mathsf{let}\ x = \pi_i(\mathsf{get}\ v)\ \mathsf{in}\ e$ instead of

$$\mathsf{let}\ x' = \mathsf{get}\ v\ \mathsf{in}\ \mathsf{let}\ x = \pi_i x'\ \mathsf{in}\ e.$$

Turning such code back into the strict λ_{GC} is very straightforward.

On the other hand, the garbage-collection code in Fig. 4 uses not only some syntactic sugar but even resorts to using a direct-style presentation of the $copy$ function. This is only for clarity of presentation, of course. The companion technical report [11] presents a fully CPS- and closure-converted code that is equivalent but more difficult to read.

The garbage collector itself is very simple: it first allocates the *to* region, asks $copy$ to move everything into it and then free the *from* region before jumping to its continuation, using the new region.

The $copy$ function is similarly straightforward, recursing over the whole heap and copying in a depth-first way. Clearly, the direct style here hides the stack. When the code is CPS converted and closed , we have to allocate that stack of continuations in an additional temporary region and unless our language is extended with some notion of stack, none of those continuations would be collected until the end of the whole garbage collection. The size of this temporary region can be bounded by the size of the *to* region since we can't allocate more than one continuation per copied object, so it is still algorithmically efficient, although this memory overhead is a considerable shortcoming.

6. A CLOSER LOOK AT λ_{GC}

Programs in λ_{GC} use an allocation semantics which makes the allocation of data in memory explicit. The semantics, defined in Fig. 5, maps a machine state P to a new machine state P'. A machine state is a pair (M, e) of a memory M and a term e being executed. A memory consists of a set of regions; hence, it is defined formally as a map between region names ν and regions R. A region, in turn, is a map from offsets ℓ to storable values v. Therefore, an address is given by a pair of a region and an offset $\nu.\ell$. We assign a type to every location allocated in a region with the memory environment Ψ. Fig. 6 shows the form of environments while Fig. 7 presents the static semantics.

6.1 Functions and code

Since function bodies can contain references to other functions in cd but we do not have an easy way for the garbage collector to analyze a function body to trace through those references, cd enjoys a special status. It cannot be freed and can only contain functions, no other kind of data.

An alternative would be to require all functions to be fully closed, but that would require the addition of recursive types for the environment containing pointers to all functions and passed around everywhere. It would save us from so many cd special cases, and would allow garbage collecting code, but on the other hand, it would be less realistic since it would amount to disallowing direct function calls.

6.2 The type calculus

The target language must be expressive enough to write a tracing garbage collector. Since the garbage collector needs to know the type of values at runtime, the language λ_{GC} must support the runtime analysis of types. Therefore, conceptually, types need to play a dual role in this language. As in the source language λ_{CLOS}, they are used at compile time to type-check well formed terms. However, they are also used at runtime, as tags, to be inspected by the garbage collector (and, in general, by any type analyzing function). To enforce this distinction, we split types into a tag language and a type language. The tags correspond to the runtime entity, while the types correspond to the compile time entity.

During the translation from λ_{CLOS} to λ_{GC}, the tag for a value must be constructed from its type, so the tags in λ_{GC} closely resemble the type language in λ_{CLOS}. To support tag analysis, we need to add tag-level functions ($\lambda t.\tau$) and tag-level applications ($\tau\tau_1$) which in turn requires adding the function kind $\Omega \longrightarrow \Omega$.

$$
\begin{array}{ll}
(M, \nu.\ell[\vec{\tau}'][\vec{\rho}](\vec{v})) & \\
\quad \text{where } M(\nu.\ell) = (\lambda[\vec{t}][\vec{r}](\vec{x}{:}\vec{\sigma}).e) & \Longrightarrow (M, e[\vec{\rho}, \vec{\tau}', \vec{v}/\vec{r}, \vec{t}, \vec{x}]) \\
(M, \mathsf{let}\ x = v\ \mathsf{in}\ e) & \Longrightarrow (M, e[v/x]) \\
(M, \mathsf{let}\ x = \pi_i(v_1, v_2)\ \mathsf{in}\ e) & \Longrightarrow (M, e[v_i/x]) \\
(M, \mathsf{let}\ x = \mathsf{put}[\nu]v\ \mathsf{in}\ e) & \Longrightarrow (M\{\nu.\ell \mapsto v\}, e[\nu.\ell/x]) \quad \text{where } \ell \notin Dom(M(\nu)) \\
(M, \mathsf{let}\ x = \mathsf{get}\ \nu.\ell\ \mathsf{in}\ e) & \Longrightarrow (M, e[v/x]) \quad \text{where } M(\nu.\ell) = v \\
(M, \mathsf{open}\ \langle t{=}\tau', v{:}\sigma\rangle\ \mathsf{as}\ \langle t, x\rangle\ \mathsf{in}\ e) & \Longrightarrow (M, e[\tau', v/t, x]) \\
(M, \mathsf{ifgc}\ \rho\ e_1\ e_2) & \Longrightarrow (M, e_1) \quad \text{if } \rho \text{ is full} \\
(M, \mathsf{ifgc}\ \rho\ e_1\ e_2) & \Longrightarrow (M, e_2) \quad \text{if } \rho \text{ is not full} \\
(M, \mathsf{let\ region}\ r\ \mathsf{in}\ e) & \Longrightarrow (M\{\nu \mapsto \{\}\}, e[\nu/r]) \quad \text{where } \nu \notin Dom(M) \\
(M, \mathsf{only}\ \Delta\ \mathsf{in}\ e) & \Longrightarrow (M|_\Delta, e) \\
(M, \mathsf{typecase}\ \mathsf{Int}\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\rightarrow}; t_1t_2.e_{\times}; t_e.e_{\exists})) & \Longrightarrow (M, e_i) \\
(M, \mathsf{typecase}\ \tau \rightarrow 0\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\rightarrow}; t_1t_2.e_{\times}; t_e.e_{\exists})) & \Longrightarrow (M, e_{\rightarrow}) \\
(M, \mathsf{typecase}\ \tau_1 \times \tau_2\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\rightarrow}; t_1t_2.e_{\times}; t_e.e_{\exists})) & \Longrightarrow (M, e_{\times}[\tau_1, \tau_2/t_1, t_2]) \\
(M, \mathsf{typecase}\ \exists t.\tau\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\rightarrow}; t_1t_2.e_{\times}; t_e.e_{\exists})) & \Longrightarrow (M, e_{\exists}[\lambda t.\tau/t_e])
\end{array}
$$

Figure 5: Operational semantics of λ_{GC}.

$$
\begin{array}{lll}
(\textit{tenv}) & \Theta & ::= \cdot \mid \Theta, t{:}\kappa \\
(\textit{venv}) & \Gamma & ::= \cdot \mid \Gamma, x{:}\sigma \\
(\textit{renv}) & \Delta & ::= \cdot \mid \Delta, \rho \\
(\textit{region types}) & \Upsilon & ::= \{\ell_1{:}\sigma_1, \ldots, \ell_n{:}\sigma_n\} \\
(\textit{mem types}) & \Psi & ::= \{\mathsf{cd}{:}\Upsilon_{\mathsf{cd}}, \nu_1{:}\Upsilon_1, \ldots, \nu_n{:}\Upsilon_n\} \\
(\textit{regions}) & R & ::= \{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\} \\
(\textit{memories}) & M & ::= \{\mathsf{cd} \mapsto R_{\mathsf{cd}}, \nu_1 \mapsto R_1, \ldots, \nu_n \mapsto R_n\} \\
(\textit{states}) & P & ::= (M, e)
\end{array}
$$

Figure 6: λ_{GC} environments

Types are used to classify terms. The type language includes the existential type for typing closures and the code type $\forall[\vec{t}][\vec{r}](\vec{\sigma}) \rightarrow 0$ for fully closed CPS functions. Moreover, types in the target language must include the region in which the corresponding value resides. Therefore, we use the notation $\sigma\ \mathsf{at}\ \rho$ for the type of a value of type σ in region ρ.

To reason about the safety of programs in this language, we will often need to assume that a value resides in a particular region only. For example, after the copy function is finished, we must be able to assume that all the data is contained only in the new region; so that the old region can be safely freed. Therefore, to ensure type safety, we must be able to enforce this invariant at the type level. For this, we use the built-in type operator M. The type $\mathsf{M}_\rho(\tau)$ can only contain values that are in region ρ. This should be contrasted with values of type $\sigma\ \mathsf{at}\ \rho$ which can contain references to other regions that ρ.

6.3 The term calculus

The term language must support region based memory management and runtime type analysis. New regions are created through the $\mathsf{let\ region}\ r\ \mathsf{in}\ e$ construct which allocates a new region ν at runtime and binds r to it. A term of the form $\mathsf{put}[\rho]v$ allocates a value v in the region ρ. Data is read from a region in two ways. Functions are read implicitly through a function call. Data may also be read through the $\mathsf{get}\ v$ construct.

Operationally, get takes a memory address $\nu.\ell$ and dereferences it. Since our region calculus does not admit dangling references, and since each reference implicitly carries a region handle, get does not need a region argument, as opposed to put.

Deallocation is handled implicitly through the $\mathsf{only}\ \Delta\ \mathsf{in}\ e$ construct [23]. It asserts statically that the expression e can be evaluated using only the set of regions in Δ' (i.e. Δ extended with the cd region), which is a subset of the regions currently in scope. At runtime, an implementation would treat the set of regions in Δ' as live and reclaim all other regions.

$$\frac{\Delta' = \Delta, \mathsf{cd} \quad \Psi|_{\Delta'}; \Delta'; \Theta; \Gamma|_{\Delta'} \vdash e \quad \Delta' \subset \Delta''}{\Psi; \Delta''; \Theta; \Gamma \vdash \mathsf{only}\ \Delta\ \mathsf{in}\ e}$$

The construct $|_{\Delta'}$ restricts an environment to the set of regions in Δ'. I.e. $\Psi|_{\Delta'}$ is the subset of the heap restricted to the regions in Δ'. Similarly, $\Gamma|_{\Delta'}$ eliminates from Γ all variables whose type refers to regions not mentioned in Δ'.

The use of only was chosen for its simplicity. Other approaches either do not work with a CPS language or carry a significant added complexity to handle the problem of aliasing whereby the program requests the deletion of region r_1 while r_2 is still in use and might refer to the same region. only side steps this difficulty by making the actual deletion implicit: instead of requesting deletion of r_1, the program will request to keep r_2 which will prevent r_1 from being deleted if r_1 and r_2 happen to be aliases.

The runtime type analysis is handled through a typecase construct (it should arguably be called tagcase since it analyses tags rather than types). Depending on the head of the tag being analyzed, typecase chooses one of the branches for execution. When analyzing a tag variable t, we refine types containing t in each of the branches [6].

$$\frac{\begin{array}{l}\Theta \vdash t : \Omega \\ \Psi; \Delta; \Theta; \Gamma[\mathsf{int}/t] \vdash e_i[\mathsf{int}/t] \\ \ldots\end{array}}{\Psi; \Delta; \Theta; \Gamma \vdash \mathsf{typecase}\ t\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\rightarrow}; t_1t_2.e_{\times}; t_e.e_{\exists})}$$

In the e_i branch, we know that the tag variable t is bound to Int and can therefore substitute it away. A similar rule is applied to the other cases.

$\boxed{\Theta \vdash \tau : \kappa}$

$$\frac{}{\cdot \vdash \mathsf{Int} : \Omega} \qquad \frac{\Theta(t) = \kappa}{\Theta \vdash t : \kappa} \qquad \frac{\Theta \vdash \tau_1 : \Omega \quad \Theta \vdash \tau_2 : \Omega}{\Theta \vdash \tau_1 \times \tau_2 : \Omega}$$

$$\frac{\Theta \vdash \tau_i : \Omega}{\Theta \vdash \vec{\tau} \to 0 : \Omega} \qquad \frac{\Theta, t:\Omega \vdash \tau : \Omega}{\Theta \vdash \exists t.\tau : \Omega}$$

$$\frac{\Theta, t:\Omega \vdash \tau : \Omega}{\Theta \vdash \lambda t.\tau : \Omega \to \Omega} \qquad \frac{\Theta \vdash \tau_1 : \Omega \to \Omega \qquad \Theta \vdash \tau_2 : \Omega}{\Theta \vdash \tau_1 \tau_2 : \Omega}$$

$\boxed{\Delta; \Theta \vdash \sigma}$

$$\frac{}{\Delta; \Theta \vdash \mathsf{int}} \qquad \frac{\Delta; \Theta \vdash \sigma_1 \qquad \Delta; \Theta \vdash \sigma_2}{\Delta; \Theta \vdash \sigma_1 \times \sigma_2}$$

$$\frac{\{\vec{r}\}; t:\vec{\kappa} \vdash \sigma_i}{\Delta; \Theta \vdash \forall[t:\vec{\kappa}][\vec{r}](\vec{\sigma}) \to 0} \qquad \frac{\Delta; \Theta, t:\kappa \vdash \sigma}{\Delta; \Theta \vdash \exists t:\kappa.\sigma}$$

$$\frac{\Delta; \Theta \vdash \sigma \qquad \rho \in \Delta}{\Delta; \Theta \vdash \sigma \ \mathsf{at}\ \rho} \qquad \frac{\Theta \vdash \tau : \Omega \qquad \rho \in \Delta}{\Delta; \Theta \vdash \mathsf{M}_\rho(\tau)}$$

$\boxed{\Psi; \Delta; \Theta; \Gamma \vdash v : \sigma \qquad \Psi; \Delta; \Theta; \Gamma \vdash op : \sigma}$

$$\frac{}{\Psi; \Delta; \Theta; \Gamma \vdash n : \mathsf{int}} \qquad \frac{\Gamma(x) = \sigma}{\Psi; \Delta; \Theta; \Gamma \vdash x : \sigma}$$

$$\frac{\Psi(\nu.\ell) = \sigma \quad Dom(\Psi); \cdot \vdash \sigma \ \mathsf{at}\ \nu}{\Psi; \Delta; \Theta; \Gamma \vdash \nu.\ell : \sigma \ \mathsf{at}\ \nu}$$

$$\frac{\mathsf{cd}, \vec{r}; \vec{t} \vdash \sigma_i \qquad \Psi|_{\mathsf{cd}}; \mathsf{cd}, \vec{r}; \overrightarrow{t:\kappa}; \overrightarrow{x:\sigma} \vdash e}{\Psi; \Delta; \Theta; \Gamma \vdash \lambda[t:\vec{\kappa}][\vec{r}](\overrightarrow{x:\sigma}).e : \forall[t:\vec{\kappa}][\vec{r}](\vec{\sigma}) \to 0}$$

$$\frac{\Psi; \Delta; \Theta; \Gamma \vdash v_1 : \sigma_1 \qquad \Psi; \Delta; \Theta; \Gamma \vdash v_2 : \sigma_2}{\Psi; \Delta; \Theta; \Gamma \vdash (v_1, v_2) : \sigma_1 \times \sigma_2}$$

$$\frac{\Psi; \Delta; \Theta; \Gamma \vdash v : \sigma_1 \times \sigma_2}{\Psi; \Delta; \Theta; \Gamma \vdash \pi_i v : \sigma_i} \qquad \frac{\Psi; \Delta; \Theta; \Gamma \vdash v : \sigma \ \mathsf{at}\ \rho}{\Psi; \Delta; \Theta; \Gamma \vdash \mathsf{get}\ v : \sigma}$$

$$\frac{\Theta \vdash \tau : \kappa \quad \Psi; \Delta; \Theta; \Gamma \vdash v : \sigma[\tau/t]}{\Psi; \Delta; \Theta; \Gamma \vdash \langle t = \tau, v : \sigma \rangle : \exists t:\kappa.\sigma}$$

$$\frac{\Psi; \Delta; \Theta; \Gamma \vdash v : \sigma \qquad \rho \in \Delta}{\Psi; \Delta; \Theta; \Gamma \vdash \mathsf{put}[\rho] v : \sigma \ \mathsf{at}\ \rho}$$

$\boxed{\Psi; \Delta; \Theta; \Gamma \vdash e}$

$$\frac{\Psi; \Delta; \Theta; \Gamma \vdash v : \forall[t:\vec{\kappa}][\vec{r}](\vec{\sigma}) \to 0 \ \mathsf{at}\ \rho \qquad \Psi; \Delta; \Theta; \Gamma \vdash v_i : \sigma_i[\vec{\rho}, \vec{\tau}/\vec{r}, \vec{t}] \qquad \Theta \vdash \tau_i : \kappa_i \qquad \rho_i \in \Delta}{\Psi; \Delta; \Theta; \Gamma \vdash v[\vec{r}][\vec{\rho}](\vec{v})}$$

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

$$\frac{\Psi; \Delta; \Theta; \Gamma \vdash v : \exists t':\kappa.\sigma \quad \Psi; \Delta; \Theta, t:\kappa; \Gamma, x:\sigma[t/t'] \vdash e}{\Psi; \Delta; \Theta; \Gamma \vdash \mathsf{open}\ v\ \mathsf{as}\ \langle t, x \rangle\ \mathsf{in}\ e}$$

$$\frac{\Psi; \Delta; \Theta; \Gamma \vdash e_1 \quad \Psi; \Delta; \Theta; \Gamma \vdash e_2 \quad \rho \in \Delta}{\Psi; \Delta; \Theta; \Gamma \vdash \mathsf{ifgc}\ \rho\ e_1\ e_2}$$

$$\frac{\Psi; \Delta, r; \Theta; \Gamma \vdash e}{\Psi; \Delta; \Theta; \Gamma \vdash \mathsf{let\ region}\ r\ \mathsf{in}\ e} \qquad \frac{\Psi; \Delta; \Theta; \Gamma \vdash v : \mathsf{int}}{\Psi; \Delta; \Theta; \Gamma \vdash \mathsf{halt}\ v}$$

$$\frac{\Delta' = \Delta'', \mathsf{cd} \quad \Psi|_{\Delta'}; \Delta'; \Theta; \Gamma|_{\Delta'} \vdash e \quad \Delta' \subset \Delta}{\Psi; \Delta; \Theta; \Gamma \vdash \mathsf{only}\ \Delta''\ \mathsf{in}\ e}$$

$$\frac{\Psi; \Delta; \Theta; \Gamma \vdash e_{\mathsf{i}}}{\Psi; \Delta; \Theta; \Gamma \vdash \mathsf{typecase}\ \mathsf{Int}\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\to}; t_1 t_2.e_{\times}; t_e.e_{\exists})}$$

$$\frac{\Psi; \Delta; \Theta; \Gamma \vdash e_{\to}}{\Psi; \Delta; \Theta; \Gamma \vdash \mathsf{typecase}\ \vec{\tau} \to 0\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\to}; t_1 t_2.e_{\times}; t_e.e_{\exists})}$$

$$\frac{\Psi; \Delta; \Theta; \Gamma \vdash e_{\times}[\tau_1, \tau_2/t_1, t_2]}{\Psi; \Delta; \Theta; \Gamma \vdash \mathsf{typecase}\ (\tau_1 \times \tau_2)\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\to}; t_1 t_2.e_{\times}; t_e.e_{\exists})}$$

$$\frac{\Psi; \Delta; \Theta; \Gamma \vdash e_{\exists}[\lambda t.\tau/t_e]}{\Psi; \Delta; \Theta; \Gamma \vdash \mathsf{typecase}\ \exists t.\tau\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\to}; t_1 t_2.e_{\times}; t_e.e_{\exists})}$$

$$\frac{\begin{array}{l}\Theta \vdash t : \Omega \\ \Psi; \Delta; \Theta; \Gamma[\mathsf{int}/t] \vdash e_{\mathsf{i}}[\mathsf{int}/t] \\ \Psi; \Delta; \Theta; \Gamma \vdash e_{\to} \\ \Psi; \Delta; \Theta, t_1:\Omega, t_2:\Omega; \Gamma[t_1 \times t_2/t] \vdash e_{\times}[t_1 \times t_2/t] \\ \Psi; \Delta; \Theta, t_e:\Omega \to \Omega; \Gamma[\exists t.t_e t/t] \vdash e_{\exists}[\exists t.t_e t/t]\end{array}}{\Psi; \Delta; \Theta; \Gamma \vdash \mathsf{typecase}\ t\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\to}; t_1 t_2.e_{\times}; t_e.e_{\exists})}$$

Figure 7: Static semantics of λ_{GC}.

6.4 Formal properties of the language

In this section, we show that type checking in λ_{GC} is decidable and that the calculus is sound. We omit the proofs due to space constraints. The reader may refer to the companion technical report [11] for details.

Proposition 6.1 *Reduction of well formed types is strongly normalizing.*

Proposition 6.2 *Reduction of well formed types is confluent.*

Definition 6.3 *The judgment $\vdash (M, e)$ says that the machine state (M, e) is well-formed. It is defined by:*

$$\frac{\vdash M : \Psi \qquad \Psi; Dom(\Psi); \cdot; \cdot \vdash e}{\vdash (M, e)}$$

Contrary to the other environments, Ψ is not explicitly constructed in any of the static rules, since it reflects dynamic information. Instead, the soundness proof, or more specifically the type preservation proof, needs to construct some witness Ψ' for the new state (M', e') based on the Ψ of the initial state (M, e).

Proposition 6.4 (Type Preservation) *If $\vdash (M, e)$ and $(M, e) \Longrightarrow (M', e')$ then $\vdash (M', e')$.*

Proposition 6.5 (Progress) *If $\vdash (M, e)$ then either $e = \mathsf{halt}\ v$ or there exists a (M', e') such that $(M, e) \Longrightarrow (M', e')$.*

$$\boxed{\Psi;\Delta;\Theta;\Gamma \vdash v : \sigma \qquad \Psi;\Delta;\Theta;\Gamma \vdash e}$$

$$\frac{\Psi;\Delta;\Theta;\Gamma \vdash v : \mathsf{left}\ \sigma}{\Psi;\Delta;\Theta;\Gamma \vdash \mathsf{strip}\ v : \sigma} \quad \frac{\Psi;\Delta;\Theta;\Gamma \vdash v : \mathsf{right}\ \sigma}{\Psi;\Delta;\Theta;\Gamma \vdash \mathsf{strip}\ v : \sigma}$$

$$\frac{\Psi;\Delta;\Theta;\Gamma \vdash v : \sigma}{\Psi;\Delta;\Theta;\Gamma \vdash \mathsf{inl}\ v : \mathsf{left}\ \sigma} \quad \frac{\Psi;\Delta;\Theta;\Gamma \vdash v : \sigma}{\Psi;\Delta;\Theta;\Gamma \vdash \mathsf{inr}\ v : \mathsf{right}\ \sigma}$$

$$\frac{\Psi;\Delta;\Theta;\Gamma \vdash v : \mathsf{left}\ \sigma_1 \quad \Delta;\Theta \vdash \mathsf{right}\ \sigma_2}{\Psi;\Delta;\Theta;\Gamma \vdash v : \mathsf{left}\ \sigma_1 + \mathsf{right}\ \sigma_2}$$

$$\frac{\begin{array}{c}\Psi;\Delta;\Theta;\Gamma \vdash v : \sigma_1 + \sigma_2 \\ \Psi;\Delta;\Theta;\Gamma, x:\sigma_1 \vdash e_l \quad \Psi;\Delta;\Theta;\Gamma, x:\sigma_2 \vdash e_r\end{array}}{\Psi;\Delta;\Theta;\Gamma \vdash \mathsf{ifleft}\ x = v\ e_l\ e_r}$$

$$\frac{\begin{array}{c}\Psi;\Delta;\Theta;\Gamma \vdash e \\ \Psi;\Delta;\Theta;\Gamma \vdash v_1 : \sigma\ \mathsf{at}\ \rho \quad \Psi;\Delta;\Theta;\Gamma \vdash v_2 : \sigma\end{array}}{\Psi;\Delta;\Theta;\Gamma \vdash \mathsf{set}\ v_1 := v_2\ ;\ e}$$

$$\frac{\begin{array}{c}\Psi;\Delta;\Theta;\Gamma \vdash v : \mathsf{M}_\rho(\tau) \\ \Psi|_{\mathsf{cd}};\mathsf{cd},\rho,\rho';\Theta|_{\rho\rho'};x:\mathsf{C}_{\rho,\rho'}(\tau) \vdash e\end{array}}{\Psi;\Delta;\Theta;\Gamma \vdash \mathsf{let}\ x = \mathsf{widen}[\rho'][\tau](v)\ \mathsf{in}\ e}$$

Figure 8: Static semantics for $\lambda_{\mathsf{GCforw}}$.

7. FORWARDING POINTERS

The base algorithm presented before is unrealistic in a number of ways. The first is the fact that the $copy$ function does not preserve sharing and thus turns any DAG into a tree.

We hence need to add some form of *forwarding pointers*. Wang and Appel [25] propose to pair up every object with its forwarding pointer, incurring a significant memory cost. This additional word is not unheard of, since replicating garbage collectors [16, 1] incur a similar overhead, justified by the desire to provide concurrent collection while avoiding the cost of a read-barrier. We want instead to represent objects as a sum $(\sigma + \mathsf{fwd}\ \sigma)$, which requires a single bit per object and corresponds much more closely to traditional implementations. To this end, $\lambda_{\mathsf{GCforw}}$ extends λ_{GC} with new types and terms for tag bits and sum types as well as memory assignment. We do not need a new fwd or ref type since we can use the region calculus' references for that purpose.

Another requirement for a realistic GC is that the mutator should not need to constantly check for the presence of forwarding pointers since such a read-barrier would only be justified for an incremental GC. In other words, the type as seen by the mutator should not be a sum, although it should still contain the single-bit tag that the GC will use to distinguish between forwarding pointers. Also there should be a way to switch from the mutator's view of the type of an object to the one of the collector. So we also need to add a form of cast that we call widen which we will use at the beginning of a collection to give the collector access to the forwarding pointers:

$$\begin{array}{rl}\sigma & ::= \ldots \mid \mathsf{left}\ \sigma \mid \mathsf{right}\ \sigma \mid \mathsf{left}\ \sigma_1 + \mathsf{right}\ \sigma_2 \mid \mathsf{C}_{\rho,\rho'}(\tau) \\ v & ::= \ldots \mid \mathsf{inl}\ v \mid \mathsf{inr}\ v \\ op & ::= \ldots \mid \mathsf{strip}\ v \\ e & ::= \ldots \mid \mathsf{ifleft}\ x = v\ e_l\ e_r \mid \mathsf{set}\ v_1 := v_2\ ;\ e \\ & \quad \mid \mathsf{let}\ x = \mathsf{widen}[\rho][\tau](v)\ \mathsf{in}\ e\end{array}$$

Figure 8 shows the relevant new rules of the static semantics. Here, inl and inr (and their type-level counterparts left and right) can be thought of as adding a single tag bit to an object while strip gets back the untagged object and ifleft checks the tag bit. The idea is to represent objects as $(\mathsf{left}\ \sigma)$ to the mutator (and without the "$\mathsf{right}\ \sigma$" alternative to avoid the need for checks) and cast them with the widen operator to $(\mathsf{left}\ \sigma + \mathsf{right}(\sigma\ \mathsf{at}\ to))$ when entering the garbage collector (here "to" denotes the region for the to-space).

Since a single source-level type now maps to two different possible types, we need two type operators: $\mathsf{M}_\rho(\tau)$ to map source types to the mutator's view of the data and $\mathsf{C}_{\rho,\rho'}(\tau)$ to map source types to the collector's view (which adds forwarding pointers). $\mathsf{M}_\rho(\tau)$ is the same as before for base types and for code types, but is changed for existentials and pairs by adding the left constructor that constrains the mutator to provide the tag bit needed to distinguish the forwarded pointer from the non-forwarded data.

$$\begin{array}{ll}\mathsf{M}_\rho(\mathsf{Int}) & \Longrightarrow \mathsf{int} \\ \mathsf{M}_\rho(\tau \to 0) & \Longrightarrow \forall[][r](\mathsf{M}_r(\tau)) \to 0\ \mathsf{at}\ \mathsf{cd} \\ \mathsf{M}_\rho(\exists t.\tau) & \Longrightarrow (\mathsf{left}(\exists t.\mathsf{M}_\rho(\tau)))\ \mathsf{at}\ \rho \\ \mathsf{M}_\rho(\tau_1 \times \tau_2) & \Longrightarrow (\mathsf{left}(\mathsf{M}_\rho(\tau_1) \times \mathsf{M}_\rho(\tau_2)))\ \mathsf{at}\ \rho \\ \\ \mathsf{C}_{\rho,\rho'}(\mathsf{Int}) & \Longrightarrow \mathsf{int} \\ \mathsf{C}_{\rho,\rho'}(\tau \to 0) & \Longrightarrow \mathsf{M}_\rho(\tau \to 0) \\ \mathsf{C}_{\rho,\rho'}(\exists t.\tau) & \Longrightarrow (\mathsf{left}(\exists t.\mathsf{C}_{\rho,\rho'}(\tau)) + \mathsf{right}(\mathsf{M}_{\rho'}(\exists t.\tau)))\ \mathsf{at}\ \rho \\ \\ \mathsf{C}_{\rho,\rho'}(\tau_1 \times \tau_2) & \Longrightarrow (+^{\mathsf{left}(\mathsf{C}_{\rho,\rho'}(\tau_1) \times \mathsf{C}_{\rho,\rho'}(\tau_2))}_{\mathsf{right}(\mathsf{M}_{\rho'}(\tau_1 \times \tau_2))})\ \mathsf{at}\ \rho\end{array}$$

It is worth noting again here how the M type operators cleanly encapsulate the invariants imposed on the mutator by the collector. In this case, it forces the mutator to provide the collector with free bit that the collector can then use to distinguish forwarding pointers from non-forwarded data. And we also see how the same mechanism can be used to express the difference between the restricted view offered to the mutator and the full blown access to internal data that the collector needs.

The operational semantics of the new operations is straightforward, especially since we can implement the assignment operator by reusing the indirection through the memory:

$$\begin{array}{ll}(M, \mathsf{let}\ x = \mathsf{strip}\ (\mathsf{inl}\ v)\ \mathsf{in}\ e) & \Longrightarrow (M, e[v/x]) \\ (M, \mathsf{let}\ x = \mathsf{strip}\ (\mathsf{inr}\ v)\ \mathsf{in}\ e) & \Longrightarrow (M, e[v/x]) \\ (M, \mathsf{ifleft}\ x = (\mathsf{inl}\ v)\ e_l\ e_r) & \Longrightarrow (M, e_l[\mathsf{inl}\ v/x]) \\ (M, \mathsf{ifleft}\ x = (\mathsf{inr}\ v)\ e_l\ e_r) & \Longrightarrow (M, e_r[\mathsf{inr}\ v/x]) \\ (M, \mathsf{set}\ \nu.\ell := v\ ;\ e) & \Longrightarrow (M\{\nu.\ell \mapsto v\}, e) \\ (M, \mathsf{let}\ x = \mathsf{widen}[\rho][\tau](v)\ \mathsf{in}\ e) & \Longrightarrow (M, e[v/x])\end{array}$$

The translation from λ_{CLOS} to this $\lambda_{\mathsf{GCforw}}$ is not shown since it is basically the same as before except for the insertion of all the inl and strip. The garbage collector can be seen in Fig. 9. Compared to the original algorithm, the only difference in the gc function itself is the widening of the heap from M_{r1} to $\mathsf{C}_{r1,r2}$ and the fact that we have to bundle the f and x arguments into a pair in order to pass it through the widen operator and unbundle it afterwards. The $copy$ function also needs to be changed of course: when copying a heap object such as a pair, it now has to check with ifleft whether the object was forwarded, if so it just returns the forwarded object, otherwise it does the copy as before and has to overwrite (using set) the original object with the forwarding pointer before returning the copied object. Since the copy is still fundamentally depth-first, it will loop indefinitely in the presence of a cycle. Our source calculus cannot create cycles in the heap, but for more realistic languages, this is a significant restriction.

```
fix gc[t][r1](f:M_r1(t → 0), x:M_r1(t)).
  let region r2 in
  let w = widen[r2][(t → 0 × t)](put[r1](inl (f, x))) in
  ifleft w = get w then
    let w = strip w in
    let y = copy[t][r1, r2](π2 w) in
    only {r2} in (π1 w)[][r2](y)
  else
    halt 0

fix copy[t][r1, r2](x:C_r1,r2(t)) : M_r2(t).
  typecase t of
    Int     ⇒ x
    →       ⇒ x
    t1 × t2 ⇒ let y = get x in
              ifleft y = y then
                let x1 = copy[t1][r1, r2](π1(strip y)) in
                let x2 = copy[t2][r1, r2](π2(strip y)) in
                let z = put[r2](inl (x1, x2)) in
                set x := inr z ; z
              else
                strip y
    ∃te     ⇒ let y = get x in
              ifleft y = y then
                open (strip y) as ⟨t, y⟩ in
                let y = copy[te t][r1, r2](y) in
                let z = put[r2](inl ⟨t=t, y:M_r2(te t)⟩) in
                set x := inr z ; z
              else
                strip y
```

Figure 9: GC with forwarding pointers.

7.1 How to widen safely

The only non-trivial extension is widen which allows the garbage collector to have a different view of the existing memory, provided the two views are somehow *compatible*. It seems difficult to solve the problem of allowing two views on the same data without such a form of cast. At first, it seems we are just applying a form of subtyping, but this form of subtyping is very powerful since it allows covariant subtyping of references. This means that aliasing issues have to be handled with extreme care.

When faced with the same problem, Wang and Appel came up independently with a similar idea. But their suggested cast leaves many questions open and might need more work to be made type-safe. Also its operational semantics actually does a complete copy of the heap from one region to the other. This might make it easier to prove soundness but makes it unclear whether it can really be implemented as a nop. In contrast, the operational semantics of widen is a nop and we have a proof of its soundness.

In order to handle the problem of aliasing mentioned above, it might be possible to rely on some form of linear typing or alias types [22], but given the inherent generality of a garbage collector, it seems difficult. Our approach is to rely on the consistent application of the same cast over the whole heap, so that aliases are guaranteed to be cast in the same way (or discarded, since all free variables are thrown away).

Rather than an ad-hoc widen we could provide a more general cast that consistently applies a given type transformations (as long as it obeys the notion of subtyping extended with covariant subtyping of references) to any particular set of regions, but the complexity of such an operator is out of the scope of this paper.

In Figure 8, the typing rule for widen shows that the expression e is typed in an environment that only contains x. In essence x represents the entire heap. Further, x is obtained from the value v that has type $\mathsf{M}_\rho(\tau)$. Looking at the definition of M, we can see that all values reachable from v will have a type of the form $\mathsf{M}_\rho(\tau')$. Since both M and C are iterators, we can now define a casting operation from one type to the other as an iterator. This iterator will traverse the entire heap and systematically convert from one type to the other; this systematic conversion is necessary to avoid ending up with a value that has a particular type along one path, but has a different type along another path.

The proof of soundness of widen is rather intricate. It starts by ignoring some dead objects from the heap, so that only objects of type $\mathsf{M}_\rho(\tau)$ are left, which get cast to $\mathsf{C}_{\rho,\rho'}(\tau)$. For that reason, we need to loosen our notion of a well formed machine state to allow restricting the considered memory M to just a well-typed sufficient subset $\overline{M}$, where "sufficient" means that no object outside of $\overline{M}$ is needed to complete execution. This safely permits ill-typed garbage.

Definition 7.1 *The machine state (M, e) is well formed iff*

$$\frac{\overline{M} \subset M \quad \vdash \overline{M} : \Psi \quad \Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash e}{\vdash (M, e)}$$

The main stumbling block in the soundness proof is to show type preservation for widen. The proof begins by constructing $\overline{M}$ which only contains objects whose type matches some $\mathsf{M}_\rho(\tau)$. The resulting state is still well formed since all live data is of such a type when we reach widen. We then cast every heap object from its $\mathsf{M}_\rho(\tau)$ type to $\mathsf{C}_{\rho,\rho'}(\tau)$ and show, using the subsumption rule on sum types, that the resulting state is also well-formed and that it corresponds to the state after widen. See the companion TR [11] for the complete soundness proof.

8. GENERATIONAL COLLECTION

Another important aspect of a modern GC is the support for generational garbage collection. If we first restrict ourselves to a side-effect free language, then we can collect a single generation at a time so long as we can express the fact that an object in the old generation cannot point to an object in the young generation.

To that end we need to extend λ_{GC} with existential quantification over regions, so that the mutator does not need to care whether an object is allocated in the young or the old region. We also need to add some way to check in which region an object is allocated so that GC can detect when an object is in the old generation (and hence does not need copying):

$$\begin{array}{l} \sigma ::= \ldots \mid \exists r \in \Delta.(\sigma \text{ at } r) \\ v ::= \ldots \mid \langle r \in \Delta = \rho, v : \sigma \rangle \\ e ::= \ldots \mid \text{open } v \text{ as } \langle r, x \rangle \text{ in } e \mid \text{ifreg } (\rho_1 = \rho_2)\ e_1\ e_2 \end{array}$$

Apart from those new constructs (whose static semantics is presented in Fig. 10), the M type operator also needs to be modified to reflect the new invariant imposed on the mutator. It is now indexed by two regions (the old and the new) and has to enforce the fact that objects in the old region cannot have references to the new region:

$$\begin{array}{ll} \mathsf{M}_{\rho_y,\rho_o}(\text{Int}) & \Longrightarrow \text{int} \\ \mathsf{M}_{\rho_y,\rho_o}(\tau \to 0) & \Longrightarrow \forall[][r_y, r_o](\mathsf{M}_{r_y,r_o}(\tau)) \to 0 \text{ at cd} \\ \mathsf{M}_{\rho_y,\rho_o}(\exists t.\tau) & \Longrightarrow \exists r \in \{\rho_y, \rho_o\}.((\exists t.\mathsf{M}_{r,\rho_o}(\tau)) \text{ at } r) \\ \mathsf{M}_{\rho_y,\rho_o}(\tau_1 \times \tau_2) & \Longrightarrow \exists r \in \{\rho_y, \rho_o\}.((\mathsf{M}_{r,\rho_o}(\tau_1) \times \mathsf{M}_{r,\rho_o}(\tau_2)) \text{ at } r) \end{array}$$

$$\boxed{\Delta;\Theta \vdash \sigma \qquad \Psi;\Delta;\Theta;\Gamma \vdash v : \sigma \qquad \Psi;\Delta;\Theta;\Gamma \vdash e}$$

$$\frac{\Delta' \subset \Delta \quad \Delta, r;\Theta \vdash \sigma}{\Delta;\Theta \vdash \exists r \in \Delta'.(\sigma \text{ at } r)} \qquad \frac{\Theta \vdash \tau : \Omega \quad \rho_1 \in \Delta \quad \rho_2 \in \Delta}{\Delta;\Theta \vdash \mathsf{M}_{\rho_1,\rho_2}(\tau)}$$

$$\frac{\Psi;\Delta;\Theta;\Gamma \vdash v : \sigma[\rho/r] \text{ at } \rho \quad \rho \in \Delta' \quad \Delta' \subset \Delta}{\Psi;\Delta;\Theta;\Gamma \vdash \langle r \in \Delta' = \rho, v{:}\sigma\rangle : \exists r \in \Delta'.(\sigma \text{ at } r)}$$

$$\frac{\Psi;\Delta;\Theta;\Gamma \vdash v : \exists r \in \Delta'.(\sigma \text{ at } r) \qquad \Psi;\Delta,r;\Theta;\Gamma,x{:}\sigma \text{ at } r \vdash e}{\Psi;\Delta;\Theta;\Gamma \vdash \mathsf{open}\ v\ \mathsf{as}\ \langle r,x\rangle\ \mathsf{in}\ e}$$

$$\frac{\Psi;\Delta[r,r/r_1,r_2];\Theta;\Gamma[r,r/r_1,r_2] \vdash e_1[r,r/r_1,r_2] \qquad \Psi;\Delta;\Theta;\Gamma \vdash e_2 \qquad r \notin \Delta}{\Psi;\Delta;\Theta;\Gamma \vdash \mathsf{ifreg}\ (r_1 = r_2)\ e_1\ e_2}$$

$$\frac{\Psi;\Delta[\nu/r];\Theta;\Gamma[\nu/r] \vdash e_1[\nu/r] \qquad \Psi;\Delta;\Theta;\Gamma \vdash e_2}{\Psi;\Delta;\Theta;\Gamma \vdash \mathsf{ifreg}\ (r = \nu)\ e_1\ e_2 \qquad \Psi;\Delta;\Theta;\Gamma \vdash \mathsf{ifreg}\ (\nu = r)\ e_1\ e_2}$$

$$\frac{\Psi;\Delta;\Theta;\Gamma \vdash e_2}{\Psi;\Delta;\Theta;\Gamma \vdash \mathsf{ifreg}\ (\nu_1 = \nu_2)\ e_1\ e_2}$$

$$\frac{\Psi;\Delta;\Theta;\Gamma \vdash e_1}{\Psi;\Delta;\Theta;\Gamma \vdash \mathsf{ifreg}\ (\nu_1 = \nu_1)\ e_1\ e_2}$$

Figure 10: Static semantics of λ_{GCgen}.

By using the set $\{r, \rho_o\}$ we make sure that if r is the old generation, pointers reachable from it cannot point back to the new generation.

The operational semantics are again rather simple:

$$\begin{array}{ll}(M, \mathsf{open}\ \langle r \in \Delta = \nu, v{:}\sigma\rangle\ \mathsf{as}\ \langle r,x\rangle\ \mathsf{in}\ e) & \Longrightarrow (M, e[\nu, v/r, x]) \\ (M, \mathsf{ifreg}\ (\nu = \nu)\ e_1\ e_2) & \Longrightarrow (M, e_1) \\ (M, \mathsf{ifreg}\ (\nu_1 = \nu_2)\ e_1\ e_2) & \Longrightarrow (M, e_2)\end{array}$$

Although the operational semantics do not take advantage of it (in order to simplify the soundness proof), we defined the existentials over regions in such a way that they can be implemented as nop since the encapsulated reference usually already encodes the region in its bit-pattern (or in its $\nu.\ell$).

The new term translation is again not shown since it is so similar to the original one. The new type constraint is trivially always satisfied as long as the mutator only allocates from the younger generation and as long as the memory is immutable. If side-effects were to be necessary, it should be possible to extend this scheme with one mutable region (keeping all others immutable) which would be considered similarly to the older generation but scanned at each collection. Obviously, this would first require adding some way to scan a region, but should not present any serious difficulty.

The GC itself can be seen in figure 11. The main difference with the basic GC of figure 4 is that it does not copy to a new region but to an existing one and stops traversing the tree as soon as we encounter a reference to the old generation.

When hitting such an external reference, we have to repack it just to help the type-system understand that this reference is of type $\mathsf{M}_{\rho_y,\rho_o}(\tau)$. But those operations are free anyway.

$\mathsf{fix}\ gc[t][r_y, r_o](f{:}\mathsf{M}_{r_y,r_o}(t \to 0), x{:}\mathsf{M}_{r_y,r_o}(t)).$
  $\mathsf{let}\ y = copy[t][r_y, r_o](x)\ \mathsf{in}$
  $\mathsf{only}\ \{r_o\}\ \mathsf{in}\ \mathsf{let\ region}\ r_y\ \mathsf{in}\ f[][r_y, r_o](y)$

$\mathsf{fix}\ copy[t][r_y, r_o](x{:}\mathsf{M}_{r_y,r_o}(t)) : \mathsf{M}_{r_o,r_o}(t).$
  $\mathsf{typecase}\ t\ \mathsf{of}$
    $\mathsf{Int} \Rightarrow x$
    $\to \Rightarrow x$
    $t_1 \times t_2 \Rightarrow \mathsf{open}\ x\ \mathsf{as}\ \langle r, x\rangle\ \mathsf{in}$
      $\mathsf{ifreg}\ r = r_o\ \mathsf{then}\ \langle r \in \{r_o\} = r_o, x\rangle\ \mathsf{else}$
        $\mathsf{let}\ x_1 = copy[t_1][r_y, r_o](\pi_1(\mathsf{get}\ x))\ \mathsf{in}$
        $\mathsf{let}\ x_2 = copy[t_2][r_y, r_o](\pi_2(\mathsf{get}\ x))\ \mathsf{in}$
        $\langle r \in \{r_o\} = r_o, \mathsf{put}[r](x_1, x_2)\rangle$
    $\exists t_e \Rightarrow \mathsf{open}\ x\ \mathsf{as}\ \langle r, x\rangle\ \mathsf{in}$
      $\mathsf{ifreg}\ r = r_o\ \mathsf{then}\ \langle r \in \{r_o\} = r_o, x\rangle\ \mathsf{else}$
        $\mathsf{open}\ (\mathsf{get}\ x)\ \mathsf{as}\ \langle t, y\rangle\ \mathsf{in}$
        $\mathsf{let}\ z = copy[t_e t][r_y, r_o](y)\ \mathsf{in}$
        $\langle r \in \{r_o\} = r_o, \mathsf{put}[r]\langle t{=}t, z{:}\mathsf{M}_{r,r_o}(t_e t)\rangle\rangle$

Figure 11: Generational GC.

Note that another function needs to be written to garbage collect the old generation, but that one is the same as the non-generational one.

At first sight, the λ_{GCgen} language may seem unsound because we allow existentials over regions. However, these types are not existentials in a real sense since they do not hide a region within a type. Rather, in the type $\exists r \in \Delta.(\sigma \text{ at } r)$, the set Δ is an upper bound on the regions that the variable r may range over. In this sense, our existential is closer to a bounded quantification.

See the companion TR [11] for the proof of soundness.

9. RELATED WORK

Wang and Appel [23] proposed to build a tracing garbage collector on top of a region-based calculus, thus providing both type safety and completely automatic memory management. The main weakness of their proposal is that it relies on a closure conversion algorithm due to Tolmach [20] that represents closures as datatypes. This makes closures transparent, making it easier for the copy function to analyze, but it requires whole program analysis and has major drawbacks in the presence of separate compilation. We believe it is more natural to represent closures as existentials [10, 9] and we show how to use intensional type analysis (on quantified types [21]) to typecheck the GC-copy function.

Intensional type analysis was first proposed by Harper and Morrisett [8]. They introduced the idea of having explicit type analysis operators which inductively traverse the structure of types. However, to retain decidability of type checking, they restrict the analysis to a predicative subset of the type language. Crary et al. [5] propose a very powerful type analysis framework. They define a rich kind calculus that includes sum kinds and inductive kinds. They also provide primitive recursion at the type level. Therefore, they can define new kinds within their calculus and directly encode type analysis operators within their language. They also include a novel refinement operation at the term level. Saha et al [21] shows how to handle polymorphic functions that analyze the quantified type variable—this allows the type analysis to handle arbitrary quantified types. The typerec operators (e.g., M_ρ) used in this paper do not require the full power of what is provided in [21] because our source language is only a simply typed lambda calculus.

Tofte and Talpin [19] proposed to use region calculus to type

check memory management for higher-order functional languages. Crary et al [4] presented a low-level typed intermediate language that can express explicit region allocation and deallocation. Our λ_{GC} language borrows the basic organization of memories and regions from Crary et al [4]. The main difference is that we don't require explicit capabilities—region deallocation is handled through the only primitive.

Necula and Lee [14, 15] proposed the idea of proof-carrying code and showed how to construct a certifying compiler for a type-safe subset of C. Morrisett et al. [13] showed that type-preserving compilation via typed assembly language is a good basis for building certifying compilers. This paper shows that low-level runtime services such as garbage collection can also be expressed in a type-safe language.

10. CONCLUSIONS AND FUTURE WORK

We have presented a type-safe intermediate language with regions and intensional type analysis and show how it can be used to provide a simple and provably type-safe stop-and-copy tracing garbage collector. Our key idea is to use intensional type analysis on quantified types (i.e., existentials) to express the garbage-collection invariants on the mutator data objects. We show how this same idea can be used to express more realistic scavengers with efficient forwarding pointers and generations. Because intensional type analysis is also applicable to polymorphic lambda calculus [21], we believe our type safe collector can be extended to handle polymorphic languages as well.

We intend to extend our collector with the following features, which a modern garbage collector should be able to provide:

- Polymorphism. Intensional type analysis is a powerful framework. Adding support for polymorphism is straightforward but tedious because the type system becomes a lot heavier.
- Cyclic data structures. It might be possible to extend the current depth-first copying approach to properly handle cycles, but we are more interested in a Cheney-style breadth-first copy [2].
- Side-effects and generations. A first approach could be to extend our current generation scheme with a third region containing all the mutable data. But ultimately we will need to use either card-marking or remembered-sets [26].
- Explicit tag storage. Since tags exist at run time, we need to garbage collect them as well. The most promising approach is to reify them into special terms as was done by Crary *et al* [6, 5]. This will also allow us to use a simpler closure conversion algorithm for polymorphic code, eliminating the need for translucent types.

REFERENCES

[1] G. E. Blelloch and P. Cheng. On bounding time and space for multiprocessor garbage collection. In *Proc. ACM SIGPLAN '99 Conf. on Prog. Lang. Design and Implementation*, pages 104–117, New York, 1999. ACM Press.

[2] C. J. Cheney. A non-recursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, 1970.

[3] K. Crary. Typed assembly language: Type theory for machine code. Talk presented at 2000 PCC Workshop, Santa Barbara, CA, June 2000.

[4] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Proc. Twenty-Sixth Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 262–275. ACM Press, 1999.

[5] K. Crary and S. Weirich. Flexible type analysis. In *Proc. 1999 ACM SIGPLAN International Conference on Functional Programming*, pages 233–248. ACM Press, Sept. 1999.

[6] K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type-erasure semantics. In *Proc. 1998 ACM SIGPLAN International Conference on Functional Programming*, pages 301–312. ACM Press, Sept. 1998.

[7] O. Danvy and A. Filinski. Representing control, a study of the cps transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[8] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-second Annual ACM Symp. on Principles of Prog. Languages*, pages 130–141, New York, Jan 1995. ACM Press.

[9] R. Harper and G. Morrisett. Typed closure conversion for recursively-defined functions. In *Second International Workshop on Higher Order Operational Techniques in Semantics (HOOTS98*, New York, Sep 1998. ACM Press.

[10] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Proc. 23rd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 271–283. ACM Press, 1996.

[11] S. Monnier, B. Saha, and Z. Shao. Principled scavenging. Technical Report YALEU/DCS/TR1205, Dept. of Computer Science, Yale University, New Haven, CT, November 2000.

[12] G. Morrisett. Open problems for certifying compilers. Talk presented at 2000 PCC Workshop, Santa Barbara, CA, June 2000.

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

[14] G. Necula. Proof-carrying code. In *Twenty-Fourth Annual ACM Symp. on Principles of Prog. Languages*, pages 106–119, New York, Jan 1997. ACM Press.

[15] G. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proc. ACM SIGPLAN '98 Conf. on Prog. Lang. Design and Implementation*, pages 333–344, New York, 1998. ACM Press.

[16] S. Nettles and J. O'Toole. Real-time replication garbage collection. In *Symposium on Programming Languages Design and Implementation*, 1993.

[17] B. Saha, V. Trifonov, and Z. Shao. Fully reflexive intensional type analysis. Technical Report YALEU/DCS/TR-1194, Dept. of Computer Science, Yale University, New Haven, CT, March 2000.

[18] Z. Shao, C. League, and S. Monnier. Implementing typed intermediate languages. In *Proc. 1998 ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 313–323, September 1998.

[19] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proc. 21st Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 188–201. ACM Press, 1994.

[20] A. Tolmach and D. P. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, July 1998.

[21] V. Trifonov, B. Saha, and Z. Shao. Fully reflexive intensional type analsysis. In *Proc. 2000 ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 82–93. ACM Press, September 2000.

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

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

[24] D. C. Wang and A. W. Appel. Type-preserving garbage collectors (extended version). Technical Report TR-624-00, Princeton University, 2000.

[25] D. C. Wang and A. W. Appel. Type-preserving garbage collectors. In *Proc. 28th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 166–178. ACM Press, 2001.

[26] P. Wilson. Uniprocessor garbage collection techniques. In *1992 International Workshop on Memory Management*, New York, June 1992. ACM Press.