# 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

YALEU/DCS/TR-1205  Nov 2000

## 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) [13] and Typed Assembly Languages (TAL) [12]—both of which aim to minimize the trusted computing base (TCB) by directly certifying the actual machine code. Unfortunately, these systems cannot get rid of the dependency on a trusted GC. Indeed, constructing a verifiably type-safe GC is widely considered as one of the major open problems in the area of certifying compilation [11, 2].

Recently, Wang and Appel [23] 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 contributions:

- We show how to use intensional type analysis (ITA) [19, 7] 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 (at least to some people), however, none of the previous work [21, 15, 19] have succeeded in getting it to work. Indeed, Wang and Appel [23] 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 [23] 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 functions and polymorphism.
- 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 [23] 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 Appendix).Wang and Appel's collectors [23, 22], 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 GCs in the FLINT system [16], however, making the implementation realistic still involves solving the remaining problems (e.g., breadth-first copying, remembered sets, and data structures with cycles, which we still cannot support satisfactorily) thus is 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.

## 2. Motivation and background

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

*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.

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 [13] and typed assembly languages [12] 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 [2] and Morrisett [11] 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 [21] proposed to tackle the problem by layering a stop-and-copy tracing garbage collector [24] on top of a region based calculus, thus providing both type safety and completely automatic memory management.

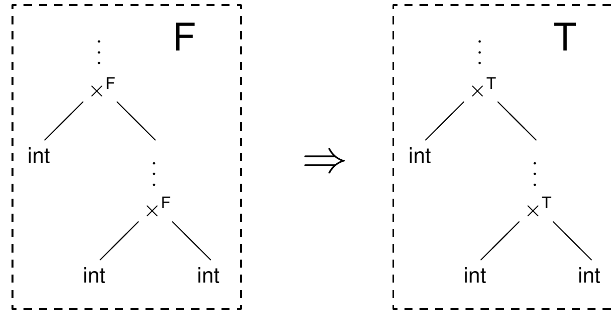
A region calculus [17] annotates the type of every heap object with the region in which it is allocated (such as $\sigma_1 \times^\rho \sigma_2$ where $\rho$ is the region), and thus allows to safely reclaim memory by freeing a whole region if that region does not appear in any of the currently live 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 $\alpha$ where the region annotation $\mathsf{T}$ is substituted for $\mathsf{F}$ (see Fig. 1). Assuming we have such a function and we don't keep any reference to the region $\mathsf{F}$, the region calculus will allow us to safely reclaim $\mathsf{F}$.

Clearly, there is no correctness guarantee in sight since the value returned by that *copy* function might have 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*.

In their followup paper [23], 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 [18]) to simplify the problem to a monomorphic first order calculus. However, this approach suffers from several major drawbacks:




$$
\begin{array}{l}
\mathit{copy} : \forall\mathsf{F}.\forall\mathsf{T}.\forall\alpha.(\alpha \longrightarrow \alpha[\mathsf{T}/\mathsf{F}])\\[4pt]
\mathit{GC} = \Lambda\mathsf{F}.\Lambda\alpha.\lambda(x\!:\!\alpha, k\!:\!\forall\rho.\alpha[\rho/\mathsf{F}] \rightarrow 0)\\
\qquad \mathsf{let\ region\ T\ in}\\
\qquad \mathsf{let}\ y = \mathit{copy}[\mathsf{F}][\mathsf{T}][\alpha](x)\ \mathsf{in}\\
\qquad \mathsf{only\ T\ in}\ k[\mathsf{T}](y)
\end{array}
$$

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

- Most importantly, it is not generally applicable and requires whole program analysis which rule out separate compilation.
- 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.
- 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 is done independently). It relies mostly on a very powerful form of *cast* which allows some amount of 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 return type of *copy* ($\alpha[\mathsf{T}/\mathsf{F}]$, a form of Typerec) as well as the need to observe types at runtime leads one very naturally to use intensional type analysis (ITA). In fact, an early paper of Wang and Appel [21] was titled "safe garbage collection = regions + intensional type analysis;" but they failed to make it work, and they subsequently gave up on using ITA and ended up opting for the lower-tech solution mentioned above [23]. Saha et al. [15, 19] 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 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 $\mathsf{Typerec}$ such as $\mathsf{S}_{\mathsf{T},\mathsf{F}}(\sigma)$ which substitutes $\mathsf{T}$ for $\mathsf{F}$ and then use $\mathsf{typecase}$ in the body of *copy*.

But that means that the type grows each time we go through the GC, from $\sigma$ 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 $\mathsf{S}$ should be reduced away anyway. But $\mathsf{S}_{\rho,\mathsf{T}}(\alpha)$ cannot be reduced further until $\alpha$ is instantiated: $\exists\alpha.\mathsf{S}_{\mathsf{T},\mathsf{F}}(\alpha)$ is a normal form. So the accumulation of $\mathsf{S}$ 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 $\mathsf{F}$ is the "initial region") except before the first collection. Also it is very ad-hoc and only works as long as $\mathsf{S}$ is a quasi-identity.

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

### 2.2.2 A case for tags

The above solution looks good until we try to copy an existential package $\exists\alpha:\{\mathsf{F}\}.\mathsf{S}_{\mathsf{F}}(\alpha)$ to $\exists\alpha:\{\mathsf{T}\}.\mathsf{S}_{\mathsf{T}}(\alpha)$.

Type variables hide region annotations, so we need to annotate their kinds with the relevant region information. We write it "$\alpha : \Delta$" to mean that $\alpha$ can only range over types that refer exclusively to the regions included in $\Delta$.

So, by opening the existential package, we can get the value $\sigma$ of $\alpha$ and the value of type $\mathsf{S}_{\mathsf{F}}(\sigma)$, and a recursive call to *copy* will return $\mathsf{S}_{\mathsf{T}}(\sigma)$, but how can we construct the new existential package ? Reusing $\sigma$ as-is will not do since $\sigma$ is not constrained to $\{\mathsf{T}\}$ but to $\{\mathsf{F}\}$. We would want to use $\mathsf{S}_{\mathsf{T}}(\sigma)$ but that 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 $\exists\alpha:\{\mathsf{T}\}.\alpha$.

We are again pushing a new $\mathsf{S}$ onto the type rather than replacing an $\mathsf{S}$ 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 $\sigma$ 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 $\tau$ (that we will call *tags*). Since tags have no region annotations, we can hide them in tag variables without any $\Delta$ constraint, which side-steps the problem of copying existentials conveniently.

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* [7, 4]. 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 $\mathsf{M}$ (formerly $\mathsf{S}$) 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 $\mathsf{M}$ 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 $\lambda_{CLOS}$

For simplicity of the presentation, the source language we propose to compile and garbage collect is the simply typed $\lambda$-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 won't go into the details of how to CPS convert that source language [6]. Similarly, for the closure conversion using existentials [9, 8].

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

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

Since functions are in CPS, they never return, which we represent with the arbitrary return type 0. To represent closures, the language includes existential packages constructed by $\langle t = \tau_1, v : \tau_2\rangle$ and of type $(\exists t.\tau_2)$. The $\mathsf{open}\ v\ \mathsf{as}\ \langle t, x\rangle\ \mathsf{in}\ e$ construct takes an existential package $v$, binds the witness type to $t$ and the value to $x$, and then executes $e$. The complete program consists of a list of mutually recursive closed function declarations followed by the main term to be executed.

# 4. Target language $\lambda_{GC}$

The language used to write the garbage-collector (and into which we translate $\lambda_{CLOS}$ programs) is shown in Fig. 2. It extends $\lambda_{CLOS}$ with regions [17] and intensional type analysis [7, 19]. 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 "references" denoted $\nu.\ell$ of type $(\sigma\ \mathsf{at}\ \rho)$ rather than annotations like $\sigma_1 \times^{\rho} \sigma_2$ (which is written $(\sigma_1 \times \sigma_2)\ \mathsf{at}\ \rho$ instead). Similarly, object allocation and memory accesses are made explicit with $\mathsf{put}$ and $\mathsf{get}$. This was only preferred because of its orthogonality.

Region allocation and reclamation is done with $\mathsf{let\ region}$ and $\mathsf{only}$. Deallocation of a region is implicit since $\mathsf{only}$ lists the regions that should be kept. This neatly works around aliasing problems, at the cost of a more expensive deallocation operation ($\mathsf{only}$ 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 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 different between $\lambda_{CLOS}$ types and $\lambda_{GC}$ tags is the addition of tag

$$
\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 \\
(\alpha\ \textit{env}) & \Phi & ::= \cdot \mid \Phi, \alpha\!:\!\Delta \\[4pt]
(\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\} \\[4pt]
(\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) \\[4pt]
(\textit{regions}) & \rho & ::= \nu \mid r \\
(\textit{kinds}) & \kappa & ::= \Omega \mid \Omega \to \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\!:\!\kappa}][\vec{r}](\vec{\sigma}) \to 0 \\
& & \quad \mid \exists t\!:\!\kappa.\sigma \mid \sigma \ \mathsf{at}\ \rho \mid \mathsf{M}_\rho(\tau) \mid \alpha \\
& & \quad \mid \forall[\![\vec{r}]\!][\vec{r}](\vec{\sigma}) \xrightarrow{\rho} 0 \mid \exists \alpha\!:\!\Delta.\sigma \\[4pt]
(\textit{values}) & v & ::= n \mid x \mid \nu.\ell \mid (v_1, v_2) \mid \langle t\!=\!\tau, v\!:\!\sigma\rangle \mid v[\![\vec{r}]\!] \\
& & \quad \mid \langle \alpha\!:\!\Delta\!=\!\sigma_1, v\!:\!\sigma_2\rangle \mid \lambda[\vec{t\!:\!\kappa}][\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{open}\ v\ \mathsf{as}\ \langle \alpha, x\rangle\ \mathsf{in}\ e \mid \mathsf{let\ region}\ r\ \mathsf{in}\ e \\
& & \quad \mid \mathsf{only}\ \Delta\ \mathsf{in}\ e \\
& & \quad \mid \mathsf{typecase}\ \tau\ \mathsf{of}\ (e_{\mathsf{i}}; e_\lambda; t_1 t_2.e_\times; t_e.e_\exists) \\[4pt]
(\textit{tagnf}) & \tau' & ::= t \mid \mathsf{Int} \mid \tau' \to 0 \mid \tau'_1 \times \tau'_2 \mid \exists t.\tau' \mid \lambda t.\tau' \\
& & \quad \mid t\tau'
\end{array}
$$

**Figure 2: Syntax of $\lambda_{GC}$**

functions $\lambda t.\tau$ and tag applications $\tau_1\tau_2$, which are needed for type analysis of existentials [19]. To do the actual analysis of tags, terms include a *refining* typecase construct, i.e. a more refined tag is substituted for $\tau$ 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 $\tau$ complemented with region annotations $\rho$:

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

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.

### 4.3 Functions and code

Since programs in $\lambda_{GC}$ are completely closed, we can separate code from data. The memory configuration enforces this by having a separate dedicated region cd for code blocks. The indirection provided by memory references allows us to do away with letrec. 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 $\lambda_{CLOS}$ and never directly appear in $\lambda_{GC}$ code.

## 5. Translating $\lambda_{CLOS}$ to $\lambda_{GC}$

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

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

$$
\frac{F \vdash_v v_1 \Rightarrow v'_1 \quad F \vdash_v v_2 \Rightarrow v'_2}{F \vdash_v (v_1, v_2) \Rightarrow \mathsf{put}[r](v'_1, v'_2)}
$$

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

$$
\frac{F \vdash_v v_1 \Rightarrow v'_1 \quad F \vdash_v v_2 \Rightarrow v'_2}{F \vdash_e v_1(v_2) \Rightarrow v'_1[\,][r](v'_2)}
\qquad
\frac{F \vdash_v v \Rightarrow v'}{F \vdash_e \mathsf{halt}\ v \Rightarrow \mathsf{halt}\ v'}
$$

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

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

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

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

$$
\frac{F = \{f_i \mapsto \ell_1, \ldots\} \quad F \vdash_f f_i = \lambda(x\!:\!\tau).e \Rightarrow f'_i \quad F \vdash_e e \Rightarrow e'}{\begin{array}{l}\vdash_p \mathsf{letrec}\ \overrightarrow{f = \lambda(x\!:\!\tau).e}\ \mathsf{in}\ e \\ \quad \Rightarrow\ (\{\mathsf{cd} \mapsto \{\ell_1 \mapsto f'_1, \ldots\}\}, \mathsf{let\ region}\ r\ \mathsf{in}\ e')\end{array}}
$$

**Figure 3: Translation of $\lambda_{CLOS}$ terms.**

The translation of terms from $\lambda_{CLOS}$ to $\lambda_{GC}$ shown in figure 3 is mostly directed by the type translation $\mathsf{M}_\rho$ presented earlier: each 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 $\lambda_{CLOS}$ function like:

$$
\begin{array}{l}
\mathsf{fix}\ swap(x\!:\!\mathsf{int} \times \mathsf{int}). \\
\quad \mathsf{let}\ x1 = \pi_1 x\ \mathsf{in}\ \mathsf{let}\ x2 = \pi_2 x\ \mathsf{in}\ \mathsf{let}\ x' = (x2, x1)\ \mathsf{in}\ \mathsf{halt}\ 0
\end{array}
$$

would turn into the following $\lambda_{GC}$ function (apart from some syntactic conveniences):

$$
\begin{array}{l}
\mathsf{fix}\ swap[\,][r](x\!:\!(\mathsf{Int} \times \mathsf{Int})\ \mathsf{at}\ r). \\
\quad \mathsf{ifgc}\ r\ (gc[\mathsf{Int} \times \mathsf{Int}][r](swap, x)) \\
\quad\quad \mathsf{let}\ x = \mathsf{get}\ x\ \mathsf{in} \\
\quad\quad \mathsf{let}\ x1 = \pi_1 x\ \mathsf{in} \\
\quad\quad \mathsf{let}\ x2 = \pi_2 x\ \mathsf{in} \\
\quad\quad \mathsf{let}\ x' = \mathsf{put}[r](x2, x1)\ \mathsf{in} \\
\quad\quad \mathsf{halt}\ 0
\end{array}
$$

An important detail here is that the garbage collector receives the tag $\tau$ rather than the type $\sigma$ of the argument. The garbage collector receives the tags for analysis as they were in $\lambda_{CLOS}$ rather than as they are translated in $\lambda_{GC}$. This maintains a clear distinction

$$
\begin{array}{l}
\text{fix } gc[t:\Omega][r_1](f:\forall[][r](\mathsf{M}_r(t)) \to 0, x:\mathsf{M}_{r_1}(t)). \\
\quad \text{let region } r_2 \text{ in} \\
\quad \text{let } y = copy[t][r_1,r_2](x) \text{ in} \\
\quad \text{only } \{r_2\} \text{ in } f[][r_2](y)
\end{array}
$$

$$
\begin{array}{l}
\text{fix } copy[t:\Omega][r_1,r_2](x:\mathsf{M}_{r_1}(t)) : \mathsf{M}_{r_2}(t). \\
\quad \text{typecase } t \text{ of} \\
\qquad \begin{array}{rl}
\text{int} & \Rightarrow x \\
\lambda & \Rightarrow x \\
t_1 \times t_2 & \Rightarrow \text{let } x_1 = copy[t_1][r_1,r_2](\pi_1(\text{get } x)) \text{ in} \\
& \quad\ \text{let } x_2 = copy[t_2][r_1,r_2](\pi_2(\text{get } x)) \text{ in} \\
& \quad\ \text{put}[r_2](x_1,x_2) \\
\exists t_e & \Rightarrow \text{open } (\text{get } x) \text{ as } \langle t,y\rangle \text{ in} \\
& \quad\ \text{let } z = copy[t_e t][r_1,r_2](y) \text{ in} \\
& \quad\ \text{put}[r_2]\langle t{=}t, z:\mathsf{M}_{r_2}(t_e t)\rangle
\end{array}
\end{array}
$$

**Figure 4: The garbage collector proper.**

between the types the programmer thinks he manipulates and the real 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 figure 3 uses $\lambda_{GC}$ in a somewhat loose way to keep the presentation concise. More specifically, it will generate terms such as let $x = \pi_i(\text{get } v)$ in $e$ instead of let $x' =$ get $v$ in let $x = \pi_i x'$ in $e$. Turning such code back into the strict $\lambda_{GC}$ is immediate.

On the other hand, the garbage collection code in figure 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. As can be seen in figure 12 in the appendix, the code after CPS and closure conversion is a lot more difficult to read, partly because of the need to do a form of typed closure conversion [9].

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 (as shown in the appendix), 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 $\lambda_{GC}$

Programs in $\lambda_{GC}$ 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 $\nu$ and regions $R$. A region, in turn, is a map from offsets $\ell$ 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; $\Upsilon$ denotes a region type. Finally, the memory type $\Psi$ assigns a region type to every region allocated in memory.

## 6.1 Closure conversion and *copy*.

Since the source language is monomorphic, closure conversion need only rely on existentials. This simplicity is however broken by the *copy* function in the garbage collector itself because this function is (recursively) polymorphic. For that reason, we also need a form of translucent type, namely $\forall[\![\vec{\tau}]\!][\vec{r}](\vec{\sigma}) \xrightarrow{\rho} 0$. Closure conversion of the CPS form of *copy* was also the only reason for introducing $\langle \alpha:\Delta{=}\sigma_1, v:\sigma_2\rangle$.

## 6.2 Functions and code

Since function bodies can contain references to other functions in cdbut 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 all that cd special casing, 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.

## 6.3 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 $\lambda_{GC}$ must support the runtime analysis of types. Therefore, conceptually, types need to play a dual role in this language. As in the source language $\lambda_{CLOS}$, 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.

While translating from $\lambda_{CLOS}$ to $\lambda_{GC}$, the tag for a value must be constructed from its type. Therefore, the tags in $\lambda_{GC}$, closely resemble the type language in the source. To support the analysis of these tags, we need to add tag level functions $(\lambda t.\tau)$ and tag level applications $(\tau\tau_1)$. In turn, this requires a kind calculus to classify the tags.

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

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 $\rho$. Notice that it is a restricted form of the fully reflexive Typerec operator [19]. Essentially, it is

$$
\begin{array}{ll}
(M, \nu.\ell[\vec{\tau}][\vec{\rho}](\vec{v})) & \Longrightarrow (M, \nu.\ell[\vec{\tau}'][\vec{\rho}](\vec{v})) \\
(M, \nu.\ell[\vec{\tau}'][\vec{\rho}](\vec{v})) & \\
\quad \text{where } M(\nu.\ell) = (\lambda[\vec{t:\kappa}][\vec{r}](\vec{x}:\vec{\sigma}).e) & \Longrightarrow (M, e[\vec{\rho}, \vec{\tau}', \vec{v}/\vec{r}, \vec{t}, \vec{x}]) \\
(M, (v[\![\vec{\tau}]\!])[\vec{\tau}][\vec{\rho}](\vec{v})) & \Longrightarrow (M, v[\vec{\tau}][\vec{\rho}](\vec{v})) \\
(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, \mathsf{open}\ \langle t = \tau', v : \sigma\rangle\ \mathsf{as}\ \langle t, x\rangle\ \mathsf{in}\ e) \\
(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{open}\ \langle \alpha : \Delta = \sigma_1, v : \sigma_2\rangle\ \mathsf{as}\ \langle \alpha, x\rangle\ \mathsf{in}\ e) & \Longrightarrow (M, e[\sigma_1, v/\alpha, 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}\ \tau\ \mathsf{of}\ (e_{\mathsf{i}}; e_\lambda; t_1 t_2.e_\times; t_e.e_\exists)) & \Longrightarrow (M, \mathsf{typecase}\ \tau'\ \mathsf{of}\ (e_{\mathsf{i}}; e_\lambda; t_1 t_2.e_\times; t_e.e_\exists)) \\
(M, \mathsf{typecase\ Int\ of}\ (e_{\mathsf{i}}; e_\lambda; t_1 t_2.e_\times; t_e.e_\exists)) & \Longrightarrow (M, e_i) \\
(M, \mathsf{typecase}\ \tau \to 0\ \mathsf{of}\ (e_{\mathsf{i}}; e_\lambda; t_1 t_2.e_\times; t_e.e_\exists)) & \Longrightarrow (M, e_\lambda) \\
(M, \mathsf{typecase}\ \tau_1 \times \tau_2\ \mathsf{of}\ (e_{\mathsf{i}}; e_\lambda; t_1 t_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_\lambda; t_1 t_2.e_\times; t_e.e_\exists)) & \Longrightarrow (M, e_\exists[\lambda t.\tau/t_e])
\end{array}
$$

**Figure 5: Operational semantics of $\lambda_{GC}$.**

a Typerec that has been hard-wired into the language.

## 6.4 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 $\nu$ at runtime and binds $r$ to it. A term of the form $\mathsf{put}[\rho]v$ allocates a value $v$ in the region $\rho$. 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, the get construct takes a memory address $\nu.\ell$ and dereferences it.

Deallocation is handled implicitly through the $\mathsf{only}\ \Delta\ \mathsf{in}\ e$ construct [21]. It asserts statically that the expression $e$ can be evaluated using the set of regions bound to $\Delta'$ and the code region, which is a subset of the region variables currently in scope.

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

The memory is restricted to the set of regions in $\Delta'$ ($\Psi|_{\Delta'}$) and the code region. Similarly, the other environments ($\Phi$ and $\Gamma$) are restricted to be well formed under $\Delta'$ ($\Phi|_{\Delta'}$ and $\Gamma|_{\Delta'}$). At runtime, an implementation would treat the set of regions in $\Delta'$ as live and reclaim other regions. Since the reclamation works on whole regions, the cost is proportional to the number of regions. Since this number is usually small, it entails an insignificant runtime penalty. *The dynamic check takes care of aliasing. This makes our system significantly simpler since we can avoid the heavy type machinery required to detect aliasing statically.*

The runtime type analysis is handled through a typecase construct. Depending on the head of the type being analyzed, the typecase chooses one of the branches for execution. When analyzing a type variable $t$, we refine types containing $t$ in each of the branches [5].

$$
\Theta \vdash t : \Omega
$$

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

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

## 6.5 Formal properties of the language

In this section, we prove that type checking in $\lambda_{GC}$ is decidable

$\boxed{\Delta \vdash \Upsilon \qquad \vdash \Psi}$

$$
\frac{\Delta; \cdot; \cdot \vdash \sigma_i}{\Delta \vdash \{\ell_1 : \sigma_1, \ldots, \ell_n : \sigma_n\}} \qquad \frac{\{\nu_1, \ldots, \nu_n\} \vdash \Upsilon_i}{\vdash \{\nu_1 : \Upsilon_1, \ldots, \nu_n : \Upsilon_n\}}
$$

$$
\Upsilon_{\mathsf{cd}} = \{\ell_1 : \forall[\vec{\tau_1}][\vec{r_1}](\overrightarrow{v_1 : \sigma_1}) \to 0, \ldots, \ell_n : \forall[\vec{\tau_n}][\vec{r_n}](\overrightarrow{v_n : \sigma_n}) \to 0\}
$$

$\boxed{\Psi \vdash R : \Upsilon \qquad \vdash M : \Psi}$

$$
\frac{\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v_i : \sigma_i}{\Psi \vdash \{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\} : \{\ell_1 : \sigma_1, \ldots, \ell_n : \sigma_n\}}
$$

$$
\frac{\vdash \{\nu_1 : \Upsilon_1, \ldots, \nu_n : \Upsilon_n\} \qquad \{\nu_1 : \Upsilon_1, \ldots, \nu_n : \Upsilon_n\} \vdash R_i : \Upsilon_i}{\vdash \{\nu_1 \mapsto R_1, \ldots, \nu_n \mapsto R_n\} : \{\nu_1 : \Upsilon_1, \ldots, \nu_n : \Upsilon_n\}}
$$

**Figure 7: Environment formation rules.**

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

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

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

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

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

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

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

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

$$\frac{\Phi(\alpha) = \Delta' \quad \Delta' \subset \Delta}{\Delta; \Theta; \Phi \vdash \alpha} \qquad \frac{\Delta; \Theta; \Phi, \alpha{:}\Delta' \vdash \sigma \quad \Delta' \subset \Delta}{\Delta; \Theta; \Phi \vdash \exists \alpha{:}\Delta'.\sigma}$$

$$\frac{\{\vec{r}\}; \Theta; \cdot \vdash \sigma_i \quad \Theta \vdash \tau_i : \kappa_i \quad \rho \in \Delta}{\Delta; \Theta; \Phi \vdash \forall[\![\vec{\tau}]\!][\vec{r}](\vec{\sigma}) \xrightarrow{\rho} 0}$$

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

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

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

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

$$\frac{\Psi; \Delta; \Theta; \Phi; \Gamma \vdash v : \forall[\overrightarrow{t{:}\kappa}][\vec{r}](\vec{\sigma}) \to 0 \ \mathsf{at}\ \rho \quad \Theta \vdash \tau_i : \kappa_i}{\Psi; \Delta; \Theta; \Phi; \Gamma \vdash v[\![\vec{\tau}]\!] : \forall[\![\vec{\tau}]\!][\vec{r}](\vec{\sigma}[\vec{\tau}/\vec{t}]) \xrightarrow{\rho} 0}$$

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

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

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

$$\frac{\Delta'; \Theta; \Phi|_{\Delta'} \vdash \sigma_1 \quad \Psi; \Delta; \Theta; \Phi; \Gamma \vdash v : \sigma_2[\sigma_1/\alpha]}{\Psi; \Delta; \Theta; \Phi; \Gamma \vdash \langle \alpha{:}\Delta'{=}\sigma_1, v{:}\sigma_2 \rangle : \exists \alpha{:}\Delta'.\sigma_2}$$

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

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

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

$$\frac{\begin{array}{c}\Psi; \Delta; \Theta; \Phi; \Gamma \vdash v : \forall[\![\vec{\tau}]\!][\vec{r}](\vec{\sigma}) \xrightarrow{\rho} 0 \\ \Psi; \Delta; \Theta; \Phi; \Gamma \vdash v_i : \sigma_i[\vec{\rho}/\vec{r}] \qquad \rho_i \in \Delta\end{array}}{\Psi; \Delta; \Theta; \Phi; \Gamma \vdash v[\vec{\tau}][\vec{\rho}](\vec{v})}$$

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

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

$$\frac{\Psi; \Delta; \Theta; \Phi; \Gamma \vdash v : \exists \alpha'{:}\Delta'.\sigma \quad \Psi; \Delta; \Theta; \Phi, \alpha{:}\Delta'; \Gamma, x{:}\sigma[\alpha/\alpha'] \vdash e}{\Psi; \Delta; \Theta; \Phi; \Gamma \vdash \mathsf{open}\ v\ \mathsf{as}\ \langle \alpha, x \rangle\ \mathsf{in}\ e}$$

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

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

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

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

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

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

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

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

**Figure 6: Static semantics of $\lambda_{GC}$.**

and that the calculus is sound. We omit the proofs due to space constraints. The reader may refer to the companion technical report for details.

### 6.5.1 Type checking is decidable

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

**Proof** Since the tag language is a simply typed lambda calculus, reduction of well formed tags is strongly normalizing and confluent. The termination of $\mathsf{M}_\rho(\tau)$ follows from a straightforward induction on the size of the tag $\tau$. □

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

**Proof** Since the reduction of well formed types is strongly normalizing, confluence of the reduction follows from local confluence. This follows easily from a case analysis of the reduction of the $\mathsf{M}_\rho(\tau)$ tag. □

### 6.5.2 Soundness

**Definition 6.3** *The machine state $(M, e)$ is well formed iff*

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

Contrary to the other environments, $\Psi$ 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 $\Psi'$ for the new state $(M', e')$ based on the $\Psi$ of the initial state $(M, e)$.

**Proposition 6.4 (Type Preservation)** *If* $\vdash (M, e)$ *and* $(M, e) \Longrightarrow (M', e')$ *then* $\vdash (M', e')$.
**Proof** See the appendix B.

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

# 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 suggest to pair up every object with its forwarding pointer, incurring a significant memory cost.[1] 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, $\lambda_{GCforw}$ extends $\lambda_{GC}$ 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

[1] This additional word is not unheard of, since replicating garbage collectors [14, 1] incur a similar overhead, justified by the desire to provide concurrent collection while avoiding the cost of a read-barrier.

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

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

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

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

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

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

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

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

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:

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

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 (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 variable 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.

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

$$
\begin{array}{ll}
\mathsf{C}_{\rho,\rho'}(\mathsf{Int}) & \Longrightarrow \mathsf{int} \\
\mathsf{C}_{\rho,\rho'}(\tau \to 0) & \Longrightarrow \mathsf{M}_\rho(\tau \to 0) \\
\mathsf{C}_{\rho,\rho'}(\exists t.\tau) & \Longrightarrow (\mathsf{left}(\exists t.\mathsf{C}_{\rho,\rho'}(\tau)) + \mathsf{right}(\mathsf{M}_{\rho'}(\exists t.\tau))) \text{ at } \rho
\end{array}
$$

$$
\mathsf{C}_{\rho,\rho'}(\tau_1 \times \tau_2) \Longrightarrow (+\begin{array}{l}\mathsf{left}(\mathsf{C}_{\rho,\rho'}(\tau_1) \times \mathsf{C}_{\rho,\rho'}(\tau_2)) \\ \mathsf{right}(\mathsf{M}_{\rho'}(\tau_1 \times \tau_2))\end{array}) \text{ at } \rho
$$

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:

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

The translation from $\lambda_{CLOS}$ to this $\lambda_{GCforw}$ 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 figure 9. Compared to the original algorithm, the only difference in the $gc$ function itself is the widening of the heap from $\mathsf{M}_{r1}$ 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 needed 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.

## 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.

$$
\begin{array}{l}
\mathsf{fix}\ gc[t:\Omega][r_1](f:\mathsf{M}_{r_1}(\lambda(t)), x:\mathsf{M}_{r_1}(t)). \\
\quad \mathsf{let\ region}\ r_2\ \mathsf{in} \\
\quad \mathsf{let}\ w = \mathsf{widen}[r_2][(\lambda(t) \times t)](\mathsf{put}[r_1](\mathsf{inl}\ (f, x)))\ \mathsf{in} \\
\quad \mathsf{ifleft}\ w = \mathsf{get}\ w\ \mathsf{then} \\
\quad\quad \mathsf{let}\ w = \mathsf{strip}\ w\ \mathsf{in} \\
\quad\quad \mathsf{let}\ y = copy[t][r_1, r_2](\pi_2 w)\ \mathsf{in} \\
\quad\quad \mathsf{only}\ \{r_2\}\ \mathsf{in}\ (\pi_1 w)[][r_2](y) \\
\quad \mathsf{else} \\
\quad\quad \mathsf{halt}\ 0
\end{array}
$$

$$
\begin{array}{l}
\mathsf{fix}\ copy[t:\Omega][r_1, r_2](x:\mathsf{C}_{r_1,r_2}(t)) : \mathsf{M}_{r_2}(t). \\
\quad \mathsf{typecase}\ t\ \mathsf{of} \\
\quad\quad \begin{array}{ll}
\mathsf{int} & \Rightarrow x \\
\lambda & \Rightarrow x \\
t_1 \times t_2 & \Rightarrow \mathsf{let}\ y = \mathsf{get}\ x\ \mathsf{in} \\
& \quad \mathsf{ifleft}\ y = y\ \mathsf{then} \\
& \quad\quad \mathsf{let}\ x_1 = copy[t_1][r_1, r_2](\pi_1(\mathsf{strip}\ y))\ \mathsf{in} \\
& \quad\quad \mathsf{let}\ x_2 = copy[t_2][r_1, r_2](\pi_2(\mathsf{strip}\ y))\ \mathsf{in} \\
& \quad\quad \mathsf{let}\ z = \mathsf{put}[r_2](\mathsf{inl}\ (x_1, x_2))\ \mathsf{in} \\
& \quad\quad \mathsf{set}\ x := \mathsf{inr}\ z\ ;\ z \\
& \quad \mathsf{else} \\
& \quad\quad \mathsf{strip}\ y \\
\exists t_e & \Rightarrow \mathsf{let}\ y = \mathsf{get}\ x\ \mathsf{in} \\
& \quad \mathsf{ifleft}\ y = y\ \mathsf{then} \\
& \quad\quad \mathsf{open}\ (\mathsf{strip}\ y)\ \mathsf{as}\ \langle t, y\rangle\ \mathsf{in} \\
& \quad\quad \mathsf{let}\ y = copy[t_e t][r_1, r_2](y)\ \mathsf{in} \\
& \quad\quad \mathsf{let}\ z = \mathsf{put}[r_2](\mathsf{inl}\ \langle t{=}t, y{:}\mathsf{M}_{r_2}(t_e t)\rangle)\ \mathsf{in} \\
& \quad\quad \mathsf{set}\ x := \mathsf{inr}\ z\ ;\ z \\
& \quad \mathsf{else} \\
& \quad\quad \mathsf{strip}\ y
\end{array}
\end{array}
$$

**Figure 9: GC with forwarding pointers.**

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 [20], 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.

Rather than an ad-hoc widen we could provide a more general cast that can consistently apply iteratively some 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 all the 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 needed 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 out-side 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*

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

**Proposition 7.2 (Type Preservation)** *If* $\vdash (M, e)$ *and* $(M, e) \Longrightarrow (M', e')$ *then* $\vdash (M', e')$.
**Proof** See the appendix C.

**Proposition 7.3 (Progress)** *If* $\vdash (M, e)$ *then either* $e = \mathsf{halt}\ v$ *or there exists a* $(M', e')$ *such that* $(M, e) \Longrightarrow (M', e')$.
**Proof** See the appendix C.

# 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 $\lambda_{GC}$ 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):

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

Apart from those new constructs (whose static semantics is presented in figure 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:

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

By using the set $\{r, \rho_o\}$ we make sure that if $r$ is the old generation, pointers underneath 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.

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

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

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

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

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

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

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

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

**Figure 10: Static semantics of $\lambda_{GCgen}$.**

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.

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 $\lambda_{GCgen}$ 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 $\exists r \in \Delta.(\sigma \ \mathsf{at}\ r)$, the set $\Delta$ 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.

**Proposition 8.1 (Type Preservation)** *If* $\vdash (M, e)$ *and* $(M, e) \Longrightarrow (M', e')$ *then* $\vdash (M', e')$.
**Proof** See the appendix D.

**Proposition 8.2 (Progress)** *If* $\vdash (M, e)$ *then either* $e = \mathsf{halt}\ v$ *or there exists a* $(M', e')$ *such that* $(M, e) \Longrightarrow (M', e')$.
**Proof** See the appendix D.

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

$$
\begin{array}{l}
\mathsf{fix}\ copy[t\!:\!\Omega][r_y,r_o](x\!:\!\mathsf{M}_{r_y,r_o}(t)) : \mathsf{M}_{r_o,r_o}(t).\\
\quad \mathsf{typecase}\ t\ \mathsf{of}\\
\qquad \mathsf{int} \quad \Rightarrow x\\
\qquad \lambda \quad\ \ \Rightarrow x\\
\quad t_1 \times t_2 \Rightarrow \mathsf{open}\ x\ \mathsf{as}\ \langle r,x\rangle\ \mathsf{in}\\
\qquad\qquad\quad \mathsf{ifreg}\ r = r_o\ \mathsf{then}\ \langle r \in \{r_o\} = r_o, x\rangle\ \mathsf{else}\\
\qquad\qquad\qquad \mathsf{let}\ x_1 = copy[t_1][r_y,r_o](\pi_1(\mathsf{get}\ x))\ \mathsf{in}\\
\qquad\qquad\qquad \mathsf{let}\ x_2 = copy[t_2][r_y,r_o](\pi_2(\mathsf{get}\ x))\ \mathsf{in}\\
\qquad\qquad\qquad \langle r \in \{r_o\} = r_o, \mathsf{put}[r](x_1,x_2)\rangle\\
\quad \exists t_e \quad \Rightarrow \mathsf{open}\ x\ \mathsf{as}\ \langle r,x\rangle\ \mathsf{in}\\
\qquad\qquad\quad \mathsf{ifreg}\ r = r_o\ \mathsf{then}\ \langle r \in \{r_o\} = r_o, x\rangle\ \mathsf{else}\\
\qquad\qquad\qquad \mathsf{open}\ (\mathsf{get}\ x)\ \mathsf{as}\ \langle t,y\rangle\ \mathsf{in}\\
\qquad\qquad\qquad \mathsf{let}\ z = copy[t_e t][r_y,r_o](y)\ \mathsf{in}\\
\qquad\qquad\qquad \langle r \in \{r_o\} = r_o, \mathsf{put}[r]\langle t\!=\!t, z\!:\!\mathsf{M}_{r,r_o}(t_e t)\rangle\rangle
\end{array}
$$

**Figure 11: Generational GC.**

# 9. Related work

Wang and Appel [21] 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 [18] 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 [9, 8] and we show how to use intentional type analysis (on quantified types [19]) to typecheck the GC-copy function.

The idea of intensional type analysis was first proposed by Harper and Morrisett [7]. 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. [4] 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 [19] 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., $M_\rho$) used in this paper do not require the full power of what is provided in [19] because our source language is only a simply typed lambda calculus.

Tofte and Talpin [17] proposed to use region calculus to type check memory management for higher-order functional languages. Crary et al [3] presented a low-level typed intermediate language that can express explicit region allocation and deallocation. Our $\lambda_{GC}$ language borrows the basic organization of memories and regions from Crary et al [3]. The main difference is that we don't require explicit capabilities—region deallocation is handled through the `only` primitive.

Necula [13] proposed the idea of a certifying compiler and showed the construction of a certifying compiler for a type-safe subset of C. Morrisett et al. [12] showed that a fully type preserving compiler generating type safe assembly code is a practical basis for a certifying compiler. This paper shows that low-level runtime services such as garbage collection can also be expressed in a type safe language.

# 10. Conclusion 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 [19], 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 [**?**].

- 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 [**?**].

- 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* [5, 4]. This will also allow us to use a simpler closure conversion algorithm for polymorphic code, eliminating the need for translucent types.

# REFERENCES


[1] G. Blelloch and P. Cheng. On bounding time and space for multiprocessor garbage collection. In *Symposium on Programming Languages Design and Implementation*, pages 104–107. ACM Press, May 1999.

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

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

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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. Available at `http://flint.cs.yale.edu`.

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

[12] 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.

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

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

[15] 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.

[16] 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.

[17] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value $\lambda$-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.

[18] 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.

[19] 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.

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

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

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

[23] D. C. Wang and A. W. Appel. Type-preserving garbage collectors. In *Proc. 28th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, page (to appear). ACM Press, 2001.

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

# Appendix

# A. Closed CPS garbage collector

Figure 12 presents the code of the basic collector after CPS and closure conversion (the direct-style code is shown in Fig. 4). The presence of free tag variables in the continuations requires the use of a form of translucent types for the typed closure conversion [9].

In the general case typed closure conversion also requires existential quantification over kinds, but in the present case, we can avoid it by using a superset of all possible kinds: since some continuations require $t_1, t_2$ of kind $\Omega, \Omega$ while others only need $t_1, t_e$ or kind $\Omega, (\Omega \to \Omega)$, we unify the two into $t_1, t_2, t_e$ where some of the arguments are simply left unused.

The verbose type annotations make it look more scary than it really is. Four new functions were introduced because of CPS conversion. They are all used as continuations to calls to *copy*:

1. *gcend* is the code executed at the end of the toplevel call to *copy* and just finishes the garbage collection by freeing the from space and calling back the mutator.
2. *copyexist1* is the continuation of the recursive call to copy an existential package.
3. Since copying a pair requires two recursive calls, we need both *copypair1* for the continuation of the first call and *copypair2* for the continuation of the second.

Since the code has to be closed, all the free variables need to be passed explicitly via the continuation object $k$ whose type is abbreviated as $t_k[t]$ but is really a big existential wrapper around the real data whose type is abbreviated as $t_c[t]$.

— Syntactic type shorthands for notational convenience: —

$t_c[t] \equiv \forall[\![t_1, t_2, t_e]\!][r_1, r_2, r_3](\mathsf{M}_{r_2}(t), \alpha_c) \xrightarrow{\rho} 0 \times \alpha_c$ — Basic type of the continuations of *copy* —

$t_k[t] \equiv (\exists t_1 : \Omega.\exists t_2 : \Omega.\exists t_e : \Omega \longrightarrow \Omega.\exists \alpha_c : \{r_1, r_2, r_3\}.t_c[t])$ at $r_3$ — Same as $t_c$ but closed with existential packages —

— The main *GC* entry point —

fix $gc[t:\Omega][r_1](f : \forall[][r](\mathsf{M}_r(t)) \rightarrow 0, x : \mathsf{M}_{r_1}(t))$.
  let region $r_2$ in
  let region $r_3$ in
  let $c = (gcend[\![t, \mathsf{int}, \lambda t.t]\!], f)$ in
  let $k = \mathsf{put}[r_3]\langle t_1 = t, t_2 = \mathsf{int}, t_e = \lambda t.t, \alpha_c : \{r_1, r_2, r_3\} = \forall[][r](\mathsf{M}_r(t)) \rightarrow 0, c : t_c[t]\rangle$ in
  $copy[t][r_1, r_2, r_3](x, k)$

— The second half of *GC*, passed as a continuation to *copy* —

fix $gcend[t_1 : \Omega, t_2 : \Omega, t_e : \Omega \longrightarrow \Omega][r_1, r_2, r_3](y : \mathsf{M}_{r_2}(t_1), f : \forall[][r](\mathsf{M}_r(t_1)) \rightarrow 0)$.
  only $\{r_2\}$ in $f[][r_2](y)$

— The main *copy* entry point —

fix $copy[t:\Omega][r_1, r_2, r_3](x : \mathsf{M}_{r_1}(t), k : t_k[t])$.
  typecase $t$ of
    int $\Rightarrow$ open (get $k$) as $\langle t_1, t_2, t_e, \alpha_c, c\rangle$ in $(\pi_1 c)[t_1, t_2, t_e][r_1, r_2, r_3](x, \pi_2 c)$
    $\lambda$ $\Rightarrow$ open (get $k$) as $\langle t_1, t_2, t_e, \alpha_c, c\rangle$ in $(\pi_1 c)[t_1, t_2, t_e][r_1, r_2, r_3](x, \pi_2 c)$
    $t_1 \times t_2 \Rightarrow$ let $c = (copypair1[\![t_1, t_2, \lambda t.t]\!], (\pi_2(\mathsf{get}\ x), k))$ in
      let $k = \mathsf{put}[r_3]\langle t_1 = t_1, t_2 = t_2, t_e = \lambda t.t, \alpha_c : \{r_1, r_2, r_3\} = \mathsf{M}_{r_1}(t_2) \times t_k[t], c : t_c[t_1]\rangle$ in
      $copy[t_1][r_1, r_2, r_3](\pi_1(\mathsf{get}\ x), k)$
    $\exists t_e$ $\Rightarrow$ open (get $x$) as $\langle t_x, y\rangle$ in
      let $c = (copyexist1[\![t_x, \mathsf{int}, t_e]\!], k)$ in
      let $k = \mathsf{put}[r_3]\langle t_1 = t_x, t_2 = \mathsf{int}, t_e = t_e, \alpha_c : \{r_1, r_2, r_3\} = t_k[t], c : t_c[t_e t_1]\rangle$ in
      $copy[t_e t_x][r_1, r_2, r_3](y, k)$

— First continuation when copying a pair —

fix $copypair1[t_1 : \Omega, t_2 : \Omega, t_e : \Omega \longrightarrow \Omega][r_1, r_2, r_3](x_1 : \mathsf{M}_{r_2}(t_1), c : \mathsf{M}_{r_1}(t_2) \times t_k[t_1 \times t_2])$.
  let $c' = (copypair2[\![t_2, t_1, \lambda t.t]\!], (x_1, \pi_2 c,))$ in
  let $k = \mathsf{put}[r_3]\langle t_1 = t_1, t_2 = t_2, t_e = \lambda t.t, \alpha_c : \{r_1, r_2, r_3\} = \mathsf{M}_{r_2}(t_1) \times t_k[t_1 \times t_2], c' : t_c[t_2]\rangle$ in
  $copy[t_2][r_1, r_2, r_3](\pi_1 c, k)$

— Second continuation when copying a pair —

fix $copypair2[t_1 : \Omega, t_2 : \Omega, t_e : \Omega \longrightarrow \Omega][r_1, r_2, r_3](x_2 : \mathsf{M}_{r_2}(t_2), c : \mathsf{M}_{r_2}(t_1) \times t_k[t_1 \times t_2])$.
  open (get $(\pi_2 c)$) as $\langle t_1, t_2, t_e, \alpha_c, c'\rangle$ in $(\pi_1 c')[t_1, t_2, t_e][r_1, r_2, r_3](\mathsf{put}[r_2](\pi_1 c, x_2), \pi_2 c')$

— Continuation when copying an existential package —

fix $copyexist1[t_1 : \Omega, t_2 : \Omega, t_e : \Omega \longrightarrow \Omega][r_1, r_2, r_3](z : \mathsf{M}_{r_2}(t_e t_1), c : t_k[t_e t_1])$.
  open (get $c$) as $\langle t_1', t_2', t_e', \alpha_c', c'\rangle$ in $(\pi_1 c')[t_1', t_2', t_e'][r_1, r_2, r_3](\mathsf{put}[r_2]\langle t = t_1, z : \mathsf{M}_{r_2}(t_e t)\rangle, \pi_2 c')$

**Figure 12: The basic GC code after CPS and closure conversion.**

## B. Soundness of $\lambda_{GC}$

*Throughout this section, we assume unique variable names. Our environments are sets with no duplicate occurrences and no ordering. It is easy to show by induction over judgments that extending environments with additional bindings is safe. We will assume this in the rest of the section.*

The code region $\mathsf{cd}$ is always implicitly part of the environment. We treat it as a constant region. Even when the environment is restricted to a particular set, say $\Psi|_{\Delta}$, the code region is included in the restricted set. Therefore $\Psi|_{\nu_1,\ldots\nu_k}$ is equivalent to $\{\mathsf{cd} : \Upsilon_{\mathsf{cd}}, \nu_1 : \Upsilon_{\nu_1}, \ldots \nu_k : \Upsilon_{\nu_k}\}$. And $\Psi|_{\mathsf{cd}}$ is equivalent to $\{\mathsf{cd} : \Upsilon_{\mathsf{cd}}\}$.

**Lemma B.1** *If* $\Delta', r; \Theta; \Phi \vdash \sigma$, *then* $\Delta[\nu/r]; \Theta; \Phi[\nu/r] \vdash \sigma[\nu/r]$ *where* $\Delta', r = \Delta$.

**Proof** The proof is a straightforward induction over the structure of $\sigma$. □

**Lemma B.2** $(\Phi[\nu/r])|_{\Delta,\nu} = (\Phi|_{\Delta,r})[\nu/r]$ *and* $(\Gamma[\nu/r])|_{\Delta,\nu} = (\Gamma|_{\Delta,r})[\nu/r]$

**Proof** The lemma is proved by considering the structure of $\Phi$ and $\Gamma$ respectively, and comparing the results in the two cases. □

**Lemma B.3** *If* $\Psi; \Delta', r; \Theta; \Phi; \Gamma \vdash op : \sigma$, *then* $\Psi; \Delta[\nu/r]; \Theta; \Phi[\nu/r]; \Gamma[\nu/r] \vdash op[\nu/r] : \sigma[\nu/r]$ *where* $\Delta', r = \Delta$.

**Proof** The proof is by induction over the structure of *op*. Most of the cases follow directly by induction. We will show only the case for type packages.

**case** $\langle \alpha : \Delta_1 = \sigma_1, v : \sigma_2 \rangle$: We know that

$$\Psi; \Delta', r; \Theta; \Phi; \Gamma \vdash \langle \alpha : \Delta_1 = \sigma_1, v : \sigma_2 \rangle : \exists \alpha : \Delta_1 . \sigma_2$$

This implies that $\Delta_1; \Theta; \Phi|_{\Delta_1} \vdash \sigma_1$ and

$$\Psi; \Delta', r; \Theta; \Phi; \Gamma \vdash v : \sigma_2[\sigma_1/\alpha]$$

Suppose $r \notin \Delta_1$. Then $r$ does not occur free in $\sigma_1$. Then applying the inductive hypothesis to the derivation for $v$, we get that

$$\Psi; \Delta[\nu/r]; \Theta; \Phi[\nu/r]; \Gamma[\nu/r] \vdash v[\nu/r] : (\sigma_2[\nu/r])[\sigma_1/\alpha]$$

Suppose $\Phi_1 = \Phi|_{\Delta_1}$. Then we have that $\Phi[\nu/r]|_{\Delta_1} = \Phi_1, \Phi_2$ and $Dom(\Phi_1) \cap Dom(\Phi_2) = \emptyset$. Therefore, we have that

$$\Delta_1; \Theta; \Phi|_{\Delta_1}, \Phi_2 \vdash \sigma_1$$

This implies that $\Delta_1; \Theta; \Phi[\nu/r]|_{\Delta_1} \vdash \sigma_1$, which leads to the required result.

Consider now that $r \in \Delta_1$. Suppose $\Delta_1 = \Delta_2, r$. Then

$$\Delta_2, r; \Theta; \Phi|_{\Delta_2,r} \vdash \sigma_1$$

Applying lemmas B.1 and B.2 we get that

$$\Delta_1[\nu/r]; \Theta; (\Phi[\nu/r])|_{\Delta_2,\nu} \vdash \sigma_1[\nu/r]$$

But $\Delta_2, \nu = \Delta_1[\nu/r]$. The second subderivation now becomes

$$\Psi; \Delta', r; \Theta; \Phi; \Gamma \vdash v : \sigma_2[\sigma_1/\alpha]$$

By applying the inductive hypothesis we get that

$$\Psi; \Delta[\nu/r]; \Theta; \Phi[\nu/r]; \Gamma[\nu/r] \vdash v[\nu/r] : (\sigma_2[\nu/r])[\sigma_1[\nu/r]/\alpha]$$

This leads to the required result. □

**Lemma B.4** *If* $\Psi; \Delta', r; \Theta; \Phi; \Gamma \vdash e$, *then* $\Psi; \Delta[\nu/r]; \Theta; \Phi[\nu/r]; \Gamma[\nu/r] \vdash e[\nu/r]$ *where* $\Delta', r = \Delta$

**Proof** The proof is by induction over the derivation of $e$. Most of the cases follow directly from the inductive hypothesis. We will consider only one case here.

**case** $\mathsf{only}\ \Delta_1\ \mathsf{in}\ e$: We get that

$$\Psi; \Delta, r; \Theta; \Phi; \Gamma \vdash \mathsf{only}\ \Delta_1\ \mathsf{in}\ e$$

This implies that

$$\Psi|_{\Delta_1}; \Delta_1, \mathsf{cd}; \Theta; \Phi|_{\Delta_1}; \Gamma|_{\Delta_1} \vdash e$$

and $\Delta_1 \subset \Delta, r$. Suppose $r \notin \Delta_1$. Then $r$ does not occur free in $e$. Also $\Delta_1[\nu/r] = \Delta_1$. Suppose $\Gamma|_{\Delta_1} = \Gamma_1$. Then we have that $\Gamma[\nu/r]|_{\Delta_1} = \Gamma_1, \Gamma_2$ and $Dom(\Gamma_1) \cap Dom(\Gamma_2) = \emptyset$. Since we can extend environments, we get that $\Psi|_{\Delta_1}; \Delta_1, \mathsf{cd}; \Theta; \Phi_1, \Phi_2; \Gamma_1, \Gamma_2 \vdash e$, where $\Phi_1$ and $\Phi_2$ are constructed similar to $\Gamma_1$ and $\Gamma_2$. This implies that

$$\Psi|_{\Delta_1}; \Delta_1, \mathsf{cd}; \Theta; \Phi_1, \Phi_2; \Gamma_1, \Gamma_2 \vdash e$$

Also $\Delta_1 \subset (\Delta, r)[\nu/r]$. This leads to the required result.

Suppose now that $r \in \Delta_1$. Suppose that $\Delta_1 = \Delta_2, r$. Then $\Delta_1[\nu/r] = \Delta_2, \nu$. Then we have that

$$\Psi|_{\Delta_2,r}; \Delta_2, r, \mathsf{cd}; \Theta; \Phi|_{\Delta_2,r}; \Gamma|_{\Delta_2,r} \vdash e$$

Applying the inductive hypothesis we get that

$$\Psi|_{\Delta_2,r}; \Delta_1[\nu/r], \mathsf{cd}; \Theta; \Phi|_{\Delta_2,r}[\nu/r]; \Gamma|_{\Delta_2,r}[\nu/r] \vdash e[\nu/r]$$

Applying lemma B.2 we get that

$$\Psi|_{\Delta_2,r}; \Delta_1[\nu/r], \mathsf{cd}; \Theta; \Phi[\nu/r]|_{\Delta_2,\nu}; \Gamma[\nu/r]|_{\Delta_2,\nu} \vdash e[\nu/r]$$

But we have that $\Psi|_{\Delta_2,r} = \Psi|_{\Delta_2}$. Moreover, $\Psi|_{\Delta_2,\nu} = \Psi|_{\Delta_2}, \Psi'$. Therefore, we get that

$$\Psi|_{\Delta_2,\nu}; \Delta_1[\nu/r], \mathsf{cd}; \Theta; \Phi[\nu/r]|_{\Delta_2,\nu}; \Gamma[\nu/r]|_{\Delta_2,\nu} \vdash e[\nu/r]$$

We also have that $\Delta_1[\nu/r] \subset \Delta[\nu/r]$. This leads to the required result. □

**Lemma B.5** *If* $\Theta, t\!:\!\kappa' \vdash \tau : \kappa$ *and* $\Theta \vdash \tau' : \kappa'$*, then* $\Theta \vdash \tau[\tau'/t] : \kappa$

**Proof** The proof is a straightforward induction over the structure of $\tau$. □

**Lemma B.6** *If* $\Delta; \Theta, t\!:\!\kappa; \Phi \vdash \sigma$ *and* $\Theta \vdash \tau : \kappa$*, then* $\Delta; \Theta; \Phi \vdash \sigma[\tau/t]$

**Proof** The proof is again a straighforward induction over the structure of $\sigma$. □

**Lemma B.7** *If* $\Psi; \Delta; \Theta, t\!:\!\kappa; \Phi; \Gamma \vdash op : \sigma$ *and* $\Theta \vdash \tau : \kappa$ *then* $\Psi; \Delta; \Theta; \Phi; \Gamma[\tau/t] \vdash op[\tau/t] : \sigma[\tau/t]$

**Proof** The proof is a straightforward induction over the structure of op. The only unusual case is when $op = \nu.\ell$. In this case, $\Psi(\nu.\ell) = \sigma$ and $Dom(\Psi); \cdot; \cdot \vdash \sigma$. Therefore, the variable $t$ does not occur free in $\sigma$ at $\nu$. □

**Lemma B.8** *If* $\Psi; \Delta; \Theta, t\!:\!\kappa; \Phi; \Gamma \vdash e$ *and* $\cdot \vdash \tau' : \kappa$ *then* $\Psi; \Delta; \Theta; \Phi; \Gamma[\tau'/t] \vdash e[\tau'/t]$

**Proof** The proof is a straightforward induction over the structure of e. The only interesting case is for a typecase when the substituted variable is being analyzed.

**case** typecase $t$ of $(e_{\mathsf{i}}; e_\lambda; t_1 t_2.e_\times; t_e.e_\exists)$: Suppose we substitute the type $\tau'$ for the variable $t$. Then $\tau'$ can only be one of Int, $\tau'' \rightarrow 0$, $\tau_1' \times \tau_2'$, or $\exists t.\tau''$. For a Int, we need to prove that

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

This implies that we need to prove that

$$\Psi; \Delta; \Theta; \Phi; \Gamma[\mathsf{Int}/t] \vdash e_i[\mathsf{Int}/t]$$

By definition, we know that

$$\Psi; \Delta; \Theta, t\!:\!\Omega; \Phi; \Gamma[\mathsf{Int}/t] \vdash e_i[\mathsf{Int}/t]$$

Since $t$ is being substituted away, this leads to the required result.

For a code type we need to prove that

$$\Psi; \Delta; \Theta; \Phi; \Gamma[\tau' \rightarrow 0/t] \vdash \\ (\mathsf{typecase}\ t\ \mathsf{of}\ (e_{\mathsf{i}}; e_\lambda; t_1 t_2.e_\times; t_e.e_\exists))[\tau' \rightarrow 0/t]$$

This implies that we need to prove that

$$\Psi; \Delta; \Theta; \Phi; \Gamma[\tau' \rightarrow 0/t] \vdash e_\lambda[\tau' \rightarrow 0/t]$$

By definition, we get that $\Psi; \Delta; \Theta, t : \Omega; \Phi; \Gamma \vdash e_\lambda$. Substituting for $t$ and applying the inductive hypothesis leads to the result.

For the pair type we need to prove that

$$\Psi; \Delta; \Theta; \Phi; \Gamma[(\tau_1' \times \tau_2')/t] \vdash \\ (\mathsf{typecase}\ t\ \mathsf{of}\ (e_{\mathsf{i}}; e_\lambda; t_1 t_2.e_\times; t_e.e_\exists))[(\tau_1' \times \tau_2')/t]$$

This implies that we need to prove that

$$\Psi; \Delta; \Theta; \Phi; \Gamma[(\tau_1' \times \tau_2')/t] \vdash e_\times[(\tau_1' \times \tau_2'), \tau_1', \tau_2'/t, t_1, t_2]$$

By definition, we know that

$$\Psi; \Delta; \Theta, t\!:\!\Omega, t_1\!:\!\Omega, t_2\!:\!\Omega; \Phi; \Gamma[t_1 \times t_2/t] \vdash e_\times[t_1 \times t_2/t]$$

Note that the variables $t_1$ and $t_2$ do not occur free separately in $\Gamma$. Substituting $\tau_1'$ for $t_1$, $\tau_2'$ for $t_2$, and $\tau_1' \times \tau_2'$ for $t_1 \times t_2$ leads to the required result.

For the existential type we need to prove that

$$\Psi; \Delta; \Theta; \Phi; \Gamma[\exists t_1.\tau'/t] \vdash \\ (\mathsf{typecase}\ t\ \mathsf{of}\ (e_{\mathsf{i}}; e_\lambda; t_1 t_2.e_\times; t_e.e_\exists))[\exists t_1.\tau'/t]$$

This implies that we need to prove that

$$\Psi; \Delta; \Theta; \Phi; \Gamma[\exists t_1.\tau'/t] \vdash e_\exists[\exists t_1.\tau', \lambda t_1.\tau'/t, t_e]$$

By definition we know that

$$\Psi; \Delta; \Theta, t\!:\!\Omega, t_e\!:\!\Omega \longrightarrow \Omega; \Phi; \Gamma[\exists t_1.t_e t_1/t] \vdash e_\exists[\exists t_1.t_e t_1/t]$$

Substituting $(\lambda t_1.\tau')$ for $t_e$ and applying the inductive hypothesis we get that

$$\Psi; \Delta; \Theta, t\!:\!\Omega; \Phi; \Gamma[\exists t_1.\tau'/t] \vdash e_\exists[\exists t_1.\tau', \lambda t_1.\tau'/t, t_e]$$

Since $t$ is being substituted away, we can remove it from the type environment. This leads to the required result. □

**Lemma B.9** *If* $\Delta; \Theta; \Phi, \alpha\!:\!\Delta' \vdash \sigma$ *and* $\Delta'; \Theta; \Phi \vdash \sigma'$*, then* $\Delta; \Theta; \Phi \vdash \sigma[\sigma'/\alpha]$

**Proof** The proof is a straighforward induction over the structure of $\sigma$. In the case of c'ode types, we use the fact that the argument types $\vec{\sigma}$ are fully closed. □

**Lemma B.10** *If* $\Psi; \Delta; \Theta; \Phi, \alpha\!:\!\Delta'; \Gamma \vdash op : \sigma$ *and* $\Delta'; \Theta; \Phi \vdash \sigma'$ *then* $\Psi; \Delta; \Theta; \Phi; \Gamma[\sigma'/\alpha] \vdash op[\sigma'/\alpha] : \sigma[\sigma'/\alpha]$

**Proof** The proof is again by induction over the typing derivation for op. We will consider only the case for packages.

**case** $\langle \beta\!:\!\Delta' = \sigma_1, v\!:\!\sigma_2 \rangle$: By definition,

$$\Psi; \Delta; \Theta; \Phi, \alpha\!:\!\Delta_1; \Gamma \vdash \langle \beta\!:\!\Delta' = \sigma_1, v\!:\!\sigma_2 \rangle : \exists \beta\!:\!\Delta'.\sigma_2$$

There are two possible cases. If $\Delta_1 \subset \Delta'$, then we get that
$\Delta'; \Theta; \Phi|_{\Delta'}, \alpha\!:\!\Delta_1 \vdash \sigma_1$ and
$\Psi; \Delta; \Theta; \Phi, \alpha\!:\!\Delta_1; \Gamma \vdash v : \sigma_2[\sigma_1/\beta]$
By lemma B.9,
$\Delta'; \Theta; \Phi|_{\Delta'} \vdash \sigma_1[\sigma'/\alpha]$
Applying the inductive hypothesis on the typing rule for $v$ leads to the result.

In the other case, we get that $\Delta'; \Theta; \Phi|_{\Delta'} \vdash \sigma_1$ and
$\Psi; \Delta; \Theta; \Phi, \alpha\!:\!\Delta_1; \Gamma \vdash v : \sigma_2[\sigma_1/\beta]$
This implies that $\alpha$ does not occur free in $\sigma_1$. Therefore, we need to prove that

$$\Psi; \Delta; \Theta; \Phi; \Gamma[\sigma'/\alpha] \vdash \\ \langle \beta\!:\!\Delta' = \sigma_1, v[\sigma'/\alpha]\!:\!\sigma_2[\sigma'/\alpha] \rangle : \exists \beta\!:\!\Delta'.\sigma_2[\sigma'/\alpha]$$

This follows from applying the inductive hypothesis to the judgment for $v$. □

**Lemma B.11** *If* $\Delta;\Theta;\Phi \vdash \sigma[\sigma'/\alpha]$ *and* $\alpha$ *occurs free in* $\sigma$*, then* $\Delta;\Theta;\Phi \vdash \sigma'$

**Proof** The proof is by induction over the structure of $\sigma$. $\square$

**Lemma B.12** *If* $\Psi;\Delta;\Theta;\Phi,\alpha{:}\Delta';\Gamma \vdash e$ *and* $\Delta';\Theta;\Phi \vdash \sigma'$ *then*

$\Psi;\Delta;\Theta;\Phi;\Gamma[\sigma'/\alpha] \vdash e[\sigma'/\alpha]$

**Proof** The proof is again by induction over the derivation. The only non-trivial case is the $\mathsf{only}$ construct.
**case** $\mathsf{only}\ \Delta_1\ \mathsf{in}\ e$: By definition,
$\Psi;\Delta;\Theta;\Phi,\alpha{:}\Delta';\Gamma \vdash \mathsf{only}\ \Delta_1\ \mathsf{in}\ e$
and $\Delta';\Theta;\Phi \vdash \sigma'$. Suppose $\Delta' \subset \Delta_1$. Then we get that

$$\Psi|_{\Delta_1};\Delta_1,\mathsf{cd};\Theta;\Phi|_{\Delta_1},\alpha{:}\Delta';\Gamma|_{\Delta_1} \vdash e$$

Applying the inductive hypothesis we get that

$$\Psi|_{\Delta_1};\Delta_1,\mathsf{cd};\Theta;\Phi|_{\Delta_1};\Gamma|_{\Delta_1}[\sigma'/\alpha] \vdash e[\sigma'/\alpha]$$

But we also have that $\Gamma|_{\Delta_1}[\sigma'/\alpha] = \Gamma[\sigma'/\alpha]|_{\Delta_1}$. From here we can conclude that

$$\Psi;\Delta;\Theta;\Phi;\Gamma[\sigma'/\alpha] \vdash \mathsf{only}\ \Delta_1\ \mathsf{in}\ e[\sigma'/\alpha]$$

If $\Delta_1 \subset \Delta'$, then we get that

$$\Psi|_{\Delta_1};\Delta_1,\mathsf{cd};\Theta;\Phi|_{\Delta_1};\Gamma|_{\Delta_1} \vdash e$$

This implies that $\alpha$ does not occur free in $e$. We also have that $\Delta';\Theta;\Phi \vdash \sigma'$. Using lemma B.11, we can show that $\Gamma[\sigma'/\alpha]|_{\Delta_1} = \Gamma|_{\Delta_1}$. Therefore, we get that

$$\Psi|_{\Delta_1};\Delta_1,\mathsf{cd};\Theta;\Phi|_{\Delta_1};\Gamma[\sigma'/\alpha]|_{\Delta_1} \vdash e$$

This implies that

$$\Psi;\Delta;\Theta;\Phi;\Gamma[\sigma'/\alpha] \vdash \mathsf{only}\ \Delta_1\ \mathsf{in}\ e$$

$\square$

**Lemma B.13** *If* $\Psi;\Delta;\Theta;\Phi;\Gamma,x{:}\sigma' \vdash op : \sigma$ *and* $\Psi;\Delta;\Theta;\Phi;\Gamma \vdash v' : \sigma'$ *then* $\Psi;\Delta;\Theta;\Phi;\Gamma \vdash op[v'/x] : \sigma$

**Proof** The proof is a straightforward induction over the typing derivation for op. $\square$

**Lemma B.14** *If* $\Psi;\Delta;\Theta;\Phi;\Gamma \vdash v : \sigma$ *and* $\Delta_1;\Theta;\Phi|_{\Delta_1} \vdash \sigma$ *and* $\Delta_1 \subset \Delta$*, then* $\Psi|_{\Delta_1};\Delta_1;\Theta;\Phi|_{\Delta_1};\Gamma|_{\Delta_1} \vdash v : \sigma$

**Proof** The proof is by induction over the derivation for $v$. Most of the cases follow directly from the inductive hypothesis. We will consider only one case here.
**case** $\nu.\ell$: We have that
$\Psi;\Delta;\Theta;\Phi;\Gamma \vdash \nu.\ell : \sigma\ \mathsf{at}\ \nu$. This implies that $\Psi(\nu.\ell) = \sigma$ and $Dom(\Psi);\cdot;\cdot \vdash \sigma\ \mathsf{at}\ \nu$. However, by assumption we also know that $\Delta_1;\Theta;\Phi|_{\Delta_1} \vdash \sigma\ \mathsf{at}\ \nu$. This implies that $\nu \in \Delta_1$. This implies that $\Psi|_{\Delta_1}(\nu.\ell) = \sigma$. Moreover, we also get that $\Delta_1;\cdot;\cdot \vdash \sigma\ \mathsf{at}\ \nu$. Therefore, we get that $Dom(\Psi|_{\Delta_1});\cdot;\cdot \vdash \sigma\ \mathsf{at}\ \nu$. From here we get that $\Psi|_{\Delta_1};\Delta_1;\Theta;\Phi|_{\Delta_1};\Gamma|_{\Delta_1} \vdash \nu.\ell : \sigma\ \mathsf{at}\ \nu$. $\square$

**Lemma B.15** *If* $\Psi;\Delta;\Theta;\Phi;\Gamma,x{:}\sigma \vdash e$ *and* $\Psi;\Delta;\Theta;\Phi;\Gamma \vdash v : \sigma$ *then* $\Psi;\Delta;\Theta;\Phi;\Gamma \vdash e[v/x]$

**Proof** The proof is again a straightforward induction over the structure of e. We will only show the proof for a couple of cases, the rest of them follow similarly.
**case** $\mathsf{only}\ \Delta\ \mathsf{in}\ e$: We have that
$\Psi;\Delta;\Theta;\Phi;\Gamma,x{:}\sigma \vdash \mathsf{only}\ \Delta_1\ \mathsf{in}\ e$. This implies that $\Psi|_{\Delta_1};\Delta_1;\Theta;\Phi|_{\Delta_1};(\Gamma,x{:}\sigma)|_{\Delta_1} \vdash e$. If we have that $\Delta_1;\Theta;\Phi|_{\Delta_1} \vdash \sigma$, then we get that
$\Psi|_{\Delta_1};\Delta_1;\Theta;\Phi|_{\Delta_1};\Gamma|_{\Delta_1},x{:}\sigma \vdash e$. By lemma B.14 we get that $\Psi|_{\Delta_1};\Delta_1;\Theta;\Phi|_{\Delta_1};\Gamma|_{\Delta_1} \vdash v : \sigma$. Applying the inductive hypothesis gives us that
$\Psi|_{\Delta_1};\Delta_1;\Theta;\Phi|_{\Delta_1};\Gamma|_{\Delta_1} \vdash e[v/x]$.
In the other case, we get that
$\Psi|_{\Delta_1};\Delta_1;\Theta;\Phi|_{\Delta_1};\Gamma|_{\Delta_1} \vdash e$. This implies that $x$ does not occur free in $e$. The required result follows from here.
**case** $\mathsf{typecase}\ t\ \mathsf{of}\ (e_{\mathsf{i}};e_\lambda;t_1t_2.e_\times;t_e.e_\exists)$: By assumption, we get that $\Theta \vdash t : \Omega$
$\Psi;\Delta;\Theta;\Phi;\Gamma[\mathsf{Int}/t],x{:}\sigma[\mathsf{Int}/t] \vdash e_i[\mathsf{Int}/t]$
$\Psi;\Delta;\Theta;\Phi;\Gamma,x{:}\sigma \vdash e_\lambda$
$\Psi;\Delta;\Theta,t_1{:}\Omega,t_2{:}\Omega;\Phi;\Gamma[t_1 \times t_2/t],x{:}\sigma[t_1 \times t_2/t] \vdash e_\times[t_1 \times t_2/t]$
$\Psi;\Delta;\Theta,t_e{:}\Omega \to \Omega;\Phi;\Gamma[\exists t_1.t_e t_1/t],x{:}\sigma[\exists t_1.t_e t_1/t] \vdash e_\exists[\exists t_1.t_e t_1/t]$
By lemma B.7, we know that if $\Psi;\Delta;\Theta;\Phi;\Gamma \vdash v : \sigma$, then $\Psi;\Delta;\Theta;\Phi;\Gamma[\tau/t] \vdash v : \sigma[\tau/t]$. Now substitute $v[\mathsf{Int}/t]$ in the $e_i$ branch, substitute $v$ in the $e_\lambda$ branch, substitute $v[t_1 \times t_2/t]$ in the $e_\times$ branch, and substitute $v[\exists t_1.t_e t_1/t]$ in the $e_\exists$ branch. The required result follows from the inductive hypothesis on each branch. $\square$

**Proposition B.16 (Type Preservation)** *If* $\vdash (M,e)$ *and* $(M,e) \Longrightarrow (M',e')$ *then* $\vdash (M',e')$.
**Proof** The proof is by induction over the evaluation relation. We will consider only the cases that do not follow directly from the inductive hypothesis and the substitution lemmas.
**case** $\nu.\ell[\vec{\tau}][\vec{\nu}](\vec{v})$: The lemma follows from the fact that tag reduction is strongly normalizing and confluent, and that tag reduction preserves kind.
**case** $\nu.\ell[\vec{\tau}'][\vec{\nu}](\vec{v})$: By definition,

$$\Psi;Dom(\Psi);\cdot;\cdot;\cdot \vdash \nu.\ell[\vec{\tau}'][\vec{\nu}](\vec{v})$$

Since $M(\nu.\ell) = (\lambda[\vec{t{:}\kappa}][\vec{r}](\vec{x}{:}\vec{\sigma}).e)$, we have that

$$\Psi;Dom(\Psi);\cdot;\cdot;\cdot \vdash \nu.\ell : \forall[\vec{t{:}\kappa}][\vec{r}](\vec{\sigma}) \to 0\ \mathsf{at}\ \nu$$

This implies that

$$\Psi|_{\mathsf{cd}};\mathsf{cd},\vec{r};\overrightarrow{t{:}\kappa};\cdot;\overrightarrow{x{:}\sigma} \vdash e$$

By the typing rule, we get that

$$\Psi;Dom(\Psi);\cdot;\cdot;\cdot \vdash v_i : \sigma_i[\vec{\nu},\vec{\tau}'/\vec{r},\vec{t}]$$

and $\cdot \vdash \tau_i' : \kappa_i$. From lemma B.4 we get that

$$\Psi|_{\mathsf{cd}};\mathsf{cd},\vec{\nu};\Theta;\cdot;x{:}\sigma[\vec{\nu}/\vec{r}] \vdash e[\vec{\nu}/\vec{r}]$$

From lemma B.8 we get that

$$\Psi|_{\mathsf{cd}};\mathsf{cd},\vec{\nu};\cdot;\cdot;x{:}\sigma[\vec{\nu},\vec{\tau}'/\vec{r},\vec{t}] \vdash e[\vec{\nu},\vec{\tau}'/\vec{r},\vec{t}]$$

Since $\Psi|_{\mathsf{cd}} \subset \Psi$ and $\mathsf{cd},\vec{\nu} \subset Dom(\Psi)$, we can extend the environment for deriving $e$. Applying lemma B.15 we get that

$$\Psi;Dom(\Psi);\cdot;\cdot;\cdot \vdash e[\vec{\nu},\vec{\tau}',\vec{v}/\vec{r},\vec{t},\vec{x}]$$

which leads to the result.
**case** $(v[\![\vec{\tau}]\!])[\vec{\tau}][\vec{\nu}](\vec{v})$: By definition,

$$\Psi;Dom(\Psi);\cdot;\cdot;\cdot \vdash (v[\![\vec{\tau}]\!])[\vec{\tau}][\vec{\nu}](\vec{v})$$

From the typing rules
$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash (v[\![\vec{\tau}]\!]) : \forall[\![\vec{\tau}]\!][\vec{r}](\vec{\sigma}) \xrightarrow{\nu} 0$ for some $\nu$ and $\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v_i : \sigma_i[\vec{\nu}/\vec{r}]$. Again from the typing rules we get that
$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v : \forall[t\vec{:}\kappa][\vec{r}](\vec{\sigma'}) \to 0$ at $\nu$ where $\sigma'_i[\vec{\tau}/\vec{t}] = \sigma_i$ and $\cdot \vdash \tau_i : \kappa_i$. We need to prove that

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v[\vec{\tau}][\vec{\nu}](\vec{v})$$

This is true if
$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v_i : \sigma'_i[\vec{\nu}, \vec{\tau}/\vec{r}, \vec{t}]$. But we already know that this holds.
**case** let $x =$ put$[\nu]v$ in $e$: By definition,

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \text{let } x = \text{put}[\nu]v \text{ in } e$$

From the typing rules,

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \text{put}[\nu]v : \sigma \text{ at } \nu$$

for some type $\sigma$, and $\nu \in Dom(\Psi)$. This implies that

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v : \sigma$$

Again, from the typing rules,

$$\Psi, \nu.\ell : \sigma; Dom(\Psi); \cdot; \cdot; \cdot \vdash \nu.\ell : \sigma \text{ at } \nu$$

The required result now follows from lemma B.15.
**case** let $x =$ get $\nu.\ell$ in $e$: By definition,

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \text{let } x = \text{get } \nu.\ell \text{ in } e$$

From the typing rules we get that

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \nu.\ell : \sigma \text{ at } \nu$$

for some type $\sigma$ Again from the typing rules, we get that $\Psi(\nu.\ell) = \sigma$. This implies that if $M(\nu.\ell) = v$, then

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v : \sigma$$

The required result follows from lemma B.15.
**case** open $\langle \alpha : \Delta = \sigma_1, v : \sigma_2 \rangle$ as $\langle \alpha, x \rangle$ in $e$: The two open constructs are proved similarly. We will show the proof for only one of them. By definition,

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \text{open } \langle \alpha : \Delta = \sigma_1, v : \sigma_2 \rangle \text{ as } \langle \alpha, x \rangle \text{ in } e$$

This implies that

$$\Psi; Dom(\Psi); \cdot; \alpha : \Delta; x : \sigma_2 \vdash e$$

The required result follows from lemmas B.15 and B.12.
**case** let region $r$ in $e$: By definition,

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \text{let region } r \text{ in } e$$

This implies that

$$\Psi; Dom(\Psi), r; \cdot; \cdot; \cdot \vdash e$$

By lemma B.4,

$$\Psi; Dom(\Psi), \nu; \cdot; \cdot; \cdot \vdash e[\nu/r]$$

Since $\nu$ is a newly introduced region, we can extend $\Psi$ with it. This implies that

$$\Psi, \nu \mapsto \{\}; Dom(\Psi), \nu; \cdot; \cdot; \cdot \vdash e[\nu/r]$$

This is the required result.
**case** only $\Delta$ in $e$: By definition,

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \text{only } \Delta \text{ in } e$$

This implies that

$$\Psi|_\Delta; \text{cd}, \Delta; \cdot; \cdot; \cdot \vdash e$$

But cd, $\Delta = Dom(\Psi|_\Delta)$. This implies that

$$\Psi|_\Delta; Dom(\Psi|_\Delta); \cdot; \cdot; \cdot \vdash e$$

which is the required result.

For all of the typecases, the required result follows directly from the typing rules since the value environment is empty.

**Lemma B.17 (Canonical forms)**

1. *If* $\Psi; \Delta; \cdot; \cdot; \cdot \vdash v :$ int *then* $v = n$.
2. *If* $\Psi; \Delta; \cdot; \cdot; \cdot \vdash v : \sigma$ at $\nu$ *then* $v = \nu.\ell$.
3. *If* $\Psi; \Delta; \cdot; \cdot; \cdot \vdash v : \sigma_1 \times \sigma_2$ *then* $v = (v_1, v_2)$.
4. *If* $\Psi; \Delta; \cdot; \cdot; \cdot \vdash v : \exists t : \kappa.\sigma$ *then* $v = \langle t = \tau, v' : \sigma \rangle$.
5. *If* $\Psi; \Delta; \cdot; \cdot; \cdot \vdash v : \exists \alpha : \Delta'.\sigma$ *then* $v = \langle \alpha : \Delta' = \sigma_1, v' : \sigma_2 \rangle$.
6. *If* $\Psi; \Delta; \cdot; \cdot; \cdot \vdash v : \forall[\![\vec{\tau}]\!][\vec{r}](\vec{\sigma}) \xrightarrow{\rho} 0$ *then* $v = v'[\![\vec{\tau}]\!]$.
7. *If* $\Psi; \Delta; \cdot; \cdot; \cdot \vdash v : \forall[t\vec{:}\kappa][\vec{r}](\vec{\sigma}) \to 0$ *then* $v = \lambda[t\vec{:}\kappa][\vec{r}](x\vec{:}\sigma).e$.

**Proof** The proof follows from the inspection of the typing rules for values. □

**Proposition B.18 (Progress)** *If* $\vdash (M, e)$ *then either* $e =$ halt $v$ *or there exists a* $(M', e')$ *such that* $(M, e) \Longrightarrow (M', e')$.
**Proof** The proof is again by induction over the structure of $e$. By definition, $\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash e$. The proof for the individual cases start from this point.
**case** $v[\vec{\tau}][\vec{\nu}](\vec{v})$: From the typing rules, either

$$v : \forall[t\vec{:}\kappa][\vec{r}](\vec{\sigma}) \to 0 \text{ at } \nu \;\; or \;\; v : \forall[\![\vec{\tau}]\!][\vec{r}](\vec{\sigma}) \xrightarrow{\nu} 0$$

In the first case by lemma B.17, $v = \nu.\ell$. From the typing rules $M(\nu.\ell) = \lambda[t\vec{:}\kappa][\vec{r}](x\vec{:}\sigma).e$ This implies that we have a reduction.

In the second case, by lemma B.17 $v = v'[\![\tau]\!]$. In this case also we have a reduction to $v'[\vec{\tau}][\vec{\nu}](\vec{v})$.
**case** let $x = op$ in $e$: If $op = v$, then we have a reduction. If $op = \pi_i v$, then from the typing rules,
$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v : \sigma_1 \times \sigma_2$. The required result follows from lemma B.17. In the case of put$[\nu]v$, the result follows directly. The constraint $\nu \in \Delta$ ensures that $\nu \in Dom(\Psi)$. In the case for get $v$, by the typing rules we know that $v = \nu.\ell$ for some $\nu.\ell$. Again from the typing rule we know that $\Psi(\nu.\ell) = \sigma$. This implies that $M(\nu.\ell) = v'$ for some value $v'$.

For the other cases of $e$, the proposition follows directly from the operational semantics. □

# C. Soundness of $\lambda_{GCforw}$

As before, We use the usual exchange and weakening properties of environments without proving them. The code region $\mathsf{cd}$ is always implicitly part of the environment. Even when the environment is restricted to a particular set, say $\Psi|_\Delta$, the code region is included in the restricted set.

We will show the proofs only for the extensions. The only non-trivial extension is the $\mathsf{widen}$ construct. The proofs of the propositions for the other constructs follow in a straightforward way by induction over the derivations.

Because of the $\mathsf{widen}$ operator, the notion of a well formed machine state has to be slightly adjusted to allow non-well-formed memory elements as long as they can be shown not to influence the execution:

**Definition C.1** *The machine state $(M, e)$ is well formed iff*

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

Note that to show well-formedness we now need to find not only a witness $\Psi$ but also a proper $\overline{M}$ subset.

The reason is that $\mathsf{widen}$ casts the whole live heap from one type to the other but cannot cast any arbitrary $M$ because it might include existential packages of the form $\langle \alpha : \Delta = \sigma_1, v : \sigma_2 \rangle$ where casting $\alpha$ would require an update of $M$, which goes against the idea that $\mathsf{widen}$ is a nop.

This problem appears when we try to prove type preservation, which is the main obstacle to show soundness of the $\mathsf{widen}$ operator, whose proof uses two lemmas for the following two steps:

1. First, we construct a $\overline{M}$ which only contains elements that we can cast (i.e. of type $\mathsf{M}_\rho(\tau)$) and we show that the state is still well-formed. This is the case because when we reach a $\mathsf{widen}$, all the live data if of such a type.
2. Then we show that when everything has a type of the form $\mathsf{M}_\rho(\tau)$, we can cast every single type consistently to its $\mathsf{C}_{\rho,\rho'}(\tau)$ equivalent and the result is still properly typed. This relies on the subsumption rule on sum types and the fact that $\mathsf{C}$ types only differs from $\mathsf{M}$ types by adding branches to sum types.

**Lemma C.2 (Region Substitution)**

1. *If $\Psi; \Delta, r; \Theta; \Phi; \Gamma \vdash op : \sigma$ then $\Psi; \Delta, \nu; \Theta; \Phi[\nu/r]; \Gamma[\nu/r] \vdash op[\nu/r] : \sigma[\nu/r]$.*
2. *If $\Psi; \Delta, r; \Theta; \Phi; \Gamma \vdash e$ then $\Psi; \Delta, \nu; \Theta; \Phi[\nu/r]; \Gamma[\nu/r] \vdash e[\nu/r]$.*

**Proof** Straightforward proof via induction over the derivation of the typing judgment. Note that $\nu \in \Psi$ is not required here.
**case** $\mathsf{let}\ x = \mathsf{widen}[\rho'][\tau](v)\ \mathsf{in}\ e$: We have that
$\Psi; \Delta, r; \Theta; \Phi; \Gamma \vdash \mathsf{let}\ x = \mathsf{widen}[\rho'][\tau](v)\ \mathsf{in}\ e$. This implies
$\Psi; \Delta, r; \Theta; \Phi; \Gamma \vdash v : \mathsf{M}_\rho(\tau)$ and
$\Psi|_{\mathsf{cd}}; \mathsf{cd}, \rho, \rho'; \Theta; \Phi|_{\rho\rho'}; x : \mathsf{C}_{\rho,\rho'}(\tau) \vdash e$. Applying lemma C.2.1 to the derivation for $v$ we get that
$\Psi; \Delta, \nu; \Theta; \Phi[\nu/r]; \Gamma[\nu/r] \vdash v[\nu/r] : \mathsf{M}_{\rho[\nu/r]}(\tau)$ since
$\mathsf{M}_\rho(\tau)[\nu/r] = \mathsf{M}_{\rho[\nu/r]}(\tau)$.

If either $\rho = r$ or $\rho' = r$ then the inductive hypothesis on the derivation of $e$ and lemma B.2 and the fact that $\mathsf{C}_{\rho,\rho'}(\tau)[\nu/r] = \mathsf{C}_{(\rho,\rho')[\nu/r]}(\tau)$ leads to the required result.

In the other case, $\Phi[\nu/r]|_{\rho\rho'} = \Phi|_{\rho\rho'}, \Phi'$ where $Dom(\Phi|_{\rho\rho'}) \cap Dom(\Phi')$ is empty. Moreover, $r$ does not occur free in $e$. Since we can extend environments with new bindings we get that $\Psi|_{\mathsf{cd}}; \mathsf{cd}, \rho, \rho'; \Theta; \Phi|_{\rho\rho'}, \Phi'; x : \mathsf{C}_{\rho,\rho'}(\tau) \vdash e$. This leads to the required result. □

**Lemma C.3 (Type Substitution)**

1. *If $\Psi; \Delta; \Theta, t : \kappa; \Phi; \Gamma \vdash op : \sigma$ and $\Theta \vdash \tau : \kappa$ then $\Psi; \Delta; \Theta; \Phi; \Gamma[\tau/t] \vdash op[\tau/t] : \sigma[\tau/t]$.*
2. *If $\Psi; \Delta; \Theta, t : \kappa; \Phi; \Gamma \vdash e$ and $\Theta \vdash \tau : \kappa$ then $\Psi; \Delta; \Theta; \Phi; \Gamma[\tau/t] \vdash e[\tau/t]$.*
3. *If $\Psi; \Delta; \Theta; \Phi, \alpha : \Delta'; \Gamma \vdash op : \sigma$ and $\Delta'; \Theta; \Phi \vdash \sigma'$ then $\Psi; \Delta; \Theta; \Phi; \Gamma[\sigma'/\alpha] \vdash op[\sigma'/\alpha] : \sigma[\sigma'/\alpha]$*
4. *If $\Psi; \Delta; \Theta; \Phi, \alpha : \Delta'; \Gamma \vdash e$ and $\Delta'; \Theta; \Phi \vdash \sigma'$ then $\Psi; \Delta; \Theta; \Phi; \Gamma[\sigma'/\alpha] \vdash e[\sigma'/\alpha]$*

**Proof** Same thing here, the induction hypothesis can always be applied directly. □

**Lemma C.4 (Value Substitution)**

1. *If $\Psi; \Delta; \Theta; \Phi; \Gamma, x : \sigma \vdash op : \sigma'$ and $\Psi; \Delta; \Theta; \Phi; \Gamma \vdash v : \sigma$ then $\Psi; \Delta; \Theta; \Phi; \Gamma \vdash op[v/x] : \sigma'$.*
2. *If $\Psi; \Delta; \Theta; \Phi; \Gamma, x : \sigma \vdash e$ and $\Psi; \Delta; \Theta; \Phi; \Gamma \vdash v : \sigma$ then $\Psi; \Delta; \Theta; \Phi; \Gamma \vdash e[v/x]$.*

**Proof** The proof is by induction over the derivation of the type judgment. Most of the cases follow directly from the induction hypothesis. Exceptions are:

$\mathsf{widen}$ We need to show that the substitution on $e$ has no effect, which is easy since the variable we are substituting cannot occur freely in $e$.

□

**Definition C.5** *Let $\Psi|_{\nu,\mathsf{M}}$ be the restriction of $\Psi$ to pieces of code or elements of type $\mathsf{M}_\nu(\tau)$:*

$$\Psi|_{\nu',\mathsf{M}} = \{\nu.\ell : \sigma \mid \Psi(\nu.\ell) = \sigma \wedge \begin{pmatrix} \sigma = \forall[\vec{\tau}][\vec{r}](\overrightarrow{v : \sigma}) \to 0 \\ \vee\ \exists \tau.(\sigma\ \mathsf{at}\ \nu = \mathsf{M}_{\nu'}(\tau)) \end{pmatrix} \}$$

Note that since $\mathsf{cd}$ only contains functions, $\Psi|_{\nu,\mathsf{M}}$ will preserve all its contents.

**Lemma C.6** *If $\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v : \mathsf{M}_\nu(\tau)$ then $\Psi|_{\nu,\mathsf{M}}; Dom(\Psi|_\nu); \cdot; \cdot; \cdot \vdash v : \mathsf{M}_\nu(\tau)$*

**Proof** By the definition of the $\mathsf{M}$ type, $v$ is either of type $\mathsf{int}$ or of type $\sigma\ \mathsf{at}\ \nu$. The lemma follows trivially if $v$ is of type $\mathsf{int}$. Consider the other cases.
**case** $\mathsf{M}_\nu(\tau \to 0)$: In this case $v = \mathsf{cd}.\ell$. Therefore
$\Psi(\mathsf{cd}.\ell) = \forall[][r](\mathsf{M}_r(\tau)) \to 0$. Therefore
$M(\mathsf{cd}.\ell) = \lambda[][r](\overline{x : \vec{\sigma}}).e$. The lemma now follows since the body $e$ is typed only under the code region $\Psi|_{\mathsf{cd}}$ and the region environment $\mathsf{cd}, r$.
**case** $\mathsf{M}_\nu(\tau_1 \times \tau_2)$: We again have that $v = \nu.\ell$. Therefore,
$\Psi(\nu.\ell) = \mathsf{left}(\mathsf{M}_\nu(\tau_1) \times \mathsf{M}_\nu(\tau_2))$. This implies that $M(\nu.\ell) = \mathsf{inl}\ v'$. We know that

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \mathsf{inl}\ v' : \mathsf{left}(\mathsf{M}_\nu(\tau_1) \times \mathsf{M}_\nu(\tau_2))$$

This implies that

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v' : (\mathsf{M}_\nu(\tau_1) \times \mathsf{M}_\nu(\tau_2))$$

This implies that $v' = (v'_1, v'_2)$. From the typing rules we get that $\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v'_1 : \mathsf{M}_\nu(\tau_1)$ and $\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v'_2 : \mathsf{M}_\nu(\tau_2)$. Applying the inductive hypothesis to the derivations for $v'_1$ and $v'_2$, we get that $\Psi|_{\nu,\mathsf{M}}; Dom(\Psi|_\nu); \cdot; \cdot; \cdot \vdash v'_1 : \mathsf{M}_\nu(\tau_1)$ and $\Psi|_{\nu,\mathsf{M}}; Dom(\Psi|_\nu); \cdot; \cdot; \cdot \vdash v'_2 : \mathsf{M}_\nu(\tau_2)$. The required result follows from this.

**case** $\mathsf{M}_\nu(\exists t.\tau)$: We again have that $v = \nu.\ell$. Therefore, $\Psi(\nu.\ell) = \mathsf{left}(\exists t.\mathsf{M}_\nu(\tau))$. This implies that $M(\nu.\ell) = \mathsf{inl}\ v'$. We know that

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \mathsf{inl}\ v' : \mathsf{left}(\exists t.\mathsf{M}_\nu(\tau))$$

This implies that

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v' : \exists t.\mathsf{M}_\nu(\tau)$$

This implies that $v' = \langle t = \tau', v'' : \mathsf{M}_\nu(\tau)\rangle$. From here we get that $\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v'' : (\mathsf{M}_\nu(\tau))[\tau'/t]$. But we have that $(\mathsf{M}_\nu(\tau))[\tau'/t] = \mathsf{M}_\nu(\tau[\tau'/t])$. Therefore, applying the inductive hypothesis to the derivation for $v''$, we get that $\Psi|_{\nu,\mathsf{M}}; Dom(\Psi|_\nu); \cdot; \cdot; \cdot \vdash v'' : \mathsf{M}_\nu(\tau[\tau'/t])$. The required result follows from this. □

**Definition C.7** *Let $T_{\rho,\rho'}$ be the type operator that turns a type of the form $\mathsf{M}_\rho(\tau)$ into a type of the form $\mathsf{C}_{\rho,\rho'}(\tau)$ and keeps code pointer types unchanged:*

$$T_{\rho,\rho'}(\mathsf{M}_\rho(\tau)) = \mathsf{C}_{\rho,\rho'}(\tau)$$
$$T_{\rho,\rho'}(\forall[\vec{\tau}][\vec{r}](\overline{v{:}\sigma}) \to 0 \text{ at } \rho) = \forall[\vec{\tau}][\vec{r}](\overline{v{:}\sigma}) \to 0 \text{ at } \rho$$

*We also use this operator on $\Psi$ where it is defined as:*

$$T_{\rho,\rho'}(\Psi) = \{\nu.\ell{:}\sigma \mid \Psi(\nu.\ell) = \sigma' \wedge T_{\rho,\rho'}(\sigma' \text{ at } \rho) = \sigma \text{ at } \rho\}$$

Note that the two parts of the definition of $T_{\rho,\rho'}(\sigma)$ overlap but are consistent since $\mathsf{M}_\rho(\tau)$ and $\mathsf{C}_{\rho,\rho'}(\tau)$ are identical in the case when $\tau$ is an arrow type. Also, since $\mathsf{cd}$ only contains functions, $T_{\rho,\rho'}(\Psi)$ does not change the type of $\mathsf{cd}$. Finally since both $\mathsf{M}$ and $\mathsf{C}$ are iterators, the $\mathsf{T}$ type operator can be defined as an iterator.

**Lemma C.8** *If $\Psi|_{\nu,\mathsf{M}}; Dom(\Psi|_\nu); \cdot; \cdot; \cdot \vdash v : \mathsf{M}_\nu(\tau)$, then $T_{\nu,\nu'}(\Psi|_{\nu,\mathsf{M}}); Dom(\Psi|_\nu), \nu'; \cdot; \cdot; \cdot \vdash v : \mathsf{C}_{\nu\nu'}(\tau)$*

**Proof** When $v$ is of type $\mathsf{int}$, the lemma follows trivially.

**case** $\mathsf{M}_\nu(\tau \to 0)$: In this case $v = \mathsf{cd}.\ell$ and $M(\mathsf{cd}.\ell) = \lambda[][r](\overline{x{:}\sigma}).e$. This implies that $\Psi|_{\mathsf{cd}}; \mathsf{cd}, r; \cdot; \cdot; \overline{x{:}\sigma} \vdash e$. Since $\mathsf{M}_\nu(\tau' \to 0) = \mathsf{C}_{\nu\nu'}(\tau' \to 0)$ the type operator $T$ is the identity for code types. Therefore, the cast will leave the type of the code region unchanged. Therefore $\Psi|_{\mathsf{cd}} = (T_{\nu,\nu'}(\Psi|_{\nu,\mathsf{M}}))|_{\mathsf{cd}}$. This leads to the required result.

**case** $\mathsf{M}_\nu(\tau_1 \times \tau_2)$: We again have that $v = \nu.\ell$. Therefore, $\Psi(\nu.\ell) = \mathsf{left}(\mathsf{M}_\nu(\tau_1) \times \mathsf{M}_\nu(\tau_2))$. This implies that $M(\nu.\ell) = \mathsf{inl}\ v'$. We know that

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \mathsf{inl}\ v' : \mathsf{left}(\mathsf{M}_\nu(\tau_1) \times \mathsf{M}_\nu(\tau_2))$$

This implies that

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v' : (\mathsf{M}_\nu(\tau_1) \times \mathsf{M}_\nu(\tau_2))$$

This implies that $v' = (v'_1, v'_2)$. From the typing rules we get that $\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v'_1 : \mathsf{M}_\nu(\tau_1)$ and $\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v'_2 : \mathsf{M}_\nu(\tau_2)$. Applying lemma C.6 to the derivations for $v'_1$ and $v'_2$, we get that $\Psi|_{\nu,\mathsf{M}}; Dom(\Psi|_\nu); \cdot; \cdot; \cdot \vdash v'_1 : \mathsf{M}_\nu(\tau_1)$ and $\Psi|_{\nu,\mathsf{M}}; Dom(\Psi|_\nu); \cdot; \cdot; \cdot \vdash v'_2 : \mathsf{M}_\nu(\tau_2)$. Applying the inductive hypothesis to the derivations for $v'_1$ and $v'_2$ we get that $T_{\nu,\nu'}(\Psi|_{\nu,\mathsf{M}}); Dom(\Psi|_\nu), \nu'; \cdot; \cdot; \cdot \vdash v'_1 : \mathsf{C}_{\nu\nu'}(\tau_1)$ and $T_{\nu,\nu'}(\Psi|_{\nu,\mathsf{M}}); Dom(\Psi|_\nu), \nu'; \cdot; \cdot; \cdot \vdash v'_2 : \mathsf{C}_{\nu\nu'}(\tau_2)$. From here we can conclude that

$$T_{\nu,\nu'}(\Psi|_{\nu,\mathsf{M}}); Dom(\Psi|_\nu), \nu'; \cdot; \cdot; \cdot \vdash \mathsf{inl}\ v' : \mathsf{left}(\mathsf{C}_{\nu\nu'}(\tau_1) \times \mathsf{C}_{\nu\nu'}(\tau_2))$$

By the subtyping rule we can conclude that

$$T_{\nu,\nu'}(\Psi|_{\nu,\mathsf{M}}); Dom(\Psi|_\nu), \nu'; \cdot; \cdot; \cdot \vdash \mathsf{inl}\ v' : (+\begin{matrix}\mathsf{left}(\mathsf{C}_{\nu\nu'}(\tau_1) \times \mathsf{C}_{\nu\nu'}(\tau_2)) \\ \mathsf{right}(\mathsf{M}_{\nu'}(\tau_2 \times \tau_2))\end{matrix})$$

This is the required result.

**case** $\mathsf{M}_\nu(\exists t.\tau)$: The proof proceeds exactly as in the previous case. We only need to use the fact that $(\mathsf{C}_{\rho,\rho'}(\tau))[\tau'/t] = \mathsf{C}_{\rho,\rho'}(\tau[\tau'/t])$ □

**Proposition C.9 (Type Preservation)**
*If* $\vdash (M, e)$ *and* $(M, e) \Longrightarrow (M', e')$ *then* $\vdash (M', e')$.

**Proof** The proof is by cases on the structure of $e$. For each possible case, we use the typing derivation together with the evaluation step to get the derivation of the new typing judgment. This mostly relies on the substitution lemmas. We only show the more interesting cases.

- $(M, \mathsf{let}\ x = v\ \mathsf{in}\ e) \Longrightarrow (M, e[v/x])$
From $\vdash (M, \mathsf{let}\ x = v\ \mathsf{in}\ e)$ we get that $\vdash \overline{M} : \Psi$ and $\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \mathsf{let}\ x = v\ \mathsf{in}\ e$. The derivation in turns tells us that $\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v : \sigma$ and $\Psi; Dom(\Psi); \cdot; \cdot, x{:}\sigma \vdash e$. At this point, we can apply value substitution to get $\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash e[v/x]$.
- $(M, \mathsf{only}\ \Delta\ \mathsf{in}\ e) \Longrightarrow (M|_\Delta, e)$
Here we get $\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \mathsf{only}\ \Delta\ \mathsf{in}\ e$ which gives us $\Psi|_\Delta; \Delta; \cdot; \cdot; \cdot \vdash e$. Since $\overline{M} \subset M$, we have that $\overline{M}|_\Delta \subset M|_\Delta$. Moreover, we know that $\Delta \subset Dom(\Psi)$. Therefore, $Dom(\Psi|_\Delta) = \Delta$.
- $(M, \mathsf{let}\ x = \mathsf{widen}[\nu'][\tau](v)\ \mathsf{in}\ e) \Longrightarrow (M, e[v/x])$
We know that if $\overline{M} : \Psi$, then we have that $\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \mathsf{let}\ x = \mathsf{widen}[\nu'][\tau](v)\ \mathsf{in}\ e$. This implies that $\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v : \mathsf{M}_\nu(\tau)$ and $\Psi|_{\mathsf{cd}}; \mathsf{cd}, \nu, \nu'; \cdot; \cdot; , x{:}\mathsf{C}_{\nu\nu'}(\tau) \vdash e$ for some $\nu$ and $\nu'$ belonging to $Dom(\Psi)$. By lemma C.6 we have that $\Psi|_{\nu,\mathsf{M}}; Dom(\Psi|_\nu); \cdot; \cdot; \cdot \vdash v : \mathsf{M}_\nu(\tau)$. By lemma C.8 we have that $T_{\nu,\nu'}(\Psi|_{\nu,\mathsf{M}}); Dom(\Psi|_\nu); \cdot; \cdot; \cdot \vdash v : \mathsf{C}_{\nu\nu'}(\tau)$. But we have that $T_{\nu,\nu'}(\Psi|_{\nu,\mathsf{M}}) = \Psi|_{\mathsf{cd}}, \Psi'$ for some $\Psi'$ since the cast leaves the code type unchanged. Also $\Psi' = T_{\nu\nu'}(\Psi|_{\nu,\mathsf{M}})$ Since we can extend the environment, we have that $\Psi|_{\mathsf{cd}}, \Psi'; \mathsf{cd}, \nu, \nu'; \cdot; \cdot; , x{:}\mathsf{C}_{\nu\nu'}(\tau) \vdash e$. Applying lemma C.4.2 we get that $\Psi|_{\mathsf{cd}}, \Psi'; \mathsf{cd}, \nu, \nu'; \cdot; \cdot; \cdot \vdash e[v/x]$. This implies that $\Psi|_{\mathsf{cd}}, \Psi', \nu' \mapsto \{\}; \mathsf{cd}, \nu, \nu'; \cdot; \cdot; \cdot \vdash e[v/x]$. If $\overline{M} : \Psi|_{\mathsf{cd}}, \Psi', \nu' \mapsto \{\}$, then clearly $\overline{M} \subset M$. This leads to the required result.

□

**Lemma C.10 (Canonical forms)**

1. *If* $\Psi; \Delta; \cdot; \cdot; \cdot \vdash v : \mathsf{int}$ *then* $v = n$.

2. *If* $\Psi; \Delta; \cdot; \cdot; \cdot \vdash v : \sigma$ at $\rho$ *then* $v = \nu.\ell$.
3. *If* $\Psi; \Delta; \cdot; \cdot; \cdot \vdash v : \sigma_1 \times \sigma_2$ *then* $v = (v_1, v_2)$.
4. *If* $\Psi; \Delta; \cdot; \cdot; \cdot \vdash v : \exists t.\sigma$ *then* $v = \langle t = \tau, v' : \sigma \rangle$.
5. *If* $\Psi; \Delta'; \cdot; \cdot; \cdot \vdash v : \exists \alpha : \Delta.\sigma$ *then* $v = \langle \alpha : \Delta = \sigma_1, v' : \sigma_2 \rangle$.
6. *If* $\Psi; \Delta; \cdot; \cdot; \cdot \vdash v : \forall [\![\vec{\tau}]\!][\vec{r}](\vec{\sigma}) \xrightarrow{\rho} 0$ *then* $v = v'[\![\vec{\tau}]\!]$.
7. *If* $\Psi; \Delta; \cdot; \cdot; \cdot \vdash v : \forall [t \vec{:} \kappa][\vec{r}](\vec{\sigma}) \rightarrow 0$ *then* $v = \lambda [t \vec{:} \kappa][\vec{r}](x \vec{:} \sigma).e$.

**Proof** The proof follows from the inspection of the typing rules for values. □

**Proposition C.11 (Progress)** *If* $\vdash (M, e)$ *then either* $e =$ halt $v$ *or there exists a* $(M', e')$ *such that* $(M, e) \Longrightarrow (M', e')$.
**Proof** The proof is again by cases on the structure of $e$. It uses lemma C.10. □

# D. Soundness of $\lambda_{GCgen}$

In this section, we prove the soundness of $\lambda_{GCgen}$. We will prove soundness in the same way – with the subject reduction and progress lemmas. Since $\lambda_{GCgen}$ is just an extension of $\lambda_{GC}$, we will prove the required lemmas only for the new cases. As usual we will use the exchange and widening lemmas for environments throughout the section.

**Lemma D.1** *If* $\Delta; \Theta; \Phi \vdash \sigma$*, then* $\Delta[\nu/r]; \Theta; \Phi[\nu/r] \vdash \sigma[\nu/r]$.

**Proof** The proof is by induction over the derivation of $\sigma$. □

**Lemma D.2** $(\Phi[\nu/r])|_{\Delta,\nu} = (\Phi|_{\Delta,r})[\nu/r]$ *and* $(\Gamma[\nu/r])|_{\Delta,\nu} = (\Gamma|_{\Delta,r})[\nu/r]$

**Proof** Same as lemma B.2. □

**Lemma D.3** $(\mathsf{M}_{\rho_1,\rho_2}(\tau))[\tau'/t] = \mathsf{M}_{\rho_1,\rho_2}(\tau[\tau'/t])$ *and* $(\mathsf{M}_{\rho_1,\rho_2}(\tau))[\nu/r] = \mathsf{M}_{\rho_1[\nu/r],\rho_2[\nu/r]}(\tau)$

**Proof** Proved by considering the different possible reductions of $\mathsf{M}_{\rho_1,\rho_2}(\tau)$. □

**Lemma D.4** *If* $\Psi; \Delta, r; \Theta; \Phi; \Gamma \vdash op : \sigma$*, then* $\Psi; \Delta, \nu; \Theta; \Phi[\nu/r]; \Gamma[\nu/r] \vdash op[\nu/r] : \sigma[\nu/r]$

**Proof** The proof is a straightforward induction over the derivation for *op*. Note that $\Delta, \nu = (\Delta, r)[\nu/r]$. It uses lemma D.3 for the case involving subtyping with the $\mathsf{M}_{\rho_1,\rho_2}(\tau)$ type.
**case** $\langle r' \in \Delta' = \rho, v\rangle$: We have that

$$\Psi; \Delta, r; \Theta; \Phi; \Gamma \vdash \langle r' \in \Delta' = \rho, v\rangle : \exists r' \in \Delta'.(\sigma \text{ at } r')$$

This implies that
$\Psi; \Delta, r; \Theta; \Phi; \Gamma \vdash v : \sigma[\rho/r'] \text{ at } \rho$ and $\rho \in \Delta'$ and $\Delta' \subset \Delta, r$. Applying the inductive hypothesis we get that

$$\Psi; \Delta, \nu; \Theta; \Phi[\nu/r]; \Gamma[\nu/r] \vdash v[\nu/r] : (\sigma[\rho/r'])[\nu/r] \text{ at } \rho[\nu/r]$$

This implies that

$$\Psi; \Delta, \nu; \Theta; \Phi[\nu/r]; \Gamma[\nu/r] \vdash v[\nu/r] : (\sigma[\nu/r])[(\rho[\nu/r])/r'] \text{ at } \rho[\nu/r]$$

Moreover, $\Delta'[\nu/r] \subset \Delta, \nu$ and $\rho[\nu/r] \in \Delta'[\nu/r]$. This implies that

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

□

**Lemma D.5** *If* $\Psi; \Delta, r; \Theta; \Phi; \Gamma \vdash e$*, then* $\Psi; \Delta, \nu; \Theta; \Phi[\nu/r]; \Gamma[\nu/r] \vdash e[\nu/r]$

**Proof** The proof is by induction over the derivation of $e$. Note that $\Delta, \nu = (\Delta, r)[\nu/r]$. We will consider only the extra cases here.
**case** $\mathsf{open}\ v \ \mathsf{as}\ \langle r', x\rangle \ \mathsf{in}\ e$: We have that

$$\Psi; \Delta, r; \Theta; \Phi; \Gamma \vdash \mathsf{open}\ v \ \mathsf{as}\ \langle r', x\rangle \ \mathsf{in}\ e$$

This implies that

$$\Psi; \Delta, r, r'; \Theta; \Phi; \Gamma, x : \sigma \text{ at } r' \vdash e$$

and $\Psi; \Delta, r; \Theta; \Phi; \Gamma \vdash v : \exists r' \in \Delta'.(\sigma \text{ at } r')$. Applying lemma D.4 to the derivation for $v$ we get that

$$\Psi; \Delta, \nu; \Theta; \Phi[\nu/r]; \Gamma[\nu/r] \vdash v[\nu/r] : \exists r' \in \Delta'[\nu/r].(\sigma[\nu/r] \text{ at } r')$$

Applying the inductive hypothesis to the derivation for $e$ we get that

$$\Psi; \Delta, \nu, r'; \Theta; \Phi[\nu/r]; \Gamma[\nu/r], x : \sigma[\nu/r] \text{ at } r' \vdash e[\nu/r]$$

From this we get that

$$\Psi; \Delta, \nu; \Theta; \Phi[\nu/r]; \Gamma[\nu/r] \vdash \mathsf{open}\ v[\nu/r] \ \mathsf{as}\ \langle r', x\rangle \ \mathsf{in}\ e[\nu/r]$$

**case** $\mathsf{ifreg}\ (r_1 = r_2)\ e_1\ e_2$: We have that

$$\Psi; (\Delta, r)[r', r'/r_1, r_2]; \Theta; \Phi[r', r'/r_1, r_2]; \Gamma[r', r'/r_1, r_2] \vdash e_1[r', r'/r_1, r_2]$$

and $\Psi; \Delta, r; \Theta; \Phi; \Gamma \vdash e_2$ and $r' \notin \Delta, r$. Applying the inductive hypothesis to the derivation for $e_2$ leads to

$$\Psi; \Delta, \nu; \Theta; \Phi[\nu/r]; \Gamma[\nu/r] \vdash e_2[\nu/r]$$

Applying the inductive hypothesis to the derivation for $e_1$ and using the fact that $r$ is different from $r_1$ and $r_2$ we get that

$$\Psi; (\Delta, \nu)[r', r'/r_1, r_2]; \Theta; (\Phi[\nu/r])[r', r'/r_1, r_2]; (\Gamma[\nu/r])[r', r'/r_1, r_2] \vdash (e_1[\nu/r])[r', r'/r_1, r_2]$$

This leads to the required result.
**case** $\mathsf{ifreg}\ (r = r_2)\ e_1\ e_2$: We have that

$$\Psi; \Delta, r; \Theta; \Phi; \Gamma \vdash \mathsf{ifreg}\ (r = r_2)\ e_1\ e_2$$

This implies that

$$\Psi; (\Delta, r)[r', r'/r, r_2]; \Theta; \Phi[r', r'/r, r_2]; \Gamma[r', r'/r, r_2] \vdash e_1[r', r'/r, r_2]$$

and $\Psi; \Delta, r; \Theta; \Phi; \Gamma \vdash e_2$. We must prove that

$$\Psi; \Delta, \nu; \Theta; \Phi[\nu/r]; \Gamma[\nu/r] \vdash \mathsf{ifreg}\ (\nu = r_2)\ e_1[\nu/r]\ e_2[\nu/r]$$

This implies that we must prove that
$\Psi; \Delta, \nu; \Theta; \Phi[\nu/r]; \Gamma[\nu/r] \vdash e_2[\nu/r]$ and

$$\Psi; (\Delta, \nu)[\nu/r_2]; \Theta; (\Phi[\nu/r])[\nu/r_2]; (\Gamma[\nu/r])[\nu/r_2] \vdash (e_1[\nu/r])[\nu/r_2]$$

This implies that we must prove that

$$\Psi; (\Delta, r)[\nu, \nu/r, r_2]; \Theta; \Phi[\nu, \nu/r, r_2]; \Gamma[\nu, \nu/r, r_2] \vdash e_1[\nu, \nu/r, r_2]$$

Applying the inductive hypothesis to the derivation for $e_2$ leads to the required result for this derivation. In the case for $e_1$, we substitute for $r'$ and applying the inductive hypothesis and the fact that $r' \notin \Delta, r$ leads to the required result.
**case** $\mathsf{ifreg}\ (\nu' = r_2)\ e_1\ e_2$: This case follows directly from the inductive hypothesis on the derivations of $e_1$ and $e_2$ and using the fact that $\nu' \neq \nu$ and $r \neq r_2$.
**case** $\mathsf{ifreg}\ (\nu = r_2)\ e_1\ e_2$: This case follows directly from the inductive hypothesis on the derivations of $e_1$ and $e_2$ and using the fact that $r \neq r_2$.
**case** $\mathsf{ifreg}\ (\nu' = r)\ e_1\ e_2$: We have that

$$\Psi; \Delta, r; \Theta; \Phi; \Gamma \vdash \mathsf{ifreg}\ (\nu' = r)\ e_1\ e_2$$

This means that we know that

$$\Psi; \Delta, r; \Theta; \Phi; \Gamma \vdash e_2$$

We have to prove that

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

This implies that we have to prove that

$$\Psi; \Delta, \nu; \Theta; \Phi[\nu/r]; \Gamma[\nu/r] \vdash e_2[\nu/r]$$

Applying the inductive hypothesis to the derivation for $e_2$ leads to the result.

**case** $\mathsf{ifreg}\ (\nu = r)\ e_1\ e_2$: We have that

$$\Psi; \Delta, r; \Theta; \Phi; \Gamma \vdash \mathsf{ifreg}\ (\nu = r)\ e_1\ e_2$$

This means that we know that

$$\Psi; \Delta, \nu; \Theta; \Phi[\nu/r]; \Gamma[\nu/r] \vdash e_1[\nu/r]$$

We have to prove that

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

This implies that we have to prove that

$$\Psi; \Delta, \nu; \Theta; \Phi[\nu/r]; \Gamma[\nu/r] \vdash e_1[\nu/r]$$

But we already know this. □

**Lemma D.6** *If* $\Theta, t\!:\!\kappa' \vdash \tau : \kappa$ *and* $\Theta \vdash \tau' : \kappa'$, *then* $\Theta \vdash \tau[\tau'/t] : \kappa$

**Proof** The proof is a straightforward induction over the structure of $\tau$. □

**Lemma D.7** *If* $\Delta; \Theta, t\!:\!\kappa; \Phi \vdash \sigma$ *and* $\Theta \vdash \tau : \kappa$, *then* $\Delta; \Theta; \Phi \vdash \sigma[\tau/t]$

**Proof** The proof is again a straighforward induction over the structure of $\sigma$. □

**Lemma D.8** *If* $\Psi; \Delta; \Theta, t\!:\!\kappa; \Phi; \Gamma \vdash op : \sigma$ *and* $\Theta \vdash \tau : \kappa$ *then* $\Psi; \Delta; \Theta; \Phi; \Gamma[\tau/t] \vdash op[\tau/t] : \sigma[\tau/t]$

**Proof** The proof is by induction over the derivation for *op*. □

**Lemma D.9** *If* $\Psi; \Delta; \Theta, t\!:\!\kappa; \Phi; \Gamma \vdash e$ *and* $\cdot \vdash \tau' : \kappa$ *then* $\Psi; \Delta; \Theta; \Phi; \Gamma[\tau'/t] \vdash e[\tau'/t]$

**Proof** The proof is again by induction over the derivation for $e$. □

**Lemma D.10** *If* $\Delta; \Theta; \Phi, \alpha\!:\!\Delta' \vdash \sigma$ *and* $\Delta'; \Theta; \Phi \vdash \sigma'$, *then* $\Delta; \Theta; \Phi \vdash \sigma[\sigma'/\alpha]$

**Proof** The proof is a straighforward induction over the structure of $\sigma$. □

**Lemma D.11** *If* $\Psi; \Delta; \Theta; \Phi, \alpha\!:\!\Delta'; \Gamma \vdash op : \sigma$ *and* $\Delta'; \Theta; \Phi \vdash \sigma'$ *then* $\Psi; \Delta; \Theta; \Phi; \Gamma[\sigma'/\alpha] \vdash op[\sigma'/\alpha] : \sigma[\sigma'/\alpha]$

**Proof** The proof is again by induction over the typing derivation for op. We will consider only the case for region packages.

**case** $\langle r \in \Delta' = \rho, v\rangle$: We have that

$$\Psi; \Delta; \Theta; \Phi, \alpha\!:\!\Delta'; \Gamma \vdash \langle r \in \Delta' = \rho, v\rangle : \exists r \!\in\! \Delta'.(\sigma \ \mathsf{at}\ r)$$

and $\Delta'; \Theta; \Phi \vdash \sigma'$. This implies that $\Psi; \Delta; \Theta; \Phi, \alpha\!:\!\Delta'; \Gamma \vdash v : \sigma[\rho/r]\ \mathsf{at}\ \rho$ and $\rho \in \Delta'$ and $\Delta' \subset \Delta$. Applying the inductive hypothesis we get that

$$\Psi; \Delta; \Theta; \Phi; \Gamma[\sigma'/\alpha] \vdash v[\sigma'/\alpha] : (\sigma[\rho/r])[\sigma'/\alpha]\ \mathsf{at}\ \rho$$

Since $r$ does not occur in $\Delta$, and $\Delta' \subset \Delta$, therefore $r$ does not occur free in $\sigma'$. Therefore we have that

$$\Psi; \Delta; \Theta; \Phi; \Gamma[\sigma'/\alpha] \vdash v[\sigma'/\alpha] : (\sigma[\sigma'/\alpha])[\rho/r]\ \mathsf{at}\ \rho$$

This leads to the lemma. □

**Lemma D.12** *If* $\Psi; \Delta; \Theta; \Phi, \alpha\!:\!\Delta'; \Gamma \vdash e$ *and* $\Delta'; \Theta; \Phi \vdash \sigma'$ *then*

$\Psi; \Delta; \Theta; \Phi; \Gamma[\sigma'/\alpha] \vdash e[\sigma'/\alpha]$

**Proof** The proof is by induction over the derivation of $e$.

**case** $\mathsf{open}\ v\ \mathsf{as}\ \langle r, x\rangle\ \mathsf{in}\ e$: We have that

$$\Psi; \Delta; \Theta; \Phi, \alpha\!:\!\Delta'; \Gamma \vdash v : \exists r \!\in\! \Delta_1.(\sigma\ \mathsf{at}\ r)$$

and $\Psi; \Delta, r; \Theta; \Phi, \alpha\!:\!\Delta'; \Gamma, x\!:\!\sigma\ \mathsf{at}\ r \vdash e$. Applying lemma D.11 to the derivation for $v$ we get that

$$\Psi; \Delta; \Theta; \Phi; \Gamma[\sigma'/\alpha] \vdash v[\sigma'/\alpha] : \exists r \!\in\! \Delta_1.(\sigma[\sigma'/\alpha]\ \mathsf{at}\ r)$$

Applying the inductive hypothesis to the derivation for $e$ we get that

$$\Psi; \Delta, r; \Theta; \Phi; \Gamma[\sigma'/\alpha], x\!:\!\sigma[\sigma'/\alpha]\ \mathsf{at}\ r \vdash e[\sigma'/\alpha]$$

This leads to the required result.

**case** $\mathsf{ifreg}\ (r_1 = r_2)\ e_1\ e_2$: We have that $\Psi; \Delta; \Theta; \Phi, \alpha\!:\!\Delta'; \Gamma \vdash e_2$ and

$$\begin{array}{l}\Psi; \Delta[r, r/r_1, r_1]; \Theta; \\ \Phi[r, r/r_1, r_1], \alpha\!:\!\Delta'[r, r/r_1, r_1]; \Gamma[r, r/r_1, r_2] \vdash \\ \quad e_1[r, r/r_1, r_2]\end{array}$$

Applying the inductive hypothesis to the derivation of $e_2$ leads to

$$\Psi; \Delta; \Theta; \Phi; \Gamma[\sigma'/\alpha] \vdash e_2[\sigma'/\alpha]$$

By lemma D.1 we have that

$$\Delta'[r, r/r_1, r_2]; \Theta; \Phi[r, r/r_1, r_2] \vdash \sigma'[r, r/r_1, r_2]$$

Substitute $\sigma'[r, r/r_1, r_2]$ in the derivation for $e_1$ and by applying the inductive hypothesis we get that

$$\begin{array}{l}\Psi; \Delta[r, r/r_1, r_1]; \Theta; \Phi[r, r/r_1, r_1]; (\Gamma[\sigma'/\alpha])[r, r/r_1, r_2] \vdash \\ \quad (e_1[\sigma'/\alpha])[r, r/r_1, r_2]\end{array}$$

This leads to the required result.

**case** $\mathsf{ifreg}\ (r = \nu)\ e_1\ e_2$: We have that $\Psi; \Delta; \Theta; \Phi, \alpha\!:\!\Delta'; \Gamma \vdash e_2$ and

$$\Psi; \Delta[\nu/r]; \Theta; \Phi[\nu/r], \alpha\!:\!\Delta'[\nu/r]; \Gamma[\nu/r] \vdash e_1[\nu/r]$$

Applying the inductive hypothesis to the derivation of $e_2$ leads to

$$\Psi; \Delta; \Theta; \Phi; \Gamma[\sigma'/\alpha] \vdash e_2[\sigma'/\alpha]$$

By lemma D.1 we have that

$$\Delta'[\nu/r]; \Theta; \Phi[\nu/r] \vdash \sigma'[\nu/r]$$

Substituting $\sigma'[\nu/r]$ in the derivation for $e_1$ and applying the inductive hypothesis gives us that

$$\Psi; \Delta[\nu/r]; \Theta; \Phi[\nu/r]; (\Gamma[\sigma'/\alpha])[\nu/r] \vdash (e_1[\sigma'/\alpha])[\nu/r]$$

which leads to the required result. □

**Lemma D.13** *If* $\Psi; \Delta; \Theta; \Phi; \Gamma, x\!:\!\sigma' \vdash op : \sigma$ *and* $\Psi; \Delta; \Theta; \Phi; \Gamma \vdash v' : \sigma'$ *then* $\Psi; \Delta; \Theta; \Phi; \Gamma \vdash op[v'/x] : \sigma$

**Proof** The proof is by induction over the typing derivation for op. The new cases follow in a straightforward way. □

**Lemma D.14** *If* $\Psi; \Delta; \Theta; \Phi; \Gamma \vdash v : \sigma$ *and* $\Delta_1; \Theta; \Phi|_{\Delta_1} \vdash \sigma$ *and* $\Delta_1 \subset \Delta$, *then* $\Psi|_{\Delta_1}; \Delta_1; \Theta; \Phi|_{\Delta_1}; \Gamma|_{\Delta_1} \vdash v : \sigma$

**Proof** The proof is by induction over the derivation for $v$. We will show only the extra case here.

**case** $\langle r \in \Delta_2 = \rho, v\rangle$: We have that
$\Psi; \Delta; \Theta; \Phi; \Gamma \vdash \langle r \in \Delta_2 = \rho, v\rangle : \exists r \in \Delta_2.(\sigma \mathsf{~at~} r)$. This implies that
$\Psi; \Delta; \Theta; \Phi; \Gamma \vdash v : \sigma[\rho/r] \mathsf{~at~} \rho$ and $\rho \in \Delta_2$ and $\Delta_2 \subset \Delta$. We also have that
$\Delta_1; \Theta; \Phi|_{\Delta_1} \vdash \exists r \in \Delta_2.(\sigma \mathsf{~at~} r)$. This implies that $\Delta_2 \subset \Delta_1$. Moreover,
$\Delta_1, r; \Theta; \Phi|_{\Delta_1} \vdash \sigma$. By lemma D.1 we get that
$\Delta_1, \rho; \Theta; (\Phi|_{\Delta_1})[\rho/r] \vdash \sigma[\rho/r]$. But since $\rho \in \Delta_1$ and $r$ does not occur free in $\Phi|_{\Delta_1}$, we get that
$\Delta_1; \Theta; \Phi|_{\Delta_1} \vdash \sigma[\rho/r]$. Applying the inductive hypothesis to the derivation for $v$ we get that
$\Psi|_{\Delta_1}; \Delta_1; \Theta; \Phi|_{\Delta_1}; \Gamma|_{\Delta_1} \vdash v : \sigma[\rho/r] \mathsf{~at~} \rho$. This implies that
$\Psi|_{\Delta_1}; \Delta_1; \Theta; \Phi|_{\Delta_1}; \Gamma|_{\Delta_1} \vdash \langle r \in \Delta_2 = \rho, v\rangle : \exists r \in \Delta_2.(\sigma \mathsf{~at~} r)$
□

**Lemma D.15** *If* $\Psi; \Delta; \Theta; \Phi; \Gamma, x : \sigma \vdash e$ *and*
$\Psi; \Delta; \Theta; \Phi; \Gamma \vdash v : \sigma$ *then*
$\Psi; \Delta; \Theta; \Phi; \Gamma \vdash e[v/x]$

**Proof** The proof is by induction over the derivation of $e$. We will only consider the case for $\mathsf{ifreg}$.

**case** $\mathsf{ifreg}\ (r = \nu)\ e_1\ e_2$: By definition we have that

$$\Psi; \Delta[\nu/r]; \Theta; \Phi[\nu/r]; \Gamma[\nu/r], x : \sigma[\nu/r] \vdash e_1[\nu/r]$$

and $\Psi; \Delta; \Theta; \Phi; \Gamma, x : \sigma \vdash e_2$. Applying the inductive hypothesis to the derivation of $e_2$, we get that
$\Psi; \Delta; \Theta; \Phi; \Gamma \vdash e_2[v/x]$. By lemma D.4 we know that

$$\Psi; \Delta[\nu/r]; \Theta; \Phi[\nu/r]; \Gamma[\nu/r] \vdash v[\nu/r] : \sigma[\nu/r]$$

Substituting $v[\nu/r]$ for $x$ in the derivation for $e_1$ leads to the lemma.

**case** $\mathsf{ifreg}\ (r_1 = r_2)\ e_1\ e_2$: By definition we have that

$$\begin{array}{l}\Psi; \Delta[r, r/r_1, r_2]; \Theta; \\ \Phi[r, r/r_1, r_2]; \Gamma[r, r/r_1, r_2], x : \sigma[r, r/r_1, r_2] \vdash \\ \quad e_1[r, r/r_1, r_2]\end{array}$$

and $\Psi; \Delta; \Theta; \Phi; \Gamma, x : \sigma \vdash e_2$. By lemma D.4 we know that

$$\begin{array}{l}\Psi; \Delta[r, r/r_1, r_2]; \Theta; \Phi[r, r/r_1, r_2]; \Gamma[r, r/r_1, r_2] \vdash \\ \quad v[r, r/r_1, r_2] : \sigma[r, r/r_1, r_2]\end{array}$$

Substituting $v[r, r/r_1, r_2]$ and $v$ in the derivation of $e_1$ and $e_2$ respectively, and applying the inductive hypothesis leads to the lemma. □

**Proposition D.16 (Type Preservation)** *If* $\vdash (M, e)$ *and* $(M, e) \Longrightarrow (M', e')$ *then* $\vdash (M', e')$.

**Proof** The proof is by induction over the evaluation relation. We will consider only the additional cases here. The lemma follows in a straightforward way for the $\mathsf{ifreg}$ cases.

**case** $\mathsf{open}\ \langle r \in \Delta = \nu, v\rangle \mathsf{~as~} \langle r, x\rangle \mathsf{~in~} e$: We know that

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \mathsf{open}\ \langle r \in \Delta = \nu, v\rangle \mathsf{~as~} \langle r, x\rangle \mathsf{~in~} e$$

This implies that

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \langle r \in \Delta = \nu, v\rangle : \exists r \in \Delta.(\sigma \mathsf{~at~} r)$$

where $\Delta \subset Dom(\Psi)$. By the typing rule for region packages, we have that $\nu \in Dom(\Psi)$ and that
$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v : \sigma[\nu/r] \mathsf{~at~} \nu$. We also have that
$\Psi; Dom(\Psi), r; \cdot; \cdot; x : \sigma \mathsf{~at~} r \vdash e$. By lemma D.5 and since $\nu \in Dom(\Psi)$ we get that

$$\Psi; Dom(\Psi); \cdot; \cdot; x : \sigma[\nu/r] \mathsf{~at~} \nu \vdash e[\nu/r]$$

Applying lemma D.15 leads to the result. □

**Lemma D.17 (Canonical forms)**

1. *If* $\Psi; \Delta; \cdot; \cdot; \cdot \vdash v : \mathsf{int}$ *then* $v = n$.
2. *If* $\Psi; \Delta; \cdot; \cdot; \cdot \vdash v : \sigma \mathsf{~at~} \nu$ *then* $v = \nu.\ell$.
3. *If* $\Psi; \Delta; \cdot; \cdot; \cdot \vdash v : \sigma_1 \times \sigma_2$ *then* $v = (v_1, v_2)$.
4. *If* $\Psi; \Delta; \cdot; \cdot; \cdot \vdash v : \exists t : \kappa.\sigma$ *then* $v = \langle t = \tau, v' : \sigma\rangle$.
5. *If* $\Psi; \Delta; \cdot; \cdot; \cdot \vdash v : \exists \alpha : \Delta'.\sigma$ *then* $v = \langle \alpha : \Delta' = \sigma_1, v' : \sigma_2\rangle$.
6. *If* $\Psi; \Delta; \cdot; \cdot; \cdot \vdash v : \exists r \in \Delta'.(\sigma \mathsf{~at~} r)$ *then* $v = \langle r \in \Delta' = \rho, v'\rangle$.
7. *If* $\Psi; \Delta; \cdot; \cdot; \cdot \vdash v : \forall[\![\vec{\tau}]\!][\vec{r}](\vec{\sigma}) \xrightarrow{\rho} 0$ *then* $v = v'[\![\vec{\tau}]\!]$.
8. *If* $\Psi; \Delta; \cdot; \cdot; \cdot \vdash v : \forall[\vec{t : \kappa}][\vec{r}](\vec{\sigma}) \to 0$ *then* $v = \lambda[\vec{t : \kappa}][\vec{r}](\vec{x : \sigma}).e$.

**Proof** The proof follows from the inspection of the typing rules for values. □

**Proposition D.18 (Progress)** *If* $\vdash (M, e)$ *then either* $e = \mathsf{halt}\ v$ *or there exists a* $(M', e')$ *such that* $(M, e) \Longrightarrow (M', e')$.

**Proof** The proof is by induction over the structure of $e$. The $\mathsf{ifreg}$ cases follow in a straightforward way. We will consider only the region open construct.

**case** $\mathsf{open}\ v \mathsf{~as~} \langle r, x\rangle \mathsf{~in~} e$: We know that

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \mathsf{open}\ v \mathsf{~as~} \langle r, x\rangle \mathsf{~in~} e$$

This means that

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v : \exists r \in \Delta.(\sigma \mathsf{~at~} r)$$

By lemma D.17 we get that $v = \langle r \in \Delta = \rho, v'\rangle$. By the typing rule for region packages, we get that $\rho \in Dom(\Psi)$. This means that $\rho = \nu$ for some $\nu$. Therefore, we have that

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \mathsf{open}\ \langle r \in \Delta = \nu, v'\rangle \mathsf{~as~} \langle r, x\rangle \mathsf{~in~} e$$

But by the operational semantics this goes to $(M, e[\nu, v'/r, x])$. □