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 functions 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, 3].

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 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.

• We show how to use intensional type analysis (ITA) [19, 8] to accurately describe the contract between the mutator and the collector and how the same framework can be applied to construct various different type-safe GCs.

• Using ITA to typecheck GC may seem to be an obvious idea (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.

• 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?
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 functions and is compatible with separate compilation; our collectors are written in provably type-safe languages with rigorous semantics and fully formalized soundness proofs.
The explosive growth of the Internet has induced newfound interest in mobile computation as well as security. Increasingly, applications are being developed at remote sites and then downloaded for execution. A robust mobile code system must allow code from potentially untrusted sources to be executed. At the same time, the system must detect and prevent the execution of malicious code.

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

Proof-carrying code [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 [3] 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 \( \sigma_1 \times \sigma_2 \), 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 code function faithfully copies all the heap, we just need to show that it has a type looking somewhat like \( \forall \alpha . (\alpha \rightarrow (\alpha[T/F])) \) where \( \alpha[T/F] \) stands for the type \( \alpha \) where the region annotation \( T \) is substituted for \( F \) (see Fig. 1). Assuming we have such a function and we don’t keep any reference to the region \( F \), the region calculus will allow us to safely reclaim \( F \).

Clearly, there is no correctness guarantee in sight since the 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:

- 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 ($\text{copy}_{\alpha[T/F]}$, 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 TypeRec such as $\text{St}_{\tau e}(\sigma)$ which substitutes $T$ for $F$ and then use typecase in the body of copy.

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

We could arrange for $\text{St}_{\tau}(\text{St}_{\tau e}(\sigma))$ to reduce to $\text{St}_{\tau e}(\sigma)$. But then all types become $\alpha$ is a quasi-identity.

A better approach is to ensure that the input and output types are symmetric. We first redefine $\text{St}_{\tau e}(\sigma)$ which simply substitutes $\rho$ for any region annotation (why bother with an initial region) and then redefine copy to have type $\forall F.\forall \alpha.\text{St}_{\tau e}(\alpha) \rightarrow \text{St}_{\tau}(\alpha)$ which gets us rid of the special case before the first collection and does not require any special reduction rule for $\alpha$ since $\text{GC}$ 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 $\exists \alpha : \Delta$. The complete program consists of a list of mutually recursive closed function declarations followed by the main term to be executed.

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

The language used to write the garbage-collector (and into which we translate $\lambda_{\text{CLOS}}$ programs) is shown in Fig. 2. It extends $\lambda_{\text{CLOS}}$ with regions [17] and intensional type analysis [8, 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 $v : \ell$ of type $\sigma$ rather than annotations like $\sigma_1 \times \sigma_2$ (which is written $(\sigma_1 \times \sigma_2)_{\alpha}$ instead). Similarly, object allocation and memory accesses are made explicit with put and get. This was only preferred because of its orthogonality.

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

3. Source language $\lambda_{\text{CLOS}}$

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 [7]. Similarly, for the closure conversion using existentials [10, 9].

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

$$
\begin{align*}
\text{(types)} &\quad \tau ::= \text{int} \mid t \times t \mid \tau_1 \rightarrow \tau_2 \mid \exists \ell. \tau \\
\text{(values)} &\quad v ::= n \mid f \mid x \mid (v_1, v_2) \mid (t = t_1, v : \tau_2) \\
\text{(terms)} &\quad e ::= \text{let } x = v \text{ in } e \mid \ell e = \ell_1, v \in e \\
&\quad \text{open } v \text{ as } (t, x) \text{ in } e \mid v(x) \\
\text{(programs)} &\quad p ::= \text{letrec } f = \lambda x. \tau. e \text{ in } e
\end{align*}
$$

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 $(t = t_1, v : \tau_2)$ and of type $(\exists \ell. \tau_2)$. The open $v$ as $(t, x)$ 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.
4.2 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 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 \texttt{typecase} construct, i.e. a more refined tag is substituted for \( \tau \) in each arm of the \texttt{typecase}. Finally, instead of a full-blown \texttt{TypeRec} construct we only provide a hard coded \( \text{M} \), to keep the presentation simpler. \( M_\rho(\tau) \) is the type corresponding to the tag \( \tau \) complemented with region annotations \( \rho \):

\[
M_\rho(\text{Int}) \quad \Rightarrow \quad \text{int} \\
M_\rho(\tau_1 \times \tau_2) \quad \Rightarrow \quad M_\rho(\tau_1) \times M_\rho(\tau_2) \at \rho \\
M_\rho(\exists t.\tau) \quad \Rightarrow \quad \exists t.\Omega M_\rho(\tau) \at \rho \\
M_\rho(\tau \to \sigma) \quad \Rightarrow \quad \forall [r] [M_\rho(\tau) \to \sigma] \at \text{cd}
\]

This definition of \( \text{M} \) 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 exists by having a separate dedicated region \texttt{cd} for code blocks. The indirection provided by memory references allows us to do away with \texttt{letrec}. A value \( \lambda [r][\bigvee \overline{\tau}].e \) is only an array of instructions (which can contain references to other values in \texttt{cd}) and needs to be put into a region to get a function pointer before one can call it. In practice,

\[
F \vdash \lambda_{CLOS} \Rightarrow \lambda_{GC}
\]

functions are placed into the \texttt{cd} region when translating code from \( \lambda_{CLOS} \) and never directly appear in \( \lambda_{GC} \) code.

5. Translating \( \lambda_{CLOS} \) to \( \lambda_{GC} \)

The translation of terms from \( \lambda_{CLOS} \) to \( \lambda_{GC} \) shown in figure 3 is mostly directed by the type translation \( M_\rho \) 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:

\[
\text{fix} \quad \text{swap}(x : \text{int} \times \text{int}) \\
\text{let} \quad x_1 = \pi_1 x \text{ in let } x_2 = \pi_2 x \text{ in let } x' = (x_2, x_1) \text{ in halt 0}
\]

would turn into the following \( \lambda_{GC} \) function (apart from some syn-
fix \(gct: \Omega \rightarrow [r_1] f: \forall \tau \exists \nu.\ell \left[ M, \{ r \} (t) \rightarrow 0, x: M_2 (t) \right] \).

let region \(r_2\) in

let \(y = copy[t][r_1, r_2](x)\) in

only \(r_2\) in \(f\)[\(r_2\)](y)

fix \(copy[t][r_1, r_2](x: M_1 (t)) : M_2 (t)\).

typcase \(t\) of

\(\text{int} \Rightarrow x\)

\(\lambda \Rightarrow x\)

\(t_1 \times t_2 \Rightarrow \text{let } x_1 = copy[t][r_1, r_2](\pi_1 \text{get } x)\) in

\(\text{let } x_2 = copy[t][r_1, r_2](\pi_2 \text{get } x)\) in

\(\text{put}[r_2](x_1, x_2)\)

\(\exists \nu.\ell \Rightarrow \text{open } (\text{get } x)\) as \((t, y)\) in

\(\text{let } z = copy[t, t][r_1, r_2](y)\) in

\(\text{put}[r_2](t = t, z : M_2 (t, t))\)

Figure 4: The garbage collector proper.

tactic conveniences)

\(\text{fix } swap[]:\{r\}(x: \{\text{Int} \times \text{Int}\} \text{ at } r)\).

\(\text{if } gc r \{ \text{get[\{Int \times Int\}[r]}(\text{swap}, x)\})\)

\(\text{let } x = \text{get } x\) in

\(\text{let } x_1 = \pi_1 x\) in

\(\text{let } x_2 = \pi_2 x\) in

\(\text{let } x' = \text{put[\{r\}[x_2, x_1]}\) in

\(\text{halt } 0\)

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 \(\lambda_{\text{CLOS}}\) rather than as they are translated in \(\lambda_{GC}\). This maintains a clear distinction 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 \(\text{ifgc}\) 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 \(let \ x = \pi_1 (\text{get } v)\) in \(e\) instead of \(let \ x' = \text{get } v\) in \(let \ x = \pi_1 x\) in \(e\). Turning such code back into the strict \(\lambda_{CLOS}\) 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 \(\text{copy}\) 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 [10].

The garbage collector itself is very simple: it first allocates the to region, asks \(\text{copy}\) to move everything into it and then free the from region before jumping to its continuation, using the new region.

The \(\text{copy}\) 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; \(\ell\) denotes a region type. Finally, the memory type \(\Psi\) assigns a region type to every region allocated in memory.

6.1 Closure conversion and \textit{copy}.

Since the source language is monomorphic, closure conversion need only rely on existentials. This simplicity is however broken by the \textit{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 \(\forall \alpha. \Delta = \sigma_1, \nu: \sigma_2\). Closure version of the CPS form of \textit{copy} was also the only reason for introducing \(\alpha: \Delta = \sigma_1, \nu: \sigma_2\).

6.2 Functions and code

Since function bodies can contain references to other functions in \(\text{cd}\) but we do not have an easy way for the garbage collector to analyze a function body to trace through those references, \(\text{cd}\) 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 \(\text{cd}\) 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 \(\lambda_{\text{CLOS}}\), 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 \(\lambda_{\text{CLOS}}\) 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 \((M.\tau)\) and tag level applications \((\tau_t)\). 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 \(\forall \ell[\nu][\tau](\sigma) \rightarrow\)
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 \texttt{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 \( M_{\rho}(e) \) can only contain values that are in region \( \rho \). Notice that it is a restricted form of the fully reflexive \texttt{Typerec} operator [19]. Essentially, it is a \texttt{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 \texttt{let region} \( r \) in \( e \) construct which allocates a new region \( \nu \) at runtime and binds \( r \) to it. A term of the form \texttt{put}[\rho][v] allocates a value \( v \) in the region \( \rho \). Data is read from the region in two ways. Functions are read implicitly through a function call. Data may also be read through the \texttt{get} \( v \) construct. Operationally, the \texttt{get} construct takes a memory address \( \nu . v \) and dereferences it.

Deallocation is handled implicitly through the only \( \Delta \) 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.

\[
\Psi; \Delta; \Theta; \Phi; \Gamma | e \quad \Delta' \subseteq \Delta
\]

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 \texttt{typecase} construct. Depending on the head of the type being analyzed, the \texttt{typecase} chooses one of the branches for execution. When analyzing a type variable \( t \), we refine types containing \( t \) in each of the branches [6].

\[
\Theta \vdash t : \Omega \\
\Psi; \Delta; \Theta; \Phi; \Gamma | e_t | t \vdash e_t | \text{int}/t
\]

In the \( e_t \) branch, we know that the type variable \( t \) is bound to \texttt{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 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 nor-
Figure 6: Static semantics of $\lambda_G$. 
Contrary to the other environments, reduction of well formed tags is strongly normalizing and confluent. The termination of $M_\phi(\tau)$ 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 $M_\phi(\tau)$ tag.

### 6.5.2 Soundness

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

$$M \vdash \Psi$$

$$\Psi \vdash \text{Dom}(\Psi); \vdots; e \vdash (M, e)$$

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 $(M, e)$ and $(M, e) \implies (M', e')$ then $(M', e')$.  

**Proof** See the appendix A.

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

**Proof** See the appendix A.

### 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 $(\sigma + \text{fwd } \sigma)$, which requires a single bit per object and corresponds much more closely to traditional implementations. To this end, $\lambda_{GC\text{forw}}$ extends $\lambda_G$ 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\(^1\) references for that purpose.

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

\[
\begin{align*}
\sigma &::= \ldots | \text{left } \sigma | \text{right } \sigma | \text{left } \sigma_1 + \text{right } \sigma_2 | C_{\rho, \rho'}(\tau) \\
v &::= \ldots | \text{inr } v | \text{inr } v \\
op &::= \ldots | \text{strip } v \\
e &::= \ldots | \text{ifleft } x = e_1 e_2 | \text{set } v_1 := v_2 : e \\
&| \text{let } x = \text{widen}[\rho'][\tau](v) \text{ in } e
\end{align*}
\]

\(^1\)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.
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: \( M_\rho(\tau) \) to map source types to the mutator’s view of the data and \( C_{\rho,\rho'}(\tau) \) to map source types to the collector’s view (which adds forwarding pointers). \( M_\rho(\tau) \) is the same as before for base types and for code types, but is changed for existentials and pairs by adding the \( \rho \) constructor that constrains the mutator to provide the tag bit needed to distinguish the forwarded pointer from the non-forwarded data.

\[
\begin{align*}
M_\rho(\text{Int}) & \Rightarrow \text{Int} \\
M_\rho(\tau \rightarrow 0) & \Rightarrow \forall [\tau'] (M_\rho(\tau')) \rightarrow 0 \text{ at } \mathbf{cd} \\
M_\rho(\exists \tau. \tau) & \Rightarrow (\text{left}(\exists \tau. M_\rho(\tau))) \text{ at } \rho \\
M_\rho(\tau_1 \times \tau_2) & \Rightarrow (\text{left}(M_\rho(\tau_1) \times M_\rho(\tau_2))) \text{ at } \rho \\
C_{\rho,\rho'}(\text{Int}) & \Rightarrow \text{Int} \\
C_{\rho,\rho'}(\tau \rightarrow 0) & \Rightarrow M_\rho(\tau \rightarrow 0) \\
C_{\rho,\rho'}(\exists \tau. \tau) & \Rightarrow (\text{left}(\exists \tau. C_{\rho,\rho'}(\tau)) + \text{right}(M_\rho(\exists \tau. \tau))) \text{ at } \rho \\
C_{\rho,\rho'}(\tau_1 \times \tau_2) & \Rightarrow (\text{left}(C_{\rho,\rho'}(\tau_1) \times C_{\rho,\rho'}(\tau_2)) + \text{right}(M_\rho(\tau_1 \times \tau_2))) \text{ at } \rho \\
\end{align*}
\]

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:

\[
\begin{align*}
(M, \text{let } x = \text{strip (inl } v \text{ in } e) & \Rightarrow (M, e[v/x]) \\
(M, \text{let } x = \text{strip (inr } v \text{ in } e) & \Rightarrow (M, e[v/x]) \\
(M, \text{ifleft } x = (\text{inl } e_1 \text{ in } e_2) & \Rightarrow (M, e_1 \text{ in } e_2) \\
(M, \text{ifleft } x = (\text{inr } e_1 \text{ in } e_2) & \Rightarrow (M, e_1 \text{ in } e_2) \\
(M, \text{set } v. l \leftarrow v \text{ in } e & \Rightarrow (M, \langle v. l \leftarrow v \text{ in } e \rangle) \\
(M, \text{let } x = \text{widen} |\rho[\tau](v) \text{ in } e & \Rightarrow (M, e[v/x])
\end{align*}
\]

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 the all the \text{inl} and \text{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 \( M_\rho \) to \( C_{\rho,\rho'} \) 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: when copying a heap object such as a pair, it now has to check with \text{ifleft} whether the object was forwarded, if so it just returns the forwarded object, otherwise it does the copy as before and to overwrite (using \text{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 some form of cast. At first, it seems we are just applying a form of subtyping, but this form of subtyping is very powerful since it allows covariant subtyping of references. This means that aliasing issues have to be handled with extreme care.

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

In order to handle the problem of aliasing mentioned above, it might be possible to rely on some form of linear typing or alias types [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 \( M_\rho(\tau) \). Looking at the definition of \( M \), we can see that all values reachable from \( v \) will have a type of the form \( M_\rho(\tau') \). 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 \textit{widen} is rather intricate. It starts by ignoring all the dead objects from the heap, so that only objects of type $M_\lambda(\tau)$ are left, which get cast to $C_{\rho,\nu}(\tau)$. 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 outside of $\overline{M}$ is needed to complete execution. This safely permits ill-typed garbage.

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

$$\overline{M} \subseteq M \vdash \overline{M}; \Psi; \Psi; \text{Dom}(\Psi); \vdots; e \vdash (M, e)$$

**Proposition 7.2 (Type Preservation)** If $(M, e)$ and $(M, e) \implies (M', e')$ then $(M', e')$.

**Proof** See the appendix C.

**Proposition 7.3 (Progress)** If $(M, e)$ then either $e = \text{halt} v$ or there exists a $(M', e')$ such that $(M, e) \implies (M', e')$.

**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):

$$\sigma ::= \ldots \mid \exists r \in \Delta. (\sigma \text{ at } r)$$

$$v ::= \ldots \mid (r \in \Delta = \rho; v; \sigma)$$

$$e ::= \ldots \mid \begin{array}{l}
\text{open } v \text{ as } (r, x) \text{ in } e \\
\text{ifreg } (\rho_1 = \rho_2) e_1 e_2
\end{array}$$

Apart from those new constructs (whose static semantics is presented in figure 10), the $\lambda$ 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:

$$M_{\nu,r}(\text{int}) \implies \text{int}$$

$$M_{\nu,r}(\tau \to \sigma) \implies \forall v \in \{r \cup r_0\}(M_{\nu,r}(\tau) \to \sigma) \text{ at } \text{cd}$$

$$M_{\nu,r}(\exists \tau r) \implies \exists r \in \{\rho_1, \rho_2\}. (\exists \tau M_{\nu,\rho_1}(\tau) \text{ at } r)$$

$$M_{\nu,r}(\tau_1 \times \tau_2) \implies \exists r \in \{\rho_1, \rho_2\}. (M_{\nu,\rho_1}(\tau_1) \times M_{\nu,\rho_2}(\tau_2) \text{ at } r)$$

By using the set $\{r, \rho_2\}$ we make sure that if $r$ is the old generation, pointers underneath it cannot point back to the new generation.

The operational semantics are as follows:

- $(M, \text{open } v \in \Delta = \nu, v; \sigma)$ as $\langle r, x \rangle$ in $e$ $\implies (M, e[v, v/r, x])$
- $(M, \text{ifreg } (v = v) e_1 e_2) \implies (M, e_1)$
- $(M, \text{ifreg } (v_1 = v_2) e_1 e_2) \implies (M, e_2)$

Figure 10: Static semantics of $\lambda_{GC_{gen}}$.
fix copy t ; \Omega [ r_y , r_o ] ( f : M_o r_y , r_o ( y ) [ t ] ( t ) \rightarrow 0 , x : M_o r_y , r_o ( t ) ) .

let y = copy t \{ r_y , r_o ( x ) \} in

only \{ r_o \} in let region r_y in f \{ r_y , r_o \}( y )

fix copy t ; \Omega [ r_y , r_o ]( x : M_o r_y , r_o ( t ) ) : M_o r_y , r_o ( t ) ) .

typecase t of

   int \Rightarrow x

   \lambda \Rightarrow x

   t_1 \times t_2 \Rightarrow open x as \{ r , x \} in

   ifreg r = r_o then \{ r \in \{ r_o \} = r_o , x \} else

   let x_1 = copy t_1 \{ r_y , r_o ( \{ t_1 ( get x ) \} ) \} in

   let x_2 = copy t_2 \{ r_y , r_o ( \{ t_2 ( get x ) \} ) \} in

   \{ r \in \{ r_o \} = r_o . put r \{ x_1 \} ( x_2 ) \}

\exists e \Rightarrow open x as \{ r , x \} in

ifreg r = r_o then \{ r \in \{ r_o \} = r_o , x \} else

open ( get x ) as \{ t , y \} in

let z = copy t \{ r_y , r_o \}( y ) in

\{ r \in \{ r_o \} = r_o . put r \{ t = t , z : M_o r_y , r_o ( t . t ) \} \}

Figure 11: Generational GC.

sense, our existential is closer to a bounded quantification.

Proposition 8.1 (Type Preservation) If \( \vdash ( M , e ) \) and

\( ( M , e ) \rightarrow ( M' , e' ) \) then \( \vdash ( M' , e' ) \).

Proof See the appendix D.

Proposition 8.2 (Progress) If \( \vdash ( M , e ) \) then either \( e = \text{halt} v \) or

there exists a \( ( M' , e' ) \) such that \( ( M , e ) \rightarrow ( M' , e' ) \).

Proof See the appendix D.

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 [10, 9] and we show how to use intensional 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 [8]. They introduced the idea of having explicit type analysis operators which inductively traverse the structure of types. However, to retain decidability of type checking, they restrict the analysis to a predicative subset of the type language. Crary et al. [5] propose a very powerful type analysis framework. They define a rich kind calculus that includes sum kinds and inductive kinds. They also provide primitive recursion at the type level. Therefore, they can define new kinds within their calculus and directly encode type analysis operators within their language. They also include a novel refinement operation at the term level. Saha et al [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., \( M_o )\) 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 [4] 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 [4]. 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 [2].
- Side-effects and generations. A first approach could be to extend our current generation scheme with a third region containing all the mutable data. But ultimately we will need to use either card-marking or remembered-sets [25].
- Explicit tag storage. Since tags exist at run time, we need to garbage collect them as well. The most promising approach is to reify them into special terms as was done by Crary et al [6, 5]. This will also allow us to use a simpler closure conversion algorithm for polymorphic code, eliminating the need for transluent types.

References


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 $\mathcal{CD}$ 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 $\Psi; \Delta \vdash \sigma$ is equivalent to $\langle \mathcal{CD}; \nu_1; \nu_2; \ldots \nu_k : \mathcal{Y}_k \rangle$. And $\Psi; \mathcal{CD}$ is equivalent to $\langle \mathcal{CD} \rangle$.

**Lemma A.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 $\sigma$. $\square$

**Lemma A.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. $\square$

**Lemma A.3** Suppose $\Delta'; r; \Theta; \Phi; \Gamma \vdash \sigma$ then

$\Psi; \Delta[\nu/r]; \Theta; \Phi[\nu/r]; \Gamma[\nu/r] \vdash \sigma[\nu/r]$ where $\Delta', r = \Delta$.

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

case (\(\alpha; \Delta_1 = \sigma_1; \nu; \sigma_2\)): We know that

$\Psi; \Delta'[\nu/r]; \Theta; \Phi; \Gamma \vdash \alpha; \Delta_1 = \sigma_1; \nu; \sigma_2$.

This implies that $\Delta_1; \Theta; \Phi[\nu/r] \vdash \sigma_1$ and

$\Psi; \Delta'[\nu/r]; \Theta; \Phi; \Gamma \vdash \nu; \sigma_2[\sigma_1/\alpha]$.

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

$\Psi; \Delta[\nu/r]; \Theta; \Phi[\nu/r]; \Gamma[\nu/r] \vdash \sigma_2[\nu/r]/\sigma_1[\alpha]$. Suppose $\Phi_1 = \Phi[\Delta_1]$. Then we have that $\Phi[\nu/r]|_{\Delta_1} = \Phi_1, \Phi_2$ and $\text{Dom}(\Phi_1) \cap \text{Dom}(\Phi_2) = \emptyset$. Therefore, we have that

$\Delta_1; \Theta; \Phi[\Delta_1] \vdash \sigma_1$.

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 A.1 and A.2 we get that

$\Delta_1[\nu/r]; \Theta; \Phi[\nu/r]|_{\Delta_2, \nu} \vdash \sigma_1[\nu/r]$. But $\Delta_2, \nu = \Delta[\nu/r]$. The second subderivation now becomes

$\Psi; \Delta'; r; \Theta; \Phi; \Gamma \vdash \nu; \sigma_2[\sigma_1/\alpha]$.

By applying the inductive hypothesis we get that

$\Psi; \Delta[\nu/r]; \Theta; \Phi[\nu/r]; \Gamma[\nu/r] \vdash e[\nu/r]$. This leads to the required result. $\square$

**Lemma A.4** If $\Psi; \Delta'[\nu/r]; \Theta; \Phi; \Gamma \vdash e$, 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 only $\Delta_1$ in $e$:** We get that

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

This implies that

$$\Psi|\Delta_1; \Delta_1, \cd; \Theta; \Phi|\Delta_1; \Gamma|\Delta_1 \vdash e$$

and $\Delta_1 \subseteq \Delta_1$. 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

$$\Gamma|\nu/r|\Delta_1 = \Gamma_1, \Gamma_2 \text{ and } Dom(\Gamma_1) \cap Dom(\Gamma_2) = \emptyset.$$  

Thus we can extend environments, we get that

$$\Psi|\Delta_1; \Delta_1, \cd; \Theta; \Phi_1, \Phi_2; \Gamma_1, \Gamma_2 \vdash e,$$

where $\Phi_1$ and $\Phi_2$ are constructed similar to $\Gamma_1$ and $\Gamma_2$. This implies that

$$\Psi|\Delta_1; \Delta_1, \cd; \Theta; \Phi_1, \Phi_2; \Gamma_1, \Gamma_2 \vdash e.$$ 

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

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

$$\Delta_1|\nu/r| = \Delta_2, \nu.$$ 

Therefore, we have that

$$\Psi|\Delta_2, r; \cd; \Theta; \Phi|\Delta_2, r; \Gamma|\Delta_2, r \vdash e$$

Applying the inductive hypothesis we get that

$$\Psi|\Delta_2, r; \cd; \Theta; \Phi|\Delta_2, r; \Gamma|\Delta_2, r \vdash e.$$ 

Applying lemma A.2 we get that

$$\Psi|\Delta_2, r; \cd; \Theta; \Phi|\Delta_2, r; \Gamma|\Delta_2, r \vdash e.$$ 

But we have that $\Psi|\Delta_2, r = \Psi|\Delta_2$. Moreover, $\Psi|\Delta_2, r = \Psi|\Delta_2, \nu$.

We also have that $\Delta_1|\nu/r| \subseteq \Delta|\nu/r|$. This leads to the required result.

**Lemma A.5** If $\Theta, t: \kappa' \vdash \tau: \kappa$ and $\Theta \vdash \tau': \kappa'$, then $\Theta \vdash \tau/	au'$. 

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

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

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

**Lemma A.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 $op = \nu.t$. In this case, $\Psi(\nu.t) = \sigma$ and $Dom(\Psi); t; \cdot \vdash \sigma$. Therefore, the variable $t$ does not occur free in $\sigma$ at $\nu$.

**Lemma A.8** If $\Psi: \Delta; \Theta, t: \kappa; \Phi; \Gamma \vdash e$ and $\vdash \tau': \kappa$ 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 $\text{typeof}$. When the substituted variable is being analyzed.

**case typeof $t$ of $(e; e_1; e_2; e_3)$:** Suppose we substitute the type $\tau'$ for the variable $t$. Then $\tau'$ can only be $\text{Int}, \tau'' \rightarrow 0$, $\tau'_r \times \tau''_r$, or $\exists t.\tau''$. For a $\text{Int}$, we need to prove that

$$\Psi; \Delta; \Theta; \Phi; \Gamma|\text{Int}/t| \vdash (\text{typeof } t \text{ of } (e_1; e_2; e_3) |\text{Int}/t|)$$

This implies that we need to prove that

$$\Psi; \Delta; \Theta; \Phi; \Gamma|\text{Int}/t| \vdash e_i|\text{Int}/t|$$

By definition, we know that

$$\Psi; \Delta; \Theta, t: \Omega; \Phi; \Gamma|\text{Int}/t| \vdash e_i|\text{Int}/t|$$

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

For a code type we need to prove that

$$\Psi; \Delta; \Theta; \Phi; \Gamma[\tau'/0/t] \vdash (\text{typeof } t \text{ of } (e_1; e_2; e_3) |\text{Int}/t|)$$

This implies that we need to prove that

$$\Psi; \Delta; \Theta, t: \Omega; \Phi; \Gamma[\tau'/0/t] \vdash e_i|\text{Int}/t|$$

By definition, we get that $\Psi; \Delta, t: \Omega; \Phi; \Gamma \vdash e_i$. Substituting for $t$ and applying the inductive hypothesis leads to the result. For the pair type we need to prove that

$$\Psi; \Delta; \Theta; \Phi; \Gamma[\tau'/0/t] \vdash (\text{typeof } t \text{ of } (e_1; e_2; e_3) |\text{Int}/t|)$$

This implies that we need to prove that

$$\Psi; \Delta, t: \Omega, t_1: \Omega, t_2: \Omega; \Phi; \Gamma[\tau'/0/t] \vdash e_i[\tau'/t]$$

By definition, we know that

$$\Psi; \Delta, t: \Omega, t_1: \Omega, t_2: \Omega; \Phi; \Gamma[\tau'/0/t] \vdash e_i[\tau'/t]$$

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

$$\Psi; \Delta; \Theta; \Phi; \Gamma[\exists t. \tau'/t] \vdash (\text{typeof } t \text{ of } (e_1; e_2; e_3) |\exists t. \tau'/t|)$$

This implies that we need to prove that

$$\Psi; \Delta, \Theta, \Phi; \Gamma[\exists t. \tau'/t] \vdash e_3[\exists t. \tau', \lambda t. \tau'/t, t_e]$$

By definition we know that

$$\Psi; \Delta, \Theta, t: \Omega, t_e: \Omega; \Phi; \Gamma[\exists t. t_e/t] \vdash e_3[\exists t. t_e/t_e]$$

Substituting $(\lambda t. \tau')$ for $t_e$ and applying the inductive hypothesis we get that

$$\Psi; \Delta; \Theta, t: \Omega; \Phi; \Gamma[\exists t. \tau'/t] \vdash e_3[\exists t. \tau', \lambda t. \tau'/t, t_e]$$

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

**Lemma A.9** If $\Delta; \Theta, t: \kappa; \Phi; \Gamma \vdash op: \sigma$ and $\Theta \vdash \tau: \kappa$, then $\Delta; \Theta; \Phi \vdash \sigma[\alpha/\tau]$.

**Proof** The proof is a straightforward induction over the structure of $\sigma$. In the case of $\text{code}$ types, we use the fact that the argument types $\sigma$ are fully closed.

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

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

**case $\langle \beta: \Delta' = \alpha, \nu: \sigma' \rangle$:** By definition,

$$\Psi; \Delta; \Theta, \Phi, \alpha: \Delta; \Gamma \vdash \langle \beta: \Delta' = \alpha, \nu: \sigma' \rangle : \exists \beta: \Delta', \sigma'$$
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 A.9,
\[ \Delta'; \Theta; \Phi|_{\Delta'} \vdash \sigma[\alpha/\alpha'] \]
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
\[ \Psi; \Delta; \Theta; \Phi; \Gamma[\sigma'/\alpha] \vdash (\beta': \Delta' = \sigma_1, v[\sigma'/\alpha]; \sigma_2[\sigma'/\alpha]) : \exists \beta: \Delta'. \sigma_2[\sigma'/\alpha] \]
This follows from applying the inductive hypothesis to the judgement for $v$.

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

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

Lemma A.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 only construct.

case only $\Delta_1 \in e$: By definition,
\[ \Psi; \Delta; \Theta; \Phi; \alpha; \Delta'; \Gamma \vdash \text{only} \Delta_1 \in e \]
and $\Delta'; \Theta; \Phi \vdash \sigma$. Suppose $\Delta' \subset \Delta_1$. Then we get that
\[ \Psi|_{\Delta_1}; \Delta_1, \Theta; \Phi|_{\Delta_1}; \alpha; \Delta'; \Gamma|_{\Delta_1} \vdash e \]
Applying the inductive hypothesis we get that
\[ \Psi|_{\Delta_1}; \Delta_1, \Theta; \Phi|_{\Delta_1}; \Gamma|_{\Delta_1} \vdash [\sigma'/\alpha] \]
But we also have that $\Gamma|_{\Delta_1}[\sigma'/\alpha] = \Gamma|_{\Delta_1}[\sigma'/\alpha]$. From here we can conclude that
\[ \Psi; \Delta; \Theta; \Phi; \Gamma[\sigma'/\alpha] \vdash \text{only} \Delta_1 \in e[\sigma'/\alpha] \]
If $\Delta_1 \subset \Delta'$, then we get that
\[ \Psi|_{\Delta_1}; \Delta_1, \Theta; \Phi|_{\Delta_1}; \Gamma|_{\Delta_1} \vdash e \]
This implies that $\alpha$ does not occur free in $e$. We also have that $\Delta'; \Theta; \Phi \vdash \sigma'$. Using lemma A.11, we can show that $\Gamma[\sigma'/\alpha]|_{\Delta_1} = \Gamma|_{\Delta_1}$. Therefore, we get that
\[ \Psi|_{\Delta_1}; \Delta_1, \Theta; \Phi|_{\Delta_1}; \Gamma[\sigma'/\alpha]|_{\Delta_1} \vdash e \]
This implies that
\[ \Psi; \Delta; \Theta; \Phi; \Gamma[\sigma'/\alpha] \vdash \text{only} \Delta_1 \in e \]

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

Proof The proof is a straightforward induction over the typing derivation for op.

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

Proof The proof is by induction over the derivation for $\tilde{\nu}$. Most of the cases follow directly from the inductive hypothesis. We will consider only one case here.

case $\nu.\ell \cdot \kappa; \ell(v/x)$: We have that
\[ \Psi; \Delta; \Theta; \Phi; \Gamma \vdash \nu.\ell : \sigma \quad \text{at} \quad \nu. \]
This implies that $\Psi(\nu.\ell) = \sigma$ and $\text{Dom}(\Psi) : \nu \vdash \sigma \quad \text{at} \quad \nu$. However, by assumption we also know that $\Delta_1; \Theta; \Phi|_{\Delta_1} \vdash \sigma \quad \text{at} \quad \nu$. This implies that $\nu \in \Delta_1$. This implies that $\Psi(\nu_1(\nu.\ell)) = \sigma$. Moreover, we also get that $\Delta_1 : \nu \vdash \sigma$. Therefore, we get that $\text{Dom}(\Psi|_{\Delta_1}) : \nu \vdash \nu. \ell : \sigma \quad \text{at} \quad \nu$.

Lemma A.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 only $\Delta \in e$: We have that
\[ \Psi; \Delta; \Theta; \Phi; \Gamma, x: \sigma \vdash \text{only} \Delta \in e \]
This implies that
\[ \Psi|_{\Delta}; \Delta, \Theta; \Phi|_{\Delta}; \Gamma \vdash e \]
If we have that $\Delta; \Theta; \Phi|_{\Delta} \vdash \sigma$, then we get that
\[ \Psi|_{\Delta}; \Delta, \Theta; \Phi|_{\Delta}; \Gamma|_{\Delta} \vdash e[\sigma/\Delta] \]
By lemma A.14 we get that $\Psi|_{\Delta}; \Delta, \Theta; \Phi|_{\Delta}; \Gamma|_{\Delta} \vdash \sigma$. Applying the inductive hypothesis gives us that
\[ \Psi|_{\Delta}; \Delta, \Theta; \Phi|_{\Delta}; \Gamma|_{\Delta} \vdash e[v/x] \]
In the other case, we get that
\[ \Psi|_{\Delta}; \Delta, \Theta; \Phi|_{\Delta}; \Gamma|_{\Delta} \vdash e \]
This implies that $x$ does not occur free in $e$. The required result follows from here.

case typecase $t$ of $(e_1; e; t_1; t_2; e_3; t; e_2)$: By assumption, we get that $\nu \vdash t : \Omega$
\[ \Psi; \Delta; \Theta; \Phi; \Gamma, x: \sigma \vdash \text{only} \Delta \in e \]
If we have that $\Delta; \Theta; \Phi|_{\Delta} \vdash \sigma$, then we get that
\[ \Psi|_{\Delta}; \Delta, \Theta; \Phi|_{\Delta}; \Gamma \vdash e[\sigma/\Delta] \]
This implies that $\nu \vdash t : \Omega$. We have that
\[ \Psi; \Delta; \Theta; \Phi; \Gamma \vdash \nu. \ell \cdot \kappa; \ell(v/x) \]
Since $M(\nu.\ell) = (\lambda[t; \kappa]; [\ell](x; \sigma)), e$, we have that
\[ \Psi; \Delta; \Theta; \Phi; \Gamma \vdash \nu. \ell \cdot \kappa; \ell(v/x) \]
This implies that
\[ \Psi|_{\Delta}; \Delta, \Theta; \Phi|_{\Delta}; \Gamma|_{\Delta} \vdash e \]
By the typing rule, we get that
Again, from the typing rules, we get that
\[ \forall \Psi; \text{Dom}(\Psi); \cdot \vdash v : \sigma_1[\vec{v}, \vec{r} / \vec{r}, \vec{t}] \]
and \( \cdot \vdash \tau_1 : \kappa_i \). From lemma A.4 we get that
\[ \Psi/\text{cd}; \text{cd}; \text{cd}; \cdot \vdash x : \sigma[\vec{v}, \vec{r} / \vec{r}, \vec{t}] \vdash e[\vec{v}/\vec{r}] \]
From lemma A.8 we get that
\[ \Psi/\text{cd}; \text{cd}; \cdot \vdash \cdot \vdash x : \sigma[\vec{v}, \vec{r} / \vec{r}, \vec{t}] \vdash e[\vec{v}, \vec{r} / \vec{r}, \vec{t}] \]
Since \( \Psi/\text{cd} \subset \Psi \) and cd, \( \cdot \vdash \cdot \vdash e \in \text{Dom}(\Psi) \), we can extend the environment for deriving \( e \). Applying lemma A.15 we get that
\[ \Psi ; \text{Dom}(\Psi); \cdot \vdash e[\vec{v}, \vec{r}, \vec{t} / \vec{r}, \vec{t}] \]
which leads to the result.

**Lemma A.15 (Canonical forms)**

1. If \( \Psi; \Delta; \cdot \vdash v : \text{int} \) then \( v = n \).
2. If \( \Psi; \Delta; \cdot \vdash v : \sigma \) at \( \nu \) then \( v = \nu\ell \).
3. If \( \Psi; \Delta; \cdot \vdash v : \sigma_1 \times \sigma_2 \) then \( v = (v_1, v_2) \).
4. If \( \Psi; \Delta; \cdot \vdash v : \exists \kappa.\sigma \) then \( v = (\ell = \tau, v' : \sigma) \).
5. If \( \Psi; \Delta; \cdot \vdash v : \exists \kappa.\sigma \) then \( v = (\ell = \tau, v' : \sigma) \).
6. If \( \Psi; \Delta; \cdot \vdash v : \forall \kappa.\tau[\vec{v}](\vec{\sigma}) \downarrow \downarrow 0 \) then \( v = v'[\vec{\sigma}] \).
7. If \( \Psi; \Delta; \cdot \vdash v : \forall \kappa.\tau[\vec{v}](\vec{\sigma}) \downarrow \downarrow 0 \) then \( v = \lambda(\ell)\tau[\vec{v}](\vec{\sigma})v' \).

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

**Proposition A.18 (Progress)** If \( \vdash \cdot \vdash (M, e) \) then either \( e = \text{halt} \) or there exists a \((M', e')\) such that \( (M, e) \Rightarrow (M', e') \).

**Proof** The proof is again by induction over the structure of \( e \). By definition, \( \Psi; \text{Dom}(\Psi); \cdot \vdash e \). The proof for the individual cases start from this point.

**case** \( e[\vec{v} / \vec{r}] \): From the typing rules, either
\[ v : \forall \kappa.\tau[\vec{v}](\vec{\sigma}) \downarrow \downarrow 0 \] at \( \nu \) or \( v : \forall \kappa.\tau[\vec{v}](\vec{\sigma}) \downarrow \downarrow 0 \)

In the first case by lemma A.17, \( v = \nu\ell \). From the typing rules \( M(\nu\ell) = \lambda(\ell)\tau[\vec{v}](\vec{\sigma})v' \). This implies that we have a reduction.

In the second case, by lemma A.17 \( v = v'[\vec{\sigma}] \). In this case also we have a reduction to \( v'[\vec{\sigma}] \).

**case** \( x = \text{op} \) in \( e \): If \( \text{op} = \text{v} \), then we have a reduction. If \( \text{op} = \pi_i \), then from the typing rules,
\[ \Psi; \text{Dom}(\Psi); \cdot \vdash v : \sigma_1 \times \sigma_2 \]
The required result follows from lemma A.17. In the case of \( \text{put} \), the result follows directly. The constraint \( \nu \in \Delta \) ensures that \( \nu \in \text{Dom}(\Psi) \). In the case for \( \text{get} \), 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. □
— Continuation when copying an existential package —

$$\exists \alpha \forall t_1, t_2, t_3 \{ r_1, r_2, r_3 \} \{ M_r(t_r) \} \Rightarrow 0 \times \alpha_c$$

$$t_k[t] \equiv (\exists t_1 : \Omega. \exists t_2 : \Omega. \exists t_e : \Omega \Rightarrow \Omega. \alpha_c : \{ r_1, r_2, r_3 \}. t_e[t])$$

t at $$r_3$$

— Basic type of the continuations of copy —

— Same as $$t_c$$ but closed with existential packages —

---

— Syntactic type shorthands for notational convenience: —

$$\text{fix}$$

— Continuation when copying an existential package —

$$\text{copy}$$

— First continuation when copying a pair —

$$\text{copypair1}$$

— Second continuation when copying a pair —

$$\text{copypair2}$$

— Continuation when copying an existential package —

$$\text{copyexist1}$$

— Continuation when copying an existential package —

$$\text{copyexist2}$$

1. $$\text{gcend}$$ is the code executed at the end of the toplevel call to $$\text{copy}$$ and just finishes the garbage collection by freeing the from space and calling back the mutator.

2. $$\text{copyexist1}$$ is the continuation of the recursive call to copy an existential package.

3. Since copying a pair requires two recursive calls, we need both $$\text{copypair1}$$ for the continuation of the first call and $$\text{copypair2}$$ 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 $$t_c[t]$$.

---

B. 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 [10].

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 $$t_1, t_c$$ or kind $$\Omega, (\Omega \Rightarrow \Omega)$$, we unify the two into $$t_1, t_2, t_c$$ 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 $$\text{copy}$$:

1. $$\text{gcend}$$ is the code executed at the end of the toplevel call to $$\text{copy}$$ and just finishes the garbage collection by freeing the from space and calling back the mutator.

2. $$\text{copyexist1}$$ is the continuation of the recursive call to copy an existential package.

---

C. Soundness of $$\lambda_{GC \text{forw}}$$

As before, We use the usual exchange and weakening properties of environments without proving them. The code region $$\text{cd}$$ 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 $$\text{wid}$$ construct. The proofs of the propositions for the other constructs follow in a straightforward way by
induction over the derivation.

Because of the \textit{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:

\textbf{Definition C.1} The machine state \((M, e)\) is well formed iff

\[
\frac{M \subseteq M^* \quad \Psi, \text{Dom}(\Psi) : \vdash \varepsilon}{\vdash (M, e)}
\]

Note that to show well-formedness we now need to find not only a witness \(\Psi\) but also a proper \(M^*\) subset.

The reason is that \textit{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 \textit{widen} is a hop.

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

1. First, we construct a \(M^*\) which only contains elements that we can cast (i.e., of type \(M_{\nu}(\tau)\)) and we show that the state is still well-formed. This is the case because when we reach a \textit{widen}, all the live data if of such a type.

2. Then we show that when everything has a type of the form \(M_{\nu}(\tau)\), we can cast every single type consistently to its \(C_{\rho, \rho'}(\tau)\) equivalent and the result is still properly typed. This relies on the subsumption rule on sum types and the fact that \(C\) types only differs from \(M\) types by adding branches to sum types.

\textbf{Lemma C.2 (Region Substitution)}

1. If \(\forall \nu. \Delta ; \Theta ; \Phi; \Gamma \vdash \text{op} : \sigma\) then \(\forall \nu. \Delta; \nu'; \Theta; \Phi[\nu/r]; \Gamma[\nu/r] \vdash \text{op}[\nu/r] : \sigma[\nu/r]\).

2. If \(\forall \nu. \Delta; \tau; \Theta; \Phi; \Gamma \vdash \varepsilon\) then \(\forall \nu. \Delta; \nu'; \Theta; \Phi[\nu/r]; \Gamma[\nu/r] \vdash \varepsilon[\nu/r]\).

\textbf{Proof} Straightforward proof via induction over the derivation of the typing judgment. Note that \(\nu \in \Psi\) is not required here.

\textbf{case let} \(x = \text{widen}[\rho'](\nu)[\tau](v)\) \(e\): We have that \(\Psi; \Delta; \tau; \Theta; \Phi; \Gamma \vdash \text{let} \ x = \text{widen}[\rho'](\nu)[\tau](v)\) \(e\). This implies \(\forall \nu. \Delta; \nu'; \Theta; \Phi[\nu/r]; \Gamma[\nu/r] \vdash \text{let} \ x = \text{widen}[\rho'](\nu)[\tau](v)\) \(e\) since \(M_{\rho}(\tau)[\nu/r] = M_{\rho}[\nu/r](\tau)\).

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

In the other case, \(\Phi[\nu/r][\rho] = \Phi[\rho']\), \(\Phi'\) where \(\text{Dom}(\Phi[\rho'])\cap \text{Dom}(\Phi')\) is empty. Moreover, \(r\) does not occur free in \(e\). Since we can extend environments with new bindings we get that \(\forall \nu. \Delta; \Theta; \Phi[\rho']; \Phi' ; x : C_{\rho, \rho'}(\tau) \vdash \varepsilon\) which leads to the required result.

\textbf{Lemma C.3 (Type Substitution)}

1. If \(\forall \nu. \Delta; \theta; t : \kappa; \Theta; \Phi; \Gamma \vdash \text{op} : \sigma\) and \(\Theta \vdash \tau : \kappa\) then \(\forall \nu. \Delta; \theta; \Phi[\nu/r]; \Gamma[\nu/r] \vdash \text{op}[\nu/r] : \sigma[\nu/r]\).

2. If \(\forall \nu. \Delta; \theta; t : \kappa; \Theta; \Phi; \Gamma \vdash \varepsilon\) and \(\Theta \vdash \tau : \kappa\) then \(\forall \nu. \Delta; \theta; \Phi[\nu/r]; \Gamma[\nu/r] \vdash \varepsilon[\nu/r]\).

\textbf{Lemma C.4 (Value Substitution)}

1. If \(\forall \nu. \Delta; \Theta; \Phi; \alpha; \Delta'; \Gamma \vdash \text{op} : \sigma\) and \(\Psi; \Delta; \Theta; \Phi[\alpha/\sigma]; \Gamma \vdash \varepsilon : \sigma'\) then \(\forall \nu. \Delta; \Theta; \Phi[\alpha/\sigma]; \Gamma[\nu/r] \vdash \text{op}[\nu/r] : \sigma'[\nu/r]\).

2. If \(\forall \nu. \Delta; \Theta; \Phi; \alpha; \Delta'; \Gamma \vdash \varepsilon\) and \(\Psi; \Delta; \Theta; \Phi[\alpha/\sigma]; \Gamma \vdash \varepsilon : \sigma'\) then \(\forall \nu. \Delta; \Theta; \Phi[\alpha/\sigma]; \Gamma[\nu/r] \vdash \varepsilon[\nu/r]\).

\textbf{Proof} Same thing here, the induction hypothesis can always be applied directly.

\textbf{Lemma C.5} Let \(\Psi|_{\nu,M}\) be the restriction of \(\Psi\) to pieces of code or elements of type \(M_{\nu}(\tau)\):

\[
\forall \nu. \Delta; \Theta; \Phi; \alpha; \Delta'; \Gamma \vdash \text{op} : \sigma\) and \(\Psi; \Delta; \Theta; \Phi[\alpha/\sigma]; \Gamma \vdash \varepsilon : \sigma'\) then \(\forall \nu. \Delta; \Theta; \Phi[\alpha/\sigma]; \Gamma[\nu/r] \vdash \text{op}[\nu/r] : \sigma'[\nu/r]\).

\textbf{Proof} By the definition of the \(M\) type, \(\nu\) is either of type \(\text{int}\) or of type \(\tau\) at \(\nu\). The lemma follows trivially if \(\nu\) is of type \(\text{int}\). Consider the other cases.

\textbf{case let} \(x = \text{widen}[\rho'](\nu)[\tau](v)\) \(e\): We have that \(\forall \nu. \Delta; \theta; t : \kappa; \Theta; \Phi[\rho']; \Phi' ; x : C_{\rho, \rho'}(\tau) \vdash \varepsilon\) which leads to the required result.

In the other case, \(\Phi[\nu/r][\rho] = \Phi[\rho']\), \(\Phi'\) where \(\text{Dom}(\Phi[\rho'])\cap \text{Dom}(\Phi')\) is empty. Moreover, \(r\) does not occur free in \(e\). Since we can extend environments with new bindings we get that \(\forall \nu. \Delta; \theta; \Phi[\rho']; \Phi' ; x : C_{\rho, \rho'}(\tau) \vdash \varepsilon\) which leads to the required result.
This implies that

\[
\Psi; \text{Dom}(\Psi); \vdash v' : \exists t. M_v(\tau)
\]

This implies that

\[
v' = (t = \tau', v' : M_v(\tau')).
\]

From here we get that

\[
\Psi; \text{Dom}(\Psi); \vdash v'' : (M_v(\tau'))[\tau'/t].
\]

But we have that

\[M_v(\tau)[\tau'/t] = M_v(\tau'[\tau'/t]).\]

Therefore, applying the inductive hypothesis to the derivation for \(v''\), we get that

\[\Psi; \text{Dom}(\Psi); \vdash v'' : M_v(\tau'[\tau'/t]).\]

The required result follows from this.

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

\[T_{\rho,\nu}(M_v(\tau)) = C_{\rho,\nu}(\tau)\]

\[T_{\rho,\nu}(\forall \bar{\tau}[\tau][\bar{\nu};\bar{\sigma}] \to \rho) = \forall \bar{\tau}[\tau][\bar{\nu};\bar{\sigma}] \to \rho\]

We also use this operator on \(\Psi\) where it is defined as:

\[T_{\rho,\nu}(\Psi) = \{ \nu.\ell : \sigma | \Psi(\nu.\ell) = \sigma' \land T_{\rho,\nu}(\nu.\ell' \to \sigma) = \rho \to \sigma \}\]

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

**Lemma C.8** If \(\Psi; \text{M}_v; \text{Dom}(\Psi); ; \vdash v : M_v(\tau)\), then

\[T_{\rho,\nu}(\Psi; \text{M}_v; \text{Dom}(\Psi); ; v) \vdash \exists t. M_v(\tau)
\]

**Proof** When \(v\) is of type int, the lemma follows trivially.

**case \(M_v(\tau) \to 0\):** In this case \(v = \text{cd}.\ell\) and \(M(\text{cd}.\ell) = \lambda[\bar{\nu};\bar{\sigma}] . e\). This implies that \(v' = \text{left}(M_v(\tau_1) \times M_v(\tau_2))\). This implies that \(M(\nu.\ell) = \text{inl} v'\). We know that

\[
\Psi; \text{Dom}(\Psi); ; v' : (M_v(\tau_1) \times M_v(\tau_2))
\]

This implies that

\[
\Psi; \text{Dom}(\Psi); ; v' : (M_v(\tau_1) \times M_v(\tau_2))
\]

This implies that \(\Psi; \text{Dom}(\Psi); ; v' : (M_v(\tau_1) \times M_v(\tau_2))\).

Similarly, from the typing rule we get that

\[
\Psi; \text{Dom}(\Psi); ; v' : (M_v(\tau_1) \times M_v(\tau_2))\]

By the subtyping rule we can conclude that

\[
\text{left}(C_{\rho,\nu}(\tau_1) \times C_{\rho,\nu}(\tau_2))
\]

This is the required result.

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

**Proposition C.9 (Type Preservation)**

If \(\vdash (M, e)\) and \(\vdash (M', e')\) 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, let \(x = v\) in \(e\)) \(\implies (M, e[v/x])\)**
  - From \(\vdash (M, let x = v in e)\) we get that \(\vdash M' ; \Psi\) and \(\Psi; \text{Dom}(\Psi); ; v \vdash M_x(\tau)\). The derivation in terms tells us that \(\Psi; \text{Dom}(\Psi); ; v \vdash e : \sigma\) and \(\Psi; \text{Dom}(\Psi); ; v \vdash \sigma\to e\). At this point, we can apply value substitution to get \(\vdash \Psi; \text{Dom}(\Psi); ; v \vdash e[v/x]\).

- **(M, only \(\Delta\) in \(e\)) \(\implies (M, \Delta)\)**
  - Here we get \(\vdash \Psi; \text{Dom}(\Psi); ; \Delta \vdash v : M_x(\tau)\). By Lemma C.8 we have that \(T_{\rho,\nu}(\Psi; \text{M}_v; \text{Dom}(\Psi); ; v) \vdash M_v(\tau)\). But we have that \(T_{\rho,\nu}(\Psi; \text{M}_v; \text{Dom}(\Psi); ; v) = \Psi; \text{cd}\) for some \(\Psi'\) since the cast leaves the code type unchanged. Also \(\Psi' = T_{\rho,\nu}(\Psi; \text{M}_v; \text{Dom}(\Psi); ; v)\). Since we can extend the environment, we have that \(\Psi; \text{cd}\vdash \Psi; \text{cd}\vdash v : \text{Dom}(\tau)\) which gives us \(\Psi; \text{cd}\vdash \Psi; \text{cd}\vdash \sigma\to e\). Applying Lemma C.4.2 we get that \(\Psi; \text{cd}\vdash \Psi; \text{cd}\vdash v : \text{Dom} \to e\). This implies that \(\Psi; \text{cd}\vdash \Psi; \text{cd}\vdash \tau' \to \tau\). Then clearly \(M \in M\). This leads to the required result.

- **Lemma C.10 (Canonical forms)**
  1. If \(\Psi; \Delta; ; v \vdash \text{int} \text{then} v = n\).
  2. If \(\Psi; \Delta; ; v \vdash \sigma \to \rho \text{ at } \rho \text{ then } v = \nu.\ell\).
  3. If \(\Psi; \Delta; ; v \vdash \sigma \times \sigma \text{ at } \sigma \text{ then } v = (v_1, v_2)\).
  4. If \(\Psi; \Delta; ; v \vdash \exists \sigma \text{ at } \rho \text{ then } v = \langle t = \sigma, v' \rangle\).
  5. If \(\Psi; \Delta; ; v \vdash \exists \alpha : \Delta, \sigma \text{ at } \rho \text{ then } v = \langle \alpha = \Delta, v' \to \sigma\rangle\).
  6. If \(\Psi; \Delta; ; v \vdash \text{left}(\tau)[\tau][\sigma] \text{ at } \rho \text{ then } v = v'[\tau]\).
  7. If \(\Psi; \Delta; ; v \vdash \text{left}(\tau)[\tau][\sigma] \text{ at } \rho \text{ then } v = v'[\tau]\).
D. Soundness of $\lambda_{GC\text{gen}}$

In this section, we prove the soundness of $\lambda_{GC\text{gen}}$. We will prove soundness in the same way – with the subject reduction and progress lemmas. Since $\lambda_{GC\text{gen}}$ 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 $\Delta[v/r]; \Theta; \Phi[v/r] \vdash \sigma[v/r]$.

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

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

**Proof** Same as lemma A.2. □

**Lemma D.3** $(M_{\rho_1,\rho_2}(\tau))[\tau'/\ell] = M_{\rho_1,\rho_2}(\tau)[\tau'/\ell]$ and $(M_{\rho_1,\rho_2}(\tau))[\tau'] = M_{\rho_1[v/r],\rho_2[v/r]}(\tau)$

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

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

**Proof** The proof is a straightforward induction over the derivation for $\text{op}$. Note that $\Delta, v = (\Delta, r)[v/r]$. It uses lemma D.3 for the case involving subtyping with the $M_{\rho_1,\rho_2}(\tau)$ type.

**case** $(\tau' \in \Delta' = \rho, v)$: We have that

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

This implies that

\[ \Psi; \Delta, v; \Theta; \Phi[v/r]; \Gamma[v/r] \vdash v[v/r] : (\sigma[p/r'])[\tau'][\rho/v/r] \]

This implies that

\[ \Psi; \Delta, v; \Theta; \Phi[v/r]; \Gamma[v/r] \vdash v[v/r] : (\sigma[p/r'])[\tau'][\rho/v/r] \]

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

\[ \Psi; \Delta, v; \Theta; \Phi[v/r]; \Gamma[v/r] \vdash v[v/r] : (\sigma[p/r'])[\tau'][\rho/v/r] \]

□

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

**Proof** The proof is by induction over the derivation of $e$. Note that $\Delta, v = (\Delta, r)[v/r]$. We will consider only the extra cases here.

**case** $\text{open} v$ as $(r', x)$ in $e$: We have that

\[ \Psi; \Delta, r; \Theta; \Phi; \Gamma \vdash \text{open} v$ as $(r', x)$ in $e \]

This implies that

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

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

\[ \Psi; \Delta, v; \Theta; \Phi[v/r]; \Gamma[v/r], x : \sigma[v/r] \vdash r' \vdash e[v/r] \]

From this we get that

\[ \Psi; \Delta, v; \Theta; \Phi[v/r]; \Gamma[v/r] \vdash \text{open} v[v/r] \}

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

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

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

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

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

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

This leads to the required result.

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

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

This implies that

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

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

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

This implies that we must prove that

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

This implies that we must prove that

\[ \Psi; (\Delta, r)[r', r'/r_1, r_2]; \Theta; \Phi[v/r], \Gamma[v/r] \vdash e_2[v/r] \]

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** $\text{ifreg} (r' = r_2) e_1 e_2$: This case follows directly from the inductive hypothesis on the derivations of $e_1$ and $e_2$ and using the fact that $r' \neq r$.

**case** $\text{ifreg} (r = r_2) e_1 e_2$: 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** $\text{ifreg} (r' = r) e_1 e_2$: We have that
We have to prove that Lemma D.8

The proof is again a straightforward induction over the structure of the derivation for $e_2$.

case ifreg ($\nu = r$): We have that

This means that we know that

We have to prove that

This implies that we have to prove that

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

case open ($e$ as $(r, x)$ in $e$): We have that

This means that we know that

We have to prove that

This implies that we have to prove that

But we already know this.

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

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 $\Delta; \Theta; \Phi \vdash \sigma[\tau/\kappa]$.

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

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

Proof The proof is by induction over the derivation for $\sigma$.

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

Proof The proof is again by induction over the derivation for $\epsilon$.

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

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

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

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

case $(r \in \Delta' = \rho, v)$: We have that

and $\Delta'; \Theta; \Phi \vdash \sigma'$. This implies that

This leads to the required result.

Lemma D.12 If $\Psi, \Delta; \Theta; \Phi, \alpha : \Delta'; \Gamma \vdash e$ and $\Delta' \subset \Delta$, then $\Psi, \Delta; \Theta; \Phi \vdash e[\sigma'/\alpha]$.

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

case open ($e$ as $(r, x)$ in $e$): We have that

This leads to the required result.

case ifreg ($r_1 = r_2$): We have that

This leads to the required result.

case open ($e$ as $(r, x)$ in $e$): We have that

This leads to the required result.

Lemma D.13 If $\Psi, \Delta; \Theta; \Phi ; \Gamma \vdash \epsilon : \sigma$ and $\Psi, \Delta; \Theta; \Phi \vdash \epsilon$ then $\Psi, \Delta; \Theta; \Phi, \Gamma \vdash \epsilon[\nu/x] : \sigma$. 

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

This leads to the required result.

By lemma D.1 we have that

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

which leads to the required result.
Lemma D.14 If \( \Psi; \Delta; \Theta; \Phi; \Gamma \vdash v : \sigma \) and \( \Delta_1 \subseteq \Delta \), then \( \Psi; \Delta_1; \Delta; \Theta; \Phi; \Gamma \vdash v : \sigma \)

Proof The proof is by induction over the derivation for \( v \). We will only consider the case for ifreg.

case ifreg \( (r = v) \) \( e_1; e_2 \): By definition we have that
\[
\Psi; \Delta[v/r]; \Theta; \Phi[v/r]; \Gamma[v/r], x : \sigma[v/r] \vdash e_1[v/r]
\]
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
\[
\Psi; \Delta[v/r]; \Theta; \Phi[v/r]; \Gamma[v/r] \vdash v[v/r] : \sigma[v/r]
\]
Substituting \( v[v/r] \) for \( x \) in the derivation for \( e_1 \), leads to the lemma.

case ifreg \( (r_1 = r_2) \) \( e_1; e_2 \): By definition we have that
\[
\Psi; \Delta[r, r/r_1, r_2]; \Theta; \Phi[r/r_1, r_2]; \Gamma[r, r/r_1, r_2], x : \sigma[r, r/r_1, r_2] \vdash e_1[r, r/r_1, r_2]
\]
and \( \Psi; \Delta; \Theta; \Phi; \Gamma, x : \sigma \vdash e_2 \). By Lemma D.4 we know that
\[
\Psi; \Delta[r, r/r_1, r_2]; \Theta; \Phi[r/r_1, r_2]; \Gamma[r, r/r_1, r_2] \vdash v[r, r/r_1, r_2] : \sigma[r, r/r_1, r_2]
\]
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 \( (M, e) \implies (M', e') \) 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 ifreg cases.

case open \( (r \in \Delta = v, v) \) as \( (r, x) \) in \( e \): We know that
\[
\Psi; \text{Dom}(\Psi); \vdash \text{open} (r \in \Delta = v, v) \text{ as } (r, x) \text{ in } e
\]
This implies that
\[
\Psi; \text{Dom}(\Psi); \vdash \text{open} (r \in \Delta = v, v) \implies \exists r \in \Delta. (\sigma \text{ at } r)
\]