Managing Memory with Types

Princeton University CS-TR-640-01

Daniel Chen-An Wang

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
By the Department of
Computer Science

January 2002

Abstract

Important programming language features require specific memory-management techniques. General recursion requires the automatic stack allocation of local variables. Higher-order functions and object-oriented techniques require sophisticated services, such as garbage collection, to manage memory. When memory is a scare resource it is important for the programmer to explicitly control memory management. Unfortunately, a programmer can accidentally destroy important program properties and violate the integrity of the program through memory management related errors.

By using modern type systems, we can expose low-level memory management services to programmers and type-preserving compilers in a way that still guarantees the integrity of the program. We can build more sophisticated memory management services by using these low-level services in a way that provides useful guarantees about program integrity.

I combine existing type systems with several standard type-based compilation techniques to write strongly typed programs that include a function that acts as a tracing garbage collector for the program. Since the garbage collector is an explicit function, there is no need to provide a trusted garbage collector as a runtime service to manage memory. Since the language is strongly typed, the standard type soundness guarantee "Well typed programs do not go wrong" is extended to include the collector, making the garbage collector an untrusted piece of code. This is a desirable property for both Java and proof-carrying code systems.

I describe the technique in detail and report performance measurements for a prototype system as well as present the proofs of type soundness for important subsets of our system, and describe how to use types as a mechanism to manage memory in explicit and safe ways.

Acknowledgments

There are too many people to thank. My mother and father should be given the most credit for having faith in their son, who turned thousand of dollars of education and hours of labor into a thesis about "garbage collection". It is unfortunate that both my parents passed away before they could see me complete my studies. Besides my parents, I have been lucky to know many excellent teachers throughout my life. Peter Lee and Bob Harper are the two outstanding teachers at Carnegie Mellon who started me down the path of programing-language research. My advisor Andrew Appel deserves a great deal of credit for being a merciless and patient critic of all my ideas. He is also a model of how to be excited about doing something no one else has done. All of these people, have done an exceptional job.

Let me not forget those people whose job was not to teach me, but from whom I've learned a great deal. Various friends, roommates, officemates, and generally good company at Princeton who have kept my spirits up and my life interesting: Amal Ahmed, Dirk Balfanz, Matthias Blume, Amit Chakrabarti, Mao Chen, Yuqun Chen, Wagner Correa, Drew Dean, Georg Essl, Wenjia Fang, Jessica Fong, Ben Gum, Minwen Ji, Dongming Jiang, Scott Karlin, Jeff Korn, Sanjeev Kumar, Neophytos Michael, Patrick Min, Robert Osada, Robert Shillner, Yefim Shuf, Kedar Swadi, George Tzanetakis, Dan Wallach, Matthew Webb, and Xiaodong Wen. Let me also not forget the wonderful administrative and technical staff at Princeton who kept things working behind the scenes.

Special thanks go to Greg Morrisett, David Walker, Stephanie Weirich, and Steve Zdancewic who provided a good deal of beer, friendship, and advice about type theory while I was visiting Cornell for the summer. Thanks also to Jon Riecke who volunteered to join my committee. This work was funded in part by Defense Advanced Research Projects Agency grant F30602-99-1-0519 and by the IBM University Partnership Program. Portions of this work were originally published in Wang and Appel[WA01].

To my parents Luan and Ming Kang Wang

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Techniques for managing memory have improved as programming languages have evolved. Languages such as Fortran, which appeared as early as 1952, initially had no support for dynamic allocation primitives. All of the memory used by a program was accounted for and allocated statically before a program was run. This policy was sufficient for numerical programs, but too restrictive in general.

The next major advance in memory management was found in Algol's "pure stack discipline," which allowed the programmer to acquire memory at runtime for temporary variables and reuse the space when the variables were no longer needed. The pure stack discipline allowed for the development of structured programming methodologies and block-structured languages as well as recursion. However, the pure stack model is too limiting for some programs.

One of the most important contributions of Lisp was the introduction of the *garbage collector*, which reclaimed unused "garbage" memory automatically without any programmer intervention. The first garbage collection technique for Lisp was described by McCarthy in 1960 [McC60]. Garbage collected systems allowed languages to support first-class functions and object-oriented methodologies. More expressive memory management techniques allow for better programming language abstractions.

The focus of this work is how to apply modern type systems to provide similar safe memory management facilities in a flexible way. Cardelli informally defines a *safe language* as a language where no untrapped errors occur [Tuc97]. Lisp and Algol are safe languages, while Fortran is not.

I will use safe mobile code systems as the motivating example for my work. However, this work has many applications beyond safe mobile code. The main result of my work is a technique for programming a traditional stop-and-copy garbage collector in a type-safe programming language. To do so efficiently requires the development of several other safe memory-management techniques. I will demonstrate how modern type systems make memory-management techniques that have been too dangerous to use in safe languages explicit and safe.

1.1 Background

Cardelli informally defines a *type* to be an upper bound of the range of values a program variable may assume during program execution [Tuc97]. Languages where one can assign nontrivial bounds to a variable are referred to as *typed languages*. By Cardelli's definitions Fortran and Algol are typed languages. Lisp however is considered "untyped" or more properly unityped, since variables in Lisp can be bounded by a single universal type which makes the bound trivial. Lisp is a safe language. It achieves safety by dynamically verifying operations to prevent any occurrence of untrapped errors. Algol is also safe. Algol rules out many but not all errors through the use of a *type system*. Informally, type systems provide a set of rules to apply to a program that rule out certain errors for any possible program execution. Type-safe languages are safe programming languages that use type systems to rule out many errors.

Memory management errors. Memory-management errors can be benign, lead to abnormal program termination, or lead to silent data corruption. Memory *leaks* occur when one is unable to reclaim storage which is no longer needed. *Dangling references* occur when certain memory cells which may still be referenced by an indirect address are deallocated. Dangling references are not a problem if the program never needs to inspect the memory cells referenced, but in practice most dangling references will eventually lead to either an abnormal program error or data corruption. If after memory cells are deallocated they are reused later, a dangling reference can create a situation where the program is attempting to store two distinct values of different type in the same space. This will eventually lead to data corruption, which will silently cause the program to produce erroneous results. Many languages, such as C and Pascal, that allow programmers to create dangling references through the use of primitives such as `free` or `dispose` are unsafe because they allow for silent data corruption.

Previous approaches to safe memory management. Languages that provide safe memory management have relied on a set of ad-hoc rules associated with the syntax of the programming language to guarantee the safe management of memory. To achieve safe stack allocation in Algol, an assignment to a reference variable must be the location of a variable whose *scope* is the same scope or an enclosing scope of the variable being assigned to. The scope of a variable defines where it is valid to textually refer to a defined variable. Scopes are often nested so that one scope may enclose another. Algol guarantees that all references to locations refer to values earlier in the stack of allocated variables.

These ad-hoc rules are designed to enforce one particular policy for managing the memory. In the case of Algol, the restrictions associate the *lifetime* of variables with their scope. The lifetime of a variable or value is the duration of time during the execution of a program when memory must be reserved to hold the value or the variable. Later we will show one mechanism using type systems that associates the lifetimes of variables with something independent of traditional variable scope. Using another type system, we will show how other values with nested lifetimes can have their memory managed in a stack-like way.

The pure stack model of Algol is too weak to support abstractions such as first-class functions or object-oriented programming, because using the first-class functions or objects typically requires the creation of values whose lifetime escape the scope of the variables that defined them. Typically, Algol like languages may support these features by providing programmers with a way of placing values whose storage escapes the scope of their variables in a separately managed part of memory called the *heap*. Memory allocated on the heap is usually managed explicitly by the programmer. The programmer must explicitly place values whose lifetime escapes the scope of their variables on the heap, and explicitly reclaim memory from the heap when it is no longer in use. This is a difficult task to do correctly. Programming with explicit managment of the heap memory often results in programs that use more memory than necessary or programs which have subtle but catastrophic programming errors related to dangling references.

Lisp was one of the first languages to provide these abstractions in a convenient way through the use of a garbage collector, which managed the memory on the heap automatically. A garbage collector automatically reclaims storage in a way that guarantees there are no dangling references. The absence of dangling references is important to guarantee the safety of a programming language. Garbage collectors safely deallocate memory, by conservatively approximating the set of objects a program may need to properly function. Garbage collectors still may retain objects which the program may never reference in the future, leading to memory leaks.

Semantic garbage. Garbage collectors compute the set of all values that are *reachable* by a program at a particular point during the execution of the program. A value is reachable if from the current point of execution there exists some sequence of dereference operations that the program could perform that would access the value. Just because there is a sequence of dereferences that may lead to a value does not mean that the program will ever perform that particular set of dereferences. Being reachable is a conservative approximation of the stronger property of being *live*.

A value is live during a particular point of program execution if the program will access that value at some point in the future execution of the program. We will call values that are not live *semantic garbage.* we will refer to values that are unreachable simply as garbage. In the literature the word garbage almost always means unreachable and garbage collection refers to the process of reclaiming unreachable values. The distinction between semantic garbage and ordinary garbage makes it clear that there are some reachable values which are semantically garbage. Determining whether an object is semantic garbage or live is an undecidable problem, since we must be able to predict the future behavior of an arbitrary program. Therefore, all automatic memory management schemes have situations where there is some semantic garbage that is not reclaimed [MFH95]. This unreclaimed memory is leaked memory, which is a benign memory management error.

In certain systems leaks can make garbage collectors unusable [Wen90]. In practice, however, leaks are not common enough to make garbage collectors unusable [ADM98]. Leaks are not the only problem with garbage collectors. Garbage collectors either restrict certain compiler optimizations [Boe96] or require the optimizer to provide them with a significant amount of extra information [DMH92]. Also the collector can require a running program to incur significant extra overhead in order to support the collector [TD94]. Collectors are difficult to use in highly concurrent and time-sensitive systems. Until recently, however, garbage collection seemed to be the only practical method of managing memory for safe programming languages with first-class functions or objects.

Managing memory with types. Tofte and Talpin [TT94] describe a system that uses types to prevent memory management errors in a language that supports first-class functions. The system statically segregates objects into "regions" based on automatic type inference. The type of the object tracks which region each object has been assigned to. One can statically approximate which regions a program may need to access. The region system also guarantees that the lifetimes of regions are nested, so they can be managed in a stack fashion. Region-based type system have effectively handled nontrivial ML programs [MLK] and have been adapted for programming languages such as Java [Yat99]. Region systems are not the only mechanism that uses types to manage memory. Linear type systems provide another approach [Wad90, LM92, Bak92a, Abr93, Bak93], by guaranteeing that objects are uniquely referenced. Some garbage collectors use extra type information to reclaim reachable values that are semantic garbage [GG92].

In theory, both region systems and garbage collectors leak memory. However, garbage collectors and region type systems leak memory under different situations, which are sufficiently disjoint that it makes sense to combine the two systems to achieve a system where more memory is reclaimed [Hal99]. One significant problem with region systems is that it is hard for a programmer to determine if a given program will leak memory. Small program transformations can often lead to drastic changes in asymptotic memory consumption.

Garbage collectors are not immune to similar problems but programmers who use garbage collectors typically have an easier time reasoning about what memory is retained. One, can also formalize under what rules certain transformations preserve the asymptotic memory consumption of a program being garbage collected [App92a, pp. 133–145]. Garbage collectors provide programmers with a very simple programming model, which has over the years proven acceptable. However, region systems also have their merits despite their more complex programming model.

The most attractive feature of region-based systems is that their memory management primitives are simple when compared to a garbage collector. The simplicity of these primitives is important. Simple primitives are easy to implement and verify correct. This is especially important in systems where there should be a minimum amount of trusted code, such as safe mobile-code systems.

1.2 Safe Mobile Code

Safe mobile-code systems allow programs to be distributed and run on remote host machines [TSS+97, FGS96]. What makes mobile-code systems different from traditional distributed systems is that host machines in mobile-code systems communicate by sending and receiving programs to execute from other host machines in the system. Hosts in more traditional distributed systems communicate with a set of fixed data messages which have a predetermined meaning and effect.

In mobile-code systems the host machines must be willing to run programs from authors who are unknown or untrusted. Since the authors are unknown the host machine cannot simply accept programs based on a known set of trusted authors.

```
signature CHECKED_FS =
  sig
    exception InvalidHandle

    val openFile : string → int

    (* The following may abort with InvalidHandle *)
    val read  : int → string
    val write : int → string → unit
    val close : int → unit
  end
```

Figure 1.1: An interface that relies on dynamic checking.

The host machine must guarantee that the programs it runs do not behave in a way that violate the host's notion of security [Kat97].

To achieve this a host may dynamically monitor the behavior of the program and abort the program if it attempts to violate security. While dynamic checks are sufficient, they can limit the performance of the overall system. Host systems must export a set of primitives to the running program. Systems that rely on dynamic checks provide inexpressive and complex interfaces, since the interfaces must be designed so that a malicious or buggy program cannot corrupt the host system.

Problems with dynamic checking. Consider the interface exported to a mobile program to access a host's file system. When a program requests the host open a file, the host system typically returns a *file handle* to the program after verifying that the program is allowed to access the file. In systems that use dynamic checking, file handles are often represented as plain integers that are an index in a table that contains private information, such as a cache or buffer for the file. The table is made inaccessible to the program through hardware protection mechanisms. The situation described so far is typical of a Unix environment. The interface exported by the host system can be summarize with the SML [Ull98] signature in Figure 1.1.

The file handle returned by the `openFile` function is an integer. All functions that take the file handle as an argument cannot be sure that the file handle passed to them was not "forged". They cannot be sure whether the file handle is a value that was actually returned by a valid request, or is an arbitrary integer created by a malicious program attempting to perform an unauthorized operation. In such a system each file operation must dynamically verify that the handle is valid and that the request should be allowed to occur; if the handle is invalid the operation will fail and signal an error by raising an exception. In a language without proper support for exceptions our interface would be even more cluttered.

Each operation must check that the file handle is a valid index into the table of private information. Knowing that a given integer is a valid index into the host's private table is not sufficient if the host wishes to execute more than one program at a time. If several programs run concurrently, the host must keep a separate private table for each program. If the host uses a single global table, a program could attempt to gain access to the files of another program by simply trying to guess a valid index. Since we now are maintaining separate private tables for each program, it becomes more complex for cooperating programs to share open file handles since a file handle must always be interpreted as an index relative to a particular program's private table. Capability based systems ameliorate some of these problems, by generating handles to objects that are impossible or difficult to forge via hardware mechanisms or cryptographic techniques [Lev84]. However, these systems are just a unified framework for dynamic checking and are not without their own problems [Gon89].

Using data-encapsulation. Advocates of an object-oriented approach may point out that for the example we have presented there is a natural solution involving data

```
signature OBJECT_FS =
  sig
    (* object type *)
    datatype obj = Obj of {read: unit → string,
                           write: string → unit,
                           close: unit → unit}
    val openFile : string → obj
  end

structure ObjectFS :> OBJECT_FS =
  struct
    fun openFile s = let
      val fhandle  = UnChecked.openFile s (* private to object *)
      fun read  () = UnChecked.read  fhandle
      fun write s  = UnChecked.write fhandle s
      fun close () = UnChecked.close fhandle
    in {read=read,write=write,close=close}
    end
  end
```

Figure 1.2: An interface that uses data-encapsulation

```
signature OBJECT_FS =
  sig
    (* object type *)
    datatype obj = Obj of {read: unit → string,
                           write: string → unit,
                           close: unit → unit,
                            copy: obj → unit}
    val openFile : string → obj
  end

structure ObjectFS :> OBJECT_FS =
  struct
    datatype obj = ...
    fun openFile s = let
      val fhandle = UnChecked.openFile s
      ...
      fun copy (Obj obj) = UnChecked.copy (fhandle, (* ?? *))
    in Obj {read=read,write=write,close=close,copy=copy}
    end
  end
```

Figure 1.3: Failure of data-encapsulation.

encapsulation that avoids the problems of dynamic checking. Figure 1.2 expresses this solution using higher-order functions rather than objects, but the idea is the same. Although we can describe this approach in a typed framework, the notion of data encapsulation is independent of any associated types and relies on lexical scoping. The previously exposed file handle is now kept as hidden state, which only certain functions may access. Our `openFile` function now returns a record of functions that have exclusive private access to the state.

Data encapsulation and binary methods. Consider extending our interface with a special function that reads one open file and writes the entire contents to a different file. Such a function could be implemented using our existing read and write primitives, but we will assume that for this operation a nontrivial performance improvement can be achieved by implementing the function at a lower level. For example, if each open file has a buffer associated with it, we can use the read buffer of one file as the write buffer of another. This avoids the need to allocate an extra buffer, and avoids the need to copy data from one buffer into another. In a multi-threaded environment we may want to guarantee that our primitive copy function is an atomic operation, a property that cannot be achieved by using the existing primitives.

Figure 1.3 is an unsuccessful attempt at adding a copy method to our abstract object. The body of the copy function cannot access the private data held in the closure passed to it, doing so would violate the principles of lexical scope and data encapsulation.

We simply cannot write such a function using strict data-encapsulation techniques. Our copy function must inspect its own private data and the private data of another object simultaneously. In our encoding using higher-order function, this is equivalent to examining a value captured by a closure. Faced with this sort of problem, various object-oriented languages relax the strict data-encapsulation rules, and include notions of "friend functions", "package scope", "protected scope", and "multi-methods".

Using abstract types and type-checking. With the proper use of abstract types we can avoid all of the problems mentioned so far. Consider Figure 1.4 where we have introduced a new abstract type for file handles (`fh`). We can use our type-checker to verify that our program never forges a file handle, and that the only values passed to file operations are values returned from the `openFile` function [ZGM99]. If the program fails to typecheck we refuse to run it.

Now when a file operation is requested we can carry out the operation without the need to verify the validity of the handle. This lets us pass to our client program the actual information needed to access a file, rather than indirecting through a private

```
signature ABSTRACT_FS =
  sig
    type fh     (* new abstract type *)
    val openFile : string → fh
    val read  : fh → string
    val write : fh → string → unit
    val close : fh → unit
    val copy  : fh → fh → unit
  end

structure AbstactFS :> ABSTRACT_FS =
  struct
   type fh = UnChecked.fh (* exported abstractly *)
   fun openFile s     = UnChecked.openFile s
   fun read     x     = UnChecked.read x
   fun write    x s   = UnChecked.write x s
   fun close    x     = UnChecked.close x
   fun copy     (x,y) = UnChecked.copy (x,y)
  end
```

Figure 1.4: An interface that uses abstract types

table. Even when the extra indirection is useful for other reasons we can now avoid the need for duplicating tables in the case of concurrently running programs. This makes the implementation of sharing open files easier, since we can now interpret a file handle with respect to one global table of open files.

We do not claim that all problems can be solved with type checking, but that type checking and other static verification techniques give systems designers important flexibilities not found in other approaches. Dynamic checking, data encapsulation and abstract types are not mutually exclusive ideas. All of these ideas complement each other. Systems must exploit all three to obtain good designs. For example Java relies on all three approaches to guarantee the integrity of the host.

Although Java is a type-safe language, it requires dynamic checks and data-encapsulation techniques to implement its security policy [Gos95]. Operations to create file handles call into a library function referred to as the "Security Manager" which checks if the current security policy allows the program to access a particular file. If the Security Manager allows this to occur client programs receive a file handle which they can directly access. Because of Java's type safety guarantee that no file handles can be forged future file access requests do not need to consult the Security Manager. Avoiding the need to consult the Security Manager for every single file operations such as a read or write requests allows the Security Manager to implement very rich and complex policies for file access.

As well as relying on the standard data-encapsulation techniques seen in other object-oriented languages Java's security model controls the untrusted programs access to library functions via "Class Loaders". Class Loaders determine what library functions are available to an untrusted program and define the binding of globally visible names to actual values. Doing so allows the system to control how programs share values.

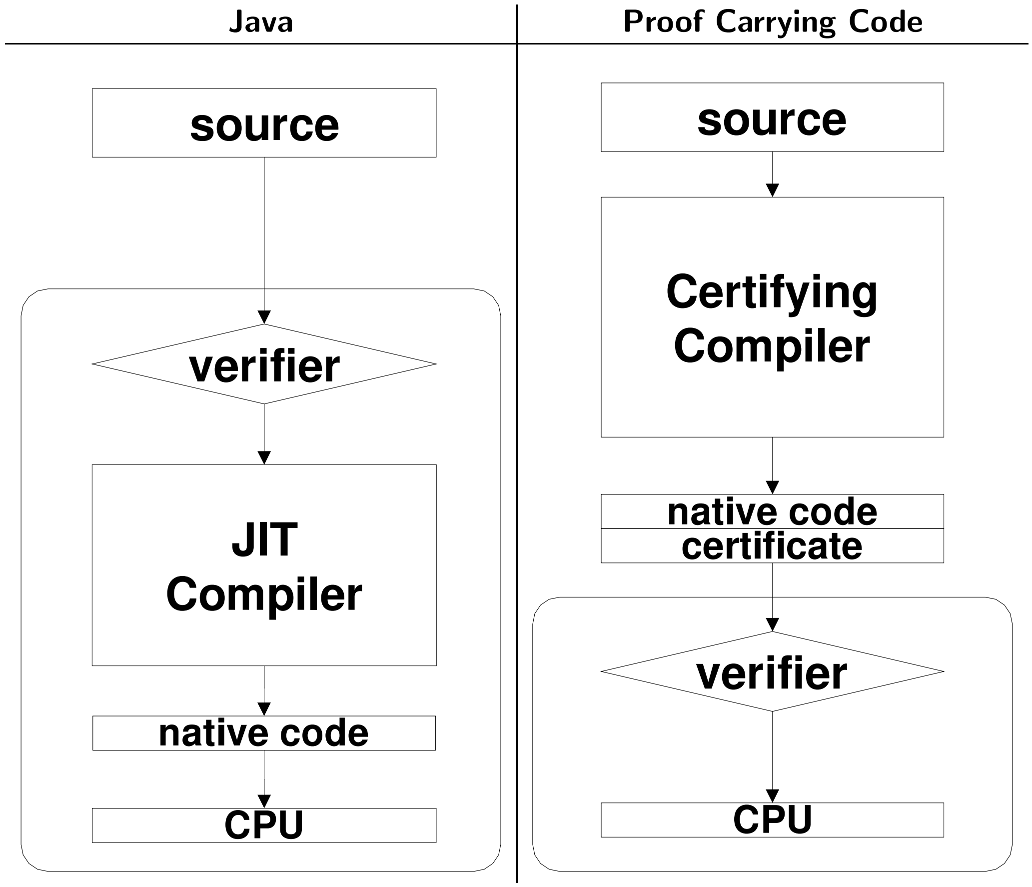


Figure 1.5: Reduced trusted computing base of Proof Carraying Code.

Other safety architectures. Systems such as Proof Carrying Code [Nec97] (PCC) provide static verifiers that do not rely on traditional type checking mechanisms. PCC systems verify properties about low-level optimized machine code by verifying proofs. One advantage of the PCC approach, in comparison to the approach taken by traditional Java systems, is that they have a smaller trusted computing base (TCB), because they reason about optimized native machine code directly and not properties of high-level bytecode.

Figure 1.5 compares the different verification architectures used by an idealized PCC system and traditional Java systems. For all practical purposes, bytecode is equivalent to the original source code. Given a bytecode program one must choose between interpreting the bytecode, which can be an order of magnitude slower, or compiling the bytecode to machine code. Most systems compile the bytecode to native machine code with a "just in time" compiler. However, since the Java bytecode verifier inspects only the bytecode and not the resulting machine code, we must trust that the compiler is correct. If the compiler incorrectly compiles the verified source code, the fact the code has been verified is irrelevant. The CPU is not running the verified source code, but incorrect output from a buggy compiler.

The approach advocated by PCC is to compile the source program with a certifying compiler which produces a machine code executable and a *certificate* of safety. The certificate is a machine checkable proof generated by the compiler that explains to the verifier why the resulting machine code program is safe to run. Since what is run on the CPU is what has been verified, the TCB only contains the verification infrastructure and the CPU which we assume executes our machine code in a way that is consistent with the assumptions used by the safety proof. However, our program must be provided access to some primitive functions to interact with our host. So the TCB also includes the implementation of these primitives.

1.3 Components of the TCB

Figure 1.5 omits a few important details. Programs after being compiled by the compiler or verified in a PCC system are not executed on a bare CPU. The running program is linked with a runtime system that provides the implementation of the various primitives that the host system provides to the running program. These primitives include input and output services and interfaces for managing the memory of the running program through the use of a garbage collector. The runtime primitives and garbage collector are part of the TCB. Both Java and PCC verification approaches

Lines of Source Code

	Trusted Compilers				Certifying Compilers	
	Java2 1.3 SDK		Kaffe	Bullet	Ginseng	FPCC
	Classic	Hotspot 2.1	1.0.6	Train		(estimated)
Optimizing	no	yes	no	yes	yes	yes
Basic TCB	38,000	205,000	38,000	116,000	25,000	$\approx$ 3,000
Primitives	9,000	9,000	7,000	8,000	6,000	$\approx$ 1,000
Collector	7,000	16,000	2,000	11,000	$\approx$ 3,000	$\approx$ 3,000
Total	54,000	230,000	47,000	135,000	$\approx$ 34,000	$\approx$ 7,000
Percent of Total						
Basic TCB	70%	89%	81%	86%	$\approx$ 74%	$\approx$ 43%
Primitives	17%	4%	15%	6%	$\approx$ 18%	$\approx$ 4%
Collector	13%	7%	4%	8%	$\approx$ 8%	$\approx$ 43%

Table 1.1: Comparison of the TCB size of various systems.

implicitly assume that the primitives of the system and garbage collector are correct or at least well behaved. Table 1.1 summarizes the TCB sizes for various systems. All of the systems above translate Java bytecode to native machine code. Some systems rely on trusted compilers and bytecode verifiers[1], while others use certifying compilers and a PCC style verification architecture.

The Java 1.3 Software Development Kit can support several different compilers. The "classic" compiler is a nonoptimizing compiler used by older releases of the SDK. The Hotspot compiler is an alternative optimizing compiler that can be plugged into the Java 1.3 SDK. Hotspot provides a more efficient garbage collector, but shares the implementations of basic primitives used by the classic compiler. The "Bullet Train" [Bul] system also is an optimizing compiler, while "Kaffe" [Kaf] is a less aggressive compiler similar to the classic compiler. "Ginseng" [CLN$^+$00] is an optimizing compiler that produces certified machine code. The quality of code produced by Ginseng should be better than the code produced by the other nonoptimizing compilers.

[1]Kaffe lacks an implementation of a bytecode verifier. Adding one would increase the size of its TCB by roughly another 4,000 lines.

Ginseng can perform optimizations based on dataflow analysis and provide a proof that such optimizations are safe. The FPCC system represents the best estimates for a system based on the ideas of "Foundational Proof-carrying Code" [AF00, App01] system that attempts to aggressively minimize the size of the TCB by using higher-order logic and other semantic techniques. The FPCC, however, does not currently exist as a usable system, so we provide the best estimates of code size based on an incomplete prototype system.

The numbers in Table 1.1 represent the number of lines of source code for various disjoint portions of the TCB, most of which is written in C or C++. Most of the systems include a few routines written in assembly. Larger parts of the TCB of Ginseng and FPCC include a set of axioms that define the logic used in the checking proofs. The logics are specified using the Edinburgh Logical Framework (LF) [PS99].

For purposes of comparison, we consider one line of code written in C, C++, assembly or LF to all be equivalent. In the table the category "Basic TCB" refers to all the code associated with the JIT compiler and verifier, or the verifier by itself in the case of Ginseng and FPCC. The "Primitives" category refers to the code needed to implement the basic `Object` class and support the Java Native Interface [Lia99]. The category "Collector" refers to code related to memory management. All of these numbers are based on a categorization of the source files without a detailed understanding of all the various systems, so they should be viewed as rough approximations.

Line counts and total audit cost. We would like to understand how much effort must be invested to be assured that a given system is indeed secure. Automated tools can help simplify and factor the problem, but in the end we must finally rely on human judgment [Tho84]. Estimating how many person hours must be spent to audit a system may seem like an impossible task. However, every line of code in the system must be inspected. Therefore, all other things equal, a system with more lines of code will take more time to verify than a system with fewer lines of code. Are all lines of code are created equal? Examining a line of assembly code and understanding what effect it has may take drastically more effort than the effort require to examine a line of code written in C.

Many studies of programmer productivity demonstrate that programmers on average produce and debug a constant number of lines of code per day regardless of programming language [Bro75]. This suggests that the ability of humans to reason about programs is relatively language independent. If we assume that the effort involved in auditing a program is proportional to the effort put into writing it and assume that programmers develop lines of code at the same rate regardless of programming language, we can safely mix line counts from different programming languages. Although we are implicitly trusting the C and C++ compilers as well as the assembler and linker [Tho84], we do not include them in the audit cost.

Observations about audit cost. Ginseng's TCB is roughly half the size of the nonoptimizing Java systems such as the Sun classic compiler and Kaffe. For systems where the performance of these simple compilers is sufficient, the effort in carefully auditing a standard compiler may not be any more difficult than auditing the verification infrastructure of a system like Ginseng. The FPCC approach may significantly reduce this auditing effort. However, in the case where we wish to have high performance with strong safety guarantees, the certifying compiler approach has a clear advantage to the approach of using a trusted compiler. We can see that trusted optimizing compilers balloon the TCB. Systems using certifying compilers can provide optimizing compilers with an auditing effort roughly equivalent to or significantly less than the auditing a nonoptimizing system. The number of lines needed to implement the primitives for systems with trusted compilers are all roughly the same.

Trusted garbage collectors. For systems that use trusted compilers, the garbage collector is a small but significant percentage of the TCB. For systems that use certifying compilers, the garbage collectors are equally a small percentage. However, as we see, optimizing compilers need more complex collectors. So as we add more optimizations to our collector, it becomes a larger portion of the TCB. For an aggressive FPCC-based system even a simple collector ends up being almost half the TCB. As new techniques such as FPCC reduce the TCB in other ways, the collector becomes a significant source of complexity.

1.4 Type-preserving Garbage Collectors

The main result of this dissertation is an approach to remove garbage collectors from the TCB of safe mobile code systems. Using ideas from existing type systems we will construct a *type-preserving garbage collector*. We can guarantee that the collector preserves the types of the running program's data structures. Traditionally a collector is a primitive runtime service, outside the model of the programming language. The type safety of running programs depends on the assumption that the collector does not violate any program invariants. However, constructing proofs of correctness for actual systems require a great deal of work. Our type-preservation guarantee does not guarantee correctness but makes us more confident that our system is likely to be correct, even when security is not our primary concern.

Proving correctness. Garbage collectors are typically written in low-level unsafe languages such as C. Most garbage-collector algorithms discuss details in terms of low-level bit and pointer manipulation operations. Morrisett, Felleisen et al. [MFH95] present a high-level semantics for garbage collection algorithms and prove the correctness of various well known algorithms. However, in their semantics, garbage collection is still viewed as an abstract operation that lies outside of the underlying language being garbage collected. This approach allows them to discuss the purely algorithmic issues without revealing the underlying implementation details.

We are only aware of one attempt to formally verify the correctness of a garbage collected system. Swarup et al. [GMR+92] first specify the behavior of an deterministic abstract machine for Scheme, called the Microcoded Stored Byte Code Machine (MSBCM). They also define the semantics for another abstract machine called the Garbage-collected Stored Byte Code Machine (GSBCM). Next they define a relation ($\simeq$) between the two machines. Their main theorem is that given initial related states the two machines produce related results. They define a partial function on the states of the GSBCM called GC which has the following property

$$\begin{aligned} S_{GSBCM} &\simeq S_{MSBCM} \\ GC(S_{GSBCM}) &\simeq S_{MSBCM} \end{aligned}$$

That is the relation $\simeq$ is preserved by the GC function. The GC function is a Cheney-style [Che70] BFS copying collector specified in roughly 120 lines of Pre-Scheme [RFG+92]. The main lemma that establishes the preservation of the state correspondence requires eight pages of formal mathematics and English. The whole proof of the correspondence between the MSBCM and GSBCM is twenty-six pages of formal mathematics.

Nettles [Net92] provides an informal proof of correctness for an abstract BFS Cheney style collector. He describes an abstract correctness criterion for a garbage collector using the Larch Shared Language (LSL) [GH93] which is a machine checkable specification language. His specification of a correctness criterion is thirteen pages of LSL specification. A function that meets the specification is guaranteed to have copied all reachable data memory from a root set into a new memory, in such a way that the copy is isomorphic to the original data. Nettles then provides an informal proof that an eighty line collector, written in C, meets the specification. The informal proof occupies eight pages of text. Nettles's result guarantees only that the collector will properly copy all data reachable from a root set. To establish correctness of an entire system, we must be assured that the root set passed to the collector is the right set of roots. Also, we are implicitly assuming that the program whose memory is being garbage-collected can tolerate the fact that the collector has moved objects to new locations. The results of Swarup et al. explicitly establish both these facts for their system.

As we can see, establishing the correctness of very simple garbage collectors requires a good deal of nontrivial effort. In both cases the proofs provided are not directly machine-checkable. Building machine checkable proofs for a higher level of assurance is likely to be a serious undertaking even for the small systems above. For realistic systems that are at least two orders of magnitude bigger than the two toy collectors described above, we believe than even informal verification will be impractical.

There are many implicit invariants that must be maintained for the garbage collector to properly function. So even if we are willing to trust that a particular garbage collector is correctly implemented, we should be suspicious whether all the implicit invariants the garbage collector relies on are being observed by our program. Even *conservative garbage collectors*, which have relatively simple interfaces, require that subtle invariants must be preserved [Boe96].

One attractive feature of region-based type systems is their simple runtime primitives. Ideally, we would like to use a region-based type system to manage memory in PCC systems [CWM99]. There already has been some progress made in this direction. However, regions by themselves have not proven to be a particularly efficient way to manage the same class of programs that garbage collectors currently do and have other undesirable features when compared to garbage collectors. Can we have the simplicity of region primitives yet all the benefits of a garbage collector?

Proving type preservation. Proving correctness, as we have shown, is quite difficult. For safe mobile-code systems the correctness of the collector is important but not necessary to guarantee safety. Mobile-code systems rely on the integrity of the type system to safely export objects to the client system. If we can guarantee that the collector does not compromise the guarantees of the type system, our verification task may be simpler. Weakening our criteria in this way means that some garbage-collection bugs may still exist, but that these bugs do not compromise the type system.

Our approach is simple: make the collector a well typed function written in the same typed intermediate language used by the compiler of the *mutator's* source language (the mutator is the program whose memory is being managed by the collector). Garbage collection is no longer a primitive runtime service, uses no unsafe primitives, and is part of our model of the programming language. Since the collector and mutator are both well typed functions, if the underlying language is type safe, we are guaranteed that the collector cannot compromise the type system. Our language uses a region-based type system for safe primitive memory management. The collector is built on top of these safe region primitives. Regions are used to implement the semispaces of a traditional copying collector. The region type system allows us to verify that it is safe for the collector to deallocate a semispace that contains only garbage. Chapter 4 describes our approach in detail.

Does type preservation guarantee value preservation? One might suspect that we have weakened the problem so much that the guarantees it provides are of little practical value. Type preservation does not immediately imply that our collector preserves the values of objects, just their type, but in richer type systems that include dependent and singleton types [XP99, Cra99], type preservation does imply value preservation. So in situations where value preservation is a critical property needed to maintain the security of the system, we can formulate the value-preservation problem as a type preservation problem in a singleton or dependently typed framework.

Comparison to region inference. Region systems and garbage collectors are incomparable in terms of their efficiency at reclaiming unused storage. Our collector dynamically traces values at runtime, allowing for more fine-grain and efficient memory management than systems that use region inference alone, which in certain cases may take asymptotically more space than a simple tracing garbage collector. From a different perspective, our collector is merely a particular way of writing programs in a language that uses regions as the primary memory management mechanism; with this perspective our work is simply a more efficient way of utilizing existing safe region-based memory management primitives, similar to the "double copying" technique used to make certain region programs more efficient [TBE+98].

Formal treatment of collector interfaces. Many details of garbage collector interfaces can be described in a high-level and type-safe way, using simple and standard typing constructs. In particular we describe one way to implement "stack walking" [DMH92] without an explicit table that maps the return address of a function to a stack frame layout. We are able to do this by encoding the table implicitly and in a checkable way.

We can catch interface bugs, such as the failure to include a live value in the root set or providing incorrect type information, at compile time. Statically catching these bugs makes the system more secure, easier to debug, more flexible, and potentially more efficient. Since the collector is not a fixed trusted piece of the system, individual programs can provide a specialized collector which may improve program performance.

1.5 Technical Challenges

Building a type-preserving collector does not rely on a single key technical advance, but results from the combination of several advances in typed compilation. The key issues that need to be addressed are safe explicit deallocation, copying, source language abstractions, and pointer sharing.

Region type systems. Collectors must use some primitive memory management service to allocate and deallocate memory. We must verify that the service used by the collector is safe. The work on type-and-effect systems done by Tofte and Talpin, and refined by others, provides type-safe explicit memory management [TT94, AFL95, CWM99, BHR99]. We can use the memory-management primitives provided by a region system to guarantee that the memory-management primitives used by our collector are safe. The region type system guarantees that the memory reclaimed by our collector is in fact garbage.

However, the original region-based type systems are not sufficiently expressive for our purposes, and they contain extra features which are not needed for our purposes.

In Chapter 3 we present a simplified region type system, and describe its formal semantics, and sketch the type soundness of the language. The full proof can be found in Appendix A.

Type-safe tracing. If the static type of every object is known at compile time, it is easy to write a well typed function that produces a copy of the object with the same type. However, when the type is not known at compile time, because of parametric polymorphism or issues of separate compilation, this task becomes more challenging. Fortunately, work in the area of intensional type analysis [CWM98, CW99, TSS00, STS00] and other forms of ad-hoc polymorphism that use dictionary passing [WB89a] provide type-safe solutions to this problem.

Dealing with data-abstraction. Traditional collectors violate data-abstraction guarantees that are present in the source language. The "private" fields of an object in Java or "private" environment of a closure in ML cannot remain private to the garbage collector. We must decide whether we wish to preserve these abstraction guarantees or violate data abstraction when performing garbage collection.

There are several well known techniques for type-preserving closure conversion [MMH96, MWCG98, TO98]. Many of the schemes provide strong guarantees that they preserve source level abstractions. In practice many compilers still must provide extra type information that describes the layout of "abstract" objects for the garbage collector, so claims of abstraction preservation break down at the level of the garbage collector. Other closure conversion techniques for first-order target languages [TO98] provide much weaker abstraction-preservation guarantees and make the layout of closures explicit during translation. Intensional type analysis formalizes the passing of extra type information (typically provided by the compiler for the garbage collector) in a fully type-safe way [CWM98]. We touch on some of the tradeoffs of these approaches in Chapter 8.

Forwarding pointers. Pointer sharing is preserved by the use of forwarding pointers which provide an efficient way to implement a map from pointers in the from-space to pointers in the to-space. This map is needed to copy an arbitrary graph of heap objects from one space to the other. Any map, such as a hash table, can be used in place of forwarding pointers. Dealing with forwarding pointers complicates reasoning about safety, but we outline one approach for dealing with forwarding pointers in a safe way. Our approach requires some inelegant ad-hoc reasoning, but our technique is as efficient as current unsafe techniques and can be formally proven sound. We discuss forwarding pointers in Chapter 5.

Stack allocation. Chapter 6 describes our experience with building a safe collector using our techniques. Unfortunately, our collector results in some performance penalties compared to other collectors. In Chapter 7 we address the chief source of the inefficiency, the inability to stack-allocate activation records. Using a linear type system, we will demonstrate how to safely perform the stack allocation of activation records, as well as other common space-saving optimizations used by garbage collectors that involve destructive update.

1.6 Contributions

The contributions of this dissertation are the following:

- We prove the soundness for a simplified region calculus that is more expressive the previous region calculi (Chapter 3).

- We demonstrate how to construct tracing garbage collectors that can be verified, through static type checking, not to violate any typing invariants of the mutator. Another important contribution of the work is the ability to think about garbage collector interfaces in a statically checkable way (Chapter 4).

- We formally sketch the needed type theory to properly support forwarding pointers (Chapter 5).

- We report the performance of a simple type-preserving garbage collector (Chapter 6).

- We demonstrate how adding linearity constraints allows us to support stack allocation of return continuations as well as other advanced control structures in provably safe ways (Chapter 7).

- We discuss the extensions needed in a type system to fully support separate compilation (Chapter 8).

Chapter 2

Traditional Techniques

2.1 Introduction

In order to better understand the issues involved, it is useful to understand how memory is managed traditionally. In this chapter, we will demonstrate several memory management techniques for a simple list reverse program and describe the various tradeoffs between them. To make things concrete, we will provide examples written in C [KR88], which has a sufficiently low-level semantics to illustrate many important issues.

After describing a simple program in C, we will discuss various ways to manage its memory use. Our first approach will use low-level explicit memory management. The explicit approach, while efficient, is fragile and requires a global program invariant that is difficult to maintain. Our second technique uses local invariants that are easier for the programmer to maintain, but which are still unchecked. Finally, we will use a garbage collector that requires several invariants both global and local to be maintained. However these invariants can be maintained automatically by the compiler, so that the programmer need not worry about them.

List reverse in C. The program for which will serve as our example is a simple list reverse program. We begin by defining our representation of lists with the following type declaration

```
typedef struct lst_s *lst;
struct lst_s { int hd; lst tl; };
```

A value of type `lst` is a pointer to a two element record whose first element (`hd`) is an integer and second element (`tl`) is a list. Since a list is represented as a pointer to a record, we will represent the empty list as the `NULL` pointer. To make our code more abstract we define the following `lst` constant and constructor function.

```
const lst Nil = NULL;
lst Cons(int hd, lst tl) {
  lst t = malloc(sizeof(struct lst_s));
  t->hd = hd; t->tl = tl;
  return t;
}
```

The constant `Nil` is simply the `NULL` pointer. The function `Cons` takes a head and a tail and returns a new list after allocating space from the system heap to hold the new list cell[1].

We define our list reverse function as the following tail-recursive function

```
lst itrev(lst l, lst acc) {
  if (l == Nil) return acc;
  else {
    int x = l->hd; lst xs = l->tl; /* deconstruct list */
    lst new_acc = Cons(x,acc);
    return itrev(xs,new_acc);
  }
}
```

[1] We are omitting some error handling code, since our constructor does not properly handle the case when memory is exhausted.

The first argument (`l`) to our function is the list to be reversed. The second argument (`acc`) is the accumulated reversed list. If we have completed reversing `l`, i.e. when `l == Nil`, we simply return the list stored in `acc`. Otherwise, we place a new cell on the accumulated list and recursively reverse the remaining list with the new accumulated list. Since our function is tail recursive a good optimizing compiler should reduce the code to a simple iterative loop.

Our program calls the function `itrev` in the following way

```
int main(int argc, char **argv) {
  lst l  = mklst(10);
  lst rl = itrev(l,Nil);
  prlst(rl);
  return 0;
}
```

The function `mklst` makes a list of integers, which we implement with the following recursive code:

```
lst mklst(int n) {
  if (n == 0) return Nil;
  else {
    lst l = mklst(n - 1);
    return Cons(n,l);
  }
}
```

The function `prlst` simply prints the list and is implemented as follows

```
void prlst(lst l) {
  if (l == Nil) return;
  else {
    int x = l->hd; lst xs = l->tl;  /* deconstruct list */
    printf("%d ",x); prlst(xs);
  }
}
```

2.2 Manual Memory Management

In our example program no space is ever deallocated, so our program will hold on to all of the cells in `l` and `rl`. This is twice as much space as needed. If we think about the behavior of `itrev` carefully, we can do better. After deconstructing a list cell into its head (`x`) and tail (`xs`) that particular list cell is no longer needed. We can immediately recycle the space used and reduce our space usage by implementing `itrev` as follows

```
lst itrev(lst l, lst acc) {
  if (l == Nil) return acc;
  else {
    int x = l->hd; lst xs = l->tl; /* deconstruct list */
    l->hd = x; l->tl = acc;        /* reuse l for new cell */
    return itrev(xs,l);
  }
}
```

Also we can modify `prlist` so that it immediately returns space to the system after it prints each list cell.

```
void prlst(lst l) {
  if (l == Nil) return;
  else {
    int x = l->hd; lst xs = l->tl; /* deconstruct list */
    free(l);                       /* reclaim storage */
    printf("%d ",x); prlst(xs);
  }
}
```

Programmers concerned about space consumption will reason at this level to reuse memory. However, these sorts of space optimizations require certain global invariants hold for them to be safe. In our program, every list cell is pointed to by exactly one pointer.

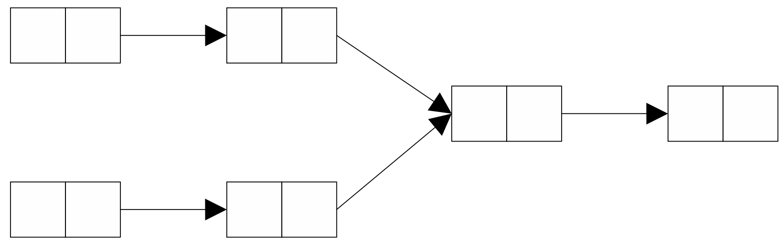


Figure 2.1: A shared list tail.

Consider Figure 2.1, where we have two lists that share a common tail, which violates our single pointer invariant. If we applied our new list reverse function to one of the lists, we would destroy the values of the other list. This will cause our program to behave in unexpected ways. Consider the result of passing a shared list to our `prlst` function; `prlst` will create dangling pointers.

The single pointer invariant is a global program property that is difficult for the programmer to maintain. Programming with a single pointer invariant maybe inefficient, because it often requires the copying of data to maintain the invariant. What's worse, programming languages like C provide no support to help the programmer catch violations of the invariant. So this programming style is unsafe in a language like C. However, there do exist languages that allow the programmer to program safely with this style of programming through the use of linear type systems [Wad90, Bak92a, Hof00].

2.3 Arena Allocation

For programs that are allocation intensive, it is often better to use a storage allocator based on *arenas* or *memory pools* [Ros67, WJNB95]. These are contiguous sequence of memory for which allocation simply requires incrementing a pointer. Objects placed in an arena cannot be deallocated individually. All the objects placed in an arena are

```
typedef struct arena_s *arena;
arena arena_new(void);
void* arena_alloc(arena sp, unsigned nbytes);
void  arena_free(arena sp);
```

Figure 2.2: A simple arena allocation interface.

deallocated at once when the entire arena is deallocated. Figure 2.2 describes a typical arena-based allocation interface. Figure 2.3 provides a simple implementation of the interface. Figure 2.4 depicts the representation of arenas used by the implementation.

An arena is represented as two pointers and a list of `hunks` that are a list of allocated contiguous regions of memory. The `alloc` pointer points to the next unallocated address of memory. The `limit` pointer points to the last address that we are allowed to allocate in the current hunk of memory. If we are unable to service a request for data we simply allocate a new hunk large enough to service such a request and retry the request. Notice, that our `arena_free` function takes time linear in the number of hunks to deallocate an arena. A more complex implementation, using doubly linked lists, can reduce this cost to a constant factor. The implementation we present is simple and easily verified correct.

Using our new arena allocation scheme will require us to parameterize our list constructor and any other allocating function with an arena within which to request memory from. Our new list interface needs to be modified as below

```
const lst Nil = NULL;
lst Cons(arena a, int hd, lst tl) {
  lst t = arena_alloc(a,(sizeof (struct lst_s)));
  t->hd = hd; t->tl = tl;
  return t;
}
```

Since the `itrev` function allocates it must be modified as follows

```
#define PTR_ADD(x,y) (((char *)(x))+(y))
#define HUNK_SIZE 4096
typedef struct hunk_s *hunk;
struct hunk_s  { void *start; hunk rest; };
struct arena_s { hunk hunks; void *alloc; void *limit; };

static hunk add_hunk(unsigned sz,hunk rest) {
  hunk h = malloc(sizeof(struct hunk_s));
  h->start = malloc(sz); h->rest = rest;
  return h;
}
static void free_hunks(hunk h) {
  if(h == NULL) return;
  else {
    void *start = h->start; hunk rest = h->rest;
    free(h); free(start); free_hunks(rest);
  }
}

arena arena_new(void) {
  arena a  = malloc(sizeof(struct arena_s));
  a->hunks = add_hunk(HUNK_SIZE,NULL);
  a->alloc = a->hunks->start;
  a->limit = PTR_ADD(a->alloc,HUNK_SIZE);
  return a;
}
void* arena_alloc(arena a, unsigned sz) {
  void *alloc = PTR_ADD(a->alloc,sz); void *ret = a->alloc;

  if (alloc <= a->limit) { a->alloc = alloc; return ret; }
  else {
    unsigned h_sz = (sz > HUNK_SIZE ? sz : HUNK_SIZE);
    a->hunks = add_hunk(h_sz,a->hunks);
    a->alloc = a->hunks->start;
    a->limit = PTR_ADD(a->alloc,h_sz);
    return arena_alloc(a,sz);
  }
}
void arena_free(arena a) { free_hunks(a->hunks); free(a); }
```

Figure 2.3: An Iimplementation of the arena interface.

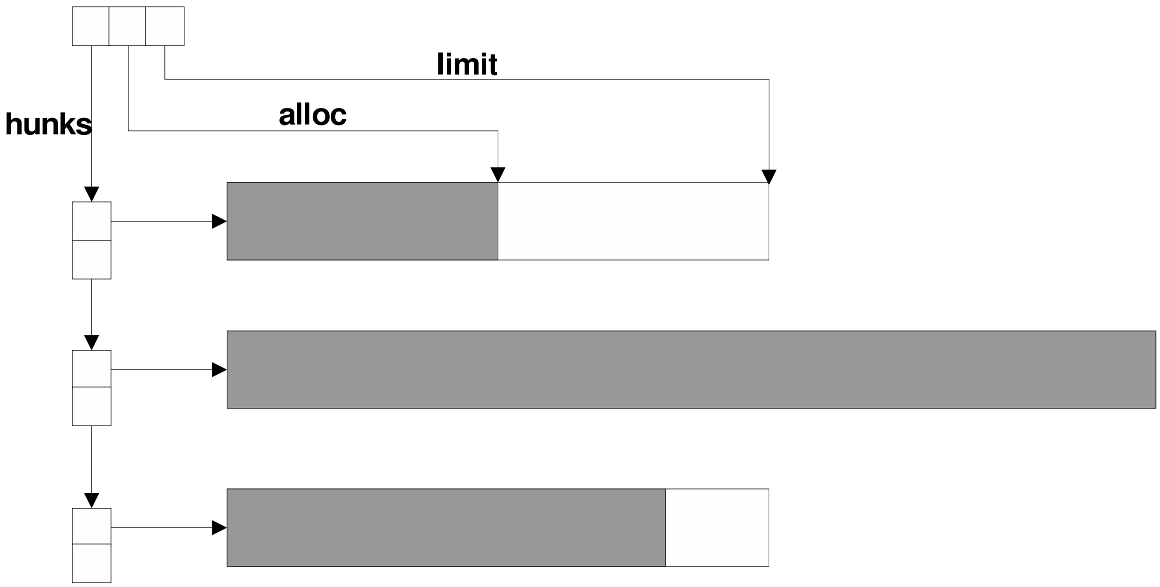


Figure 2.4: Representation of an arena.

```
lst itrev(arena a, lst l, lst acc) {
  if (l == Nil)  return acc;
  else {
    int x = l->hd; lst xs = l->tl;
    lst new_acc = Cons(a,x,acc);
    return itrev(a,xs,new_acc);
  }
}
```

Likewise our function to construct our initial list is updated to the following

```
lst mklst(arena a, int n) {
  if (n == 0) return Nil;
  else {
    lst l = mklst(a, n - 1);
    return Cons(a, n, l);
  }
}
```

Now that we have parameterized `itrev` and `mklst` with an allocation arena, we can arrange to allocate the list created by `mklst` in a different arena from the one used by `itrev`. We can rewrite our program to take advantage of this extra flexibility as seen below

```
int main(int argc, char **argv) {
  arena a1 = arena_new();     /* allocate a1 */
  arena a2 = arena_new();     /* allocate a2 */
  lst l    = mklst(a2,10);
  lst rl   = itrev(a1,l,Nil);
  arena_free(a2);             /* deallocate a2 */
  prlst(rl);
  arena_free(a1);             /* deallocate a1 */
  return 0;
}
```

Our program creates two different allocation arenas `a1` and `a2`. It uses `a1` to hold the list created by `itrev`. It uses `a2` to hold the list created by `mklst`. Since after `itrev` returns, we never need to inspect the list created by `mklst`, arena `a2` can be immediately deallocated.

Unlike our previous attempts to manage memory, our program does not require that a single pointer invariant be maintained. So we can safely use lists with shared tails. Our arena-based system may also run significantly faster, since we are able to amortize the cost of expensive allocation and deallocation operations. The memory used by the list `rl` will be deallocated in several large hunks.

However, arena allocation techniques do not guarantee error free memory management. Consider, the following simple modification to our program

```
int main(int gc, char **argv) {
  arena a1 = arena_new();
  arena a2 = arena_new();
  lst l    = mklst(a2,10);
  lst rl   = itrev(a1,l,Nil);
  arena_free(a2);
  prlst(l); /* print original list also */
  prlst(rl);
```

```
  arena_free(a1);
  return 0;
}
```

The extra call to `prlst` is in error, since we have deallocated the space associated with the list `l` when we freed arena `a2`. Region-based type systems can be used with arena-based allocation techniques to prevent these errors. More surprisingly, certain region-based systems can automatically convert naive programs, such as our original program, into ones that use arena-based allocation [TT94].

2.4 Garbage Collection

There are a wide spectrum of garbage collection techniques. Wilson [Wil92] and Jones [Jon96] provide comprehensive description for many of the widely used techniques. However, both neglect the details of how garbage collectors are interfaced with the programs being garbage collected. The details are omitted because they have little impact on the algorithmic structure of the garbage-collection algorithms. However, for actual implementations these details can be quite complex. Our main purpose is to review how a simple collector is implemented, and illustrate the complexity of the interface between a collector and program.

The underlying algorithm of our garbage collector will remain the same, but we will discuss a range of different possible interfaces between the collector and program. We will describe a precise stop-and-copy DFS collector. Our presentation of our collector differs from the traditional literature, in that it uses techniques that are inefficient in time and space, but only by constant factors. We hope to emphasize important algorithmic details rather than using more clever encodings and implementations that obscure important abstractions.

Sources of interface complexity. Garbage collectors interfaces are complex, because the program whose memory is being managed must communicate to the collector information to locate the set of root variables that point to live data and allow the collector to recursively traverse all values reachable from the root variables. Identifying the set of live root variables is difficult, because it requires compiler support. Optimizing compilers must be careful to preserve interface invariants and communicate information to the collector that is changed by optimization.

Accessing the stack frame. When a function calls another function it must save local variables. These variables are saved on the stack. The local variables are potential roots. Since some compilers will save all local variables on the stack, including those that are not needed after the call returns, a collector that assumes all stack variables are roots may retain more live data than necessary. A compiler can emit extra information to a collector that identifies dead variables which are allocated on the stack frame. Some calling conventions allow functions to place local variables in machine registers whose values are guaranteed to be preserved across a call. The garbage collectors must be able to locate these register local variables also.

Figure 2.5 describes an abstract interface to stack frames sufficient for implementing a garbage collector. It is based on the work of Ramsey et al. [JRR99], who describe a minimal interface for building portable runtime systems. We use it to give an explicit model of how a collector and mutator interact without revealing too many low-level details.

Our interfaces makes several simplifying assumptions on how code is compiled and laid out by the compiler and linker. We assume all local variables are saved into a stack frame and not stored in callee-saved registers. Our simplification of Ramsey et al.'s interface imposes many more restrictions on how code maybe compiled. To properly

```
typedef struct frame_s* frame;
frame CallerFrame(void);
frame NextFrame(frame f);
void* FindVar(frame f, int i);
void* GetDescriptor(frame f);
void  SetDescriptor(void *f,void *datum);
```

`CallerFrame` Return the stack frame of the caller of the current function or `NULL` if there is no caller.

`NextFrame` Return the next frame in the chain of stack frames, or `NULL` if we have reached the end.

`FindVar` Return the address of a local variable in a given frame or `NULL` if the variable is not live. Local variables are indexed beginning from 0, based on the textual order of their appearance in the source code. The first declared argument to a function has index 0.

`GetDescriptor` Return the frame descriptor associated with this frame. (Used to propagate type information.)

`SetDescriptor` Set the frame descriptor associated with the function whose address is `f`. (Used to propagate type information.)

Figure 2.5: Abstract interfaces to stack frames.

implement such an interface requires cooperation from the compiler. The compiler must supply stack layout and variable liveness information to properly implement `FindVar`. The compiler must also arrange for the program code to be laid out in a way such that the return address on the stack can be use to identify the calling function. Those interested in more details should read [JRR99].

Traversing values. Once we are able to examine stack frames and locate all the potentially live variables, we must be able to traverse all reachable values from our root set. The function `FindVar` returns a pointer to the address of a given variable, but does not provide us with enough type information for us to distinguish between an integer value and a pointer to a list. The approach to this problem adopted by some systems is to use a variety of different tagging schemes [SH87, App92b, DEB94]. There are several efficient software and hardware-based approaches for tagging objects. However, all of these approaches introduce extra overhead. Tagging objects also makes it difficult to interoperate across languages.

Another approach that avoids the needs for tags is to use type information available at compile time to identify the type of variables. These *tagless garbage collectors* [App89, DMH92], need a way to communicate static type information to the garbage collector. Our interface to stack frames provides us the necessary hooks to communicate this type information to our garbage collector. By examining the return address stored in a stack frame we can discover which function created a given frame [App89, DMH92]. Given a mapping from stack frames to the functions that created them, we can associate arbitrary data to any given stack frame. The two functions in our interface that allows this are `SetDescriptor` and `GetDescriptor`. The `SetDescriptor` functions binds some arbitrary piece of data with a stack frame based on the creating function, whose address is passed as its first argument. The original interface by Ramsey et al. uses a more flexible mechanism to establish the binding.

To make our discussion concrete, we will demonstrate how to build a simple collector for our list reverse program. We will start with the arena allocation version of our list reverse program, since we will need control over where objects are allocated. Our collector at first will be tagless so that we do not need to change the representation of our values to accommodate tagging. Later, we will modify our program to use a relatively simple tagging scheme.

Interfacing with the collector. An important issue to decide is when the collector will be invoked. We could arrange to have every use of the `Cons` primitive determine if the collector should be invoked before allocating any memory. However, such an approach would slow down the rate at which we could allocate data. Another approach is to place checks for when to invoke a garbage collector in our program in a way so that our program can only perform a bounded amount of allocation before executing a check.

At each point where a collection may occur, we must be able to discover all the program roots and have the appropriate type information needed by the collector. These program points are called *safe points* in the garbage collection literature. The occurrence of a safe point restricts the set of optimizations that a compiler may do across the safe point [JRR99].

Since our garbage collector is tagless we need to provide it with type information that describes the layout of every stack frame. We will describe the layout of each stack frame with a string. The character `'_'` represents a non-pointer variable that does not need to be traced and the `'l'` will represent a variable of type `lst`. This encoding can be extend to handle other types that need to be traced and is simple

```
void gc(arena *from);
int need_gc(arena a);

...

lst itrev(arena a, lst l, lst acc) {
  if(need_gc(a)) gc(&a); /* safe point */
  if (l == Nil) return acc;
  else {
    int x = l->hd; lst xs = l->tl;
    lst new_acc = Cons(a,x,acc);
    return itrev(a,xs,new_acc);
  }
}

lst mklst(arena a, int n) {
  if(need_gc(a)) gc(&a); /* safe point */
  if (n == 0) return Nil;
  else {
    lst l = mklst(a, n - 1);
    return Cons(a, n, l);
  }
}

...

void init_gc(void) {
  SetDescriptor((void*)itrev,"_ll_ll");
  SetDescriptor((void*)mklst,"__l");

...

}
```

Figure 2.6: Interfacing with the collector.

```
void gc(arena *from) {
  frame roots = CallerFrame();
  arena to = arena_new();         /* initial */
  update_roots(to,roots);         /* update roots */
  arena_free(*from); *from = to; /* flip */
}
```

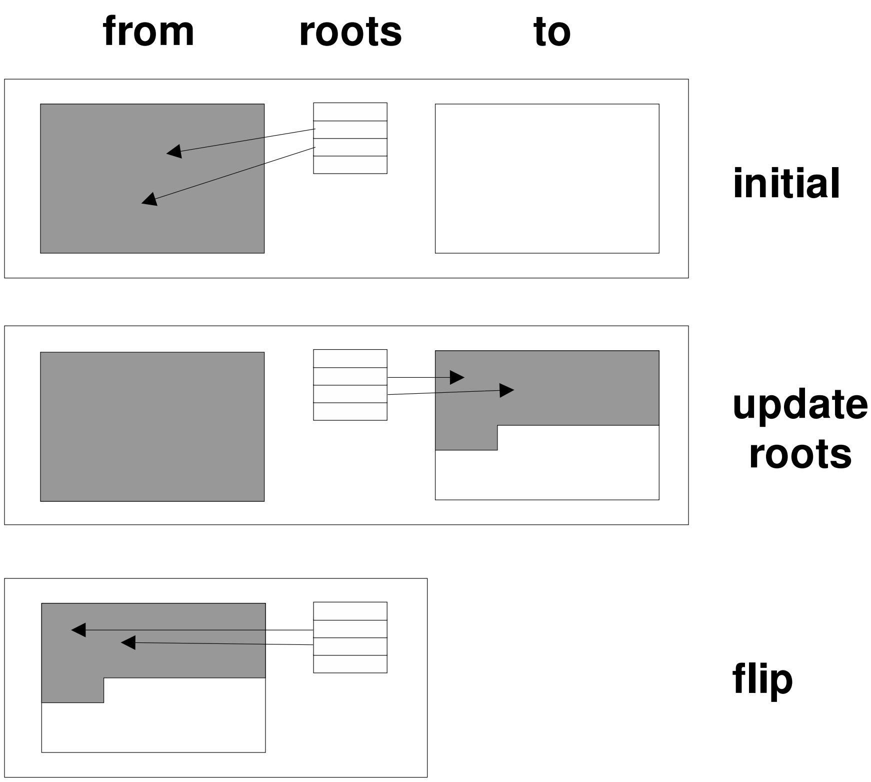


Figure 2.7: A simple tagless collector.

```
void update_roots(arena to,frame f) {
  while(f != NULL) {
    char *fdesc = GetDescriptor(f);
    if(fdesc != NULL) {
      int i = 0;
      while(fdesc[i] != '\0') {
        switch(fdesc[i]) {
        case '_': break;   /* skip */
        case 'l': {
          lst *l = FindVar(f,i);
          if(l != NULL)   /* is variable live */
            *l = copy_lst(to,*l); }
        break;
        default: abort(); /* error unknown type code */
        }
        i++;              /* do next variable */
      }
    } else abort();       /* error no frame descriptor */
    f = NextFrame(f);     /* do the next frame */
  }
}

lst copy_lst(arena a, lst l) {
  if(l == Nil) return Nil;
  else {
    int x = l->hd; lst xs = l->tl;
    lst new_l = copy_lst(a,xs);
    return Cons(a,x,new_l);
  }
}
```

Figure 2.8: Functions to update roots.

to implement.

Figure 2.6 shows how to extend our function to use the interface we have described. At each safe point we insert a test that calls the function `need_gc` on the current allocation arena. If the collector needs to be invoked, we pass the garbage collector (`gc`) the address of the current allocation arena (`a`) so that the collector can manipulate the arena. We also introduce a new function `init_gc` which is called before our program executes to propagate the information about stack frame layout to the collector. For example the string `"_ll_ll"` describes the stack frame layout for the function `itrev`. The first variable in the stack frame is the argument variable `a`, which the collector should not traverse so we use the type "`_` ". The next variable on the stack frame is the argument variable `l`, which the collector should traverse so we use the type "`l`".

A simple tagless collector. Figure 2.7 provides an implementation of the `gc` function that calls an auxiliary function `update_roots`. Our collector is a copying collector. It first creates a fresh allocation arena `to`. The `update_roots` functions walks the stack frames passed to it and updates each live value to point to a fresh copy of that value in the `to` arena. Afterwards, all the data in the `from` arena is no longer needed, so we can deallocate the arena. Finally we arrange for the `to` arena to be the new allocation arena. Figure 2.8 contains the code to properly update the roots to point to fresh copies in the `to` arena. It calls an auxiliary function `copy_lst` when it encounters a variable of the type `lst`.

Complications with optimizer. Notice that `update_roots` modifies variables of other functions saved on the stack frame. So when we return from the garbage collector, variables are modified in a way that is not locally obvious. For example in the version of `itrev` in Figure 2.7 the argument variable `l` is never modified in the body of the function. It may be potentially modified by a call to the collector. However in the function `mklst` the argument `n` is invariant because our collector modifies only variables of type `lst`. Any optimizing compiler must therefore be aware of the special semantics of our garbage collector function and assume that `l` is not invariant across the safe point while `n` is invariant across the safe point. Ramsey et al. [JRR99] provide explicit hints to an optimizing compiler about what variables are and are not invariant across a safe point. These hints are not checked, so even though a variable is marked invariant there is no guarantee that it actually is.

Dealing with shared and cyclic values. Our simple collector does not properly handle lists with shared tails. Rather than create dangling references, our simple `copy_lst` function duplicates shared tails. If `copy_lst` is handed a cyclic list it will loop. To solve this problem, we must keep track of when we have already copied a list and avoid duplicating work. One way to do this is to simply remember pointer values in a hash-table and consult the table before we copy a cell. This, however, could be inefficient. A more traditional approach is to use *forwarding pointers*.

While copying an object a traditional collector places a forwarding pointer to the copy so that if the object is seen again the collector can avoid recopying the object. In systems that tag all their data, one can sometimes overwrite the tag or first field of every object with a forwarding pointer, so there is no need to use any extra space. However, sometimes an extra word must be reserved to hold a forwarding pointer. For our tagless example it is easier to modify our representation to include an extra field to hold the forwarding pointer.

Figure 2.9 shows the changes need to handle the shared and cyclic lists in their full generality. Notice that the forwarding pointer field `fwdp` is initially set to `Nil`.

```
typedef struct lst_s *lst;
struct lst_s { lst fwdp; int hd; lst tl; };

...

lst copy_lst(arena a, lst l) {
  if(l == Nil) return Nil;
  else if(l->fwdp != Nil) return l->fwdp;
  else {
    int x = l->hd; lst xs = l->tl;
    lst new_l = Cons(a,x,Nil);  /* allocate new cell */
    l->fwdp = new_l;            /* set forwarding pointer */
    new_l->tl = copy_lst(a,xs); /* update forwarded cell */
    return new_l;
  }
}
```

Figure 2.9: Handling shared and cyclic lists.

When it is non-nil we assume the value is a forwarded pointer. Our `copy_lst` function checks for a non-nil forwarding pointer before copying an object. To handle cyclic lists properly, it must allocate a new list cell first, set the forwarding pointer to the newly allocated cell, and then copy and update the tail of the current cell. Notice that if the mutator sets the forwarding pointer of an object it can quite easily confuse the collector.

Collector using tags. Our tagless collector needs type information about the variables in the stack frame. In this case it simply needs to identify all list objects, and then use `copy_lst` to move the lists into the new allocation arena. If we generalize our collector to handle more programs, we will need a specialized copy function for every different type. Therefore, we cannot implement a tagless garbage collector without knowing in advance all the types of our program. For some languages such as Lisp which are uni-typed this is not a problem, since every Lisp program only contains one

```
typedef struct obj_s *obj;
typedef union { int n; obj p; } obj_or_int;
typedef struct desc_s *desc;
struct obj_s  { desc d; obj_or_int f[1]; }; /* variable sized object */
struct desc_s { int len; char *tmap; };

obj mkObj(arena a,desc d) {
  int sz = sizeof(struct obj_s) + (d->len - 1) * sizeof(obj_or_int);
  obj x = arena_alloc(a,sz);
  x->d = d;
  return x;
}
```

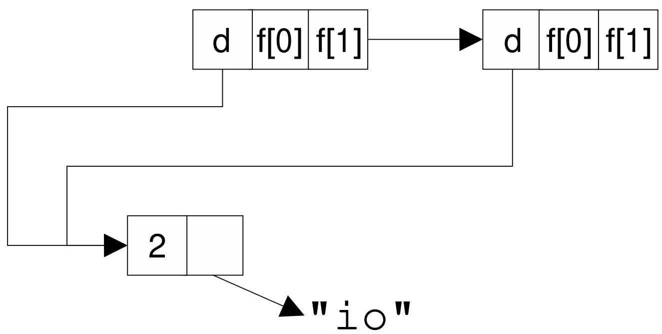
Figure 2.10: Generic tagged object representation.

type. Languages may have several different types, we must be willing to generate a garbage collector specialized for each program or resort to other techniques such as tagging.

Tag-based techniques are conceptually the same as building a tagless collector for a uni-typed language. First we must define a universal representation for all objects. However, for systems like Lisp the universal representation is typically specialized for a small set of different values used by Lisp. Our universal representation must be generic enough to accommodate a wider set of values.

Figure 2.10 describes such a generic representation and the modified `lst` type that uses the generic representation. The `obj` type is a pointer to a record whose first element is an object descriptor followed by some unknown number of fields. Each field can either be an integer or the address of another object. The type of each field and the number of fields is determined by the object descriptor (`d`), which is a record containing a length field followed by a string which describes the type of each field.

Figure 2.11 shows how a two-element list value is now represented with our new generic type, as well as the implementation of the new constructor functions. The




```
typedef obj lst;
struct desc_s lst_desc = { 2 , "io" };
const lst Nil = NULL;
lst Cons(arena a, int hd,lst tl) {
  obj t = mkObj(a,&lst_desc);
  t->f[0].n = hd; t->f[1].p = tl;
  return t;
}
```

Figure 2.11: Tagged list representation.

```
lst copy_obj(arena a, obj x) {
  if(x == NULL) return NULL;
  else {
    int i;
    int len = x->d->len;
    char *tmap = x->d->tmap;
    obj new_x = mkObj(a,x->d);
    for(i = 0; i < len; i++) {
      switch(tmap[i]) {
      case 'i':
        new_x->f[i].n = x->f[i].n;
        break;
      case 'o':
        new_x->f[i].p = copy_obj(a,x->f[i].p);
        break;
      default: abort(); /* error unknown type code */
      }
    }
    return new_x;
  }
}
```

Figure 2.12: Functions to copy tagged object representation.

descriptor for the list (`lst_desc`) describes lists objects as being objects with two fields: the first is an integer, the second an object. Notice that all our list objects share a common type descriptor. The type descriptor string (`tmap`) describes the type of each field. The character `'i'` signifies that the fields holds an integer value, while the character `'o'` signifies that a field holds a pointer, which maybe `NULL` or a pointer to another object.

Figure 2.12 contains a simple minded function that copies non-shared values of our generic representation. We can extend our function to handle cyclic shared values, by the use of forwarding pointers. However, unlike the untagged case, we do not need to change our object representation. We simply reserve a special type descriptor that represents a forwarded object, and use the first field of every object to hold the forwarding pointer.

What can go wrong? We have now seen a relatively simple but complete garbage collector. We can see that even for simple collectors, the interface with the program is quite complex. The collector must have access to the stack frame and know which variables are live. In both the tagged and untagged cases the mutator must provide accurate type information so that the collector can traverse objects. The collector and mutator must agree on how the type information is encoded. The optimizer of the mutator must be aware of the special semantics of safe points. The mutator must not mutate values used to implement forwarding pointers. Any error can cause our collector to damage data so that our program fails in mysterious ways. Since our collector is invoked infrequently these sorts of errors lead to bugs that are hard to reproduce and localize. Luckily, all these details are typically hidden from the programmer and automatically maintained by the compiler, so although there is a great deal of complexity involved in garbage collection it is hidden from the programmer.

However the burden of getting these invariants right has not been eliminated; the burden has been moved to the compiler writer. It is no small feat to get all of these invariants correct. Our primary contributions are techniques that help the compiler writer be assured that the needed invariants are in fact maintained, through the use of static type checking.

Extending to polymorphic type systems. Up until now we have been dealing with monomorphic type systems. Languages like ML that include parametric polymorphism introduce more complexities [App89, AFH94, Tol94], since the type of some variables will not be known until runtime. There are three basic approaches: The simplest is to simply remove the polymorphism by code duplication. This however is not possible in languages which support polymorphic recursion as well as making separate compilation more difficult. Another approach relies on passing type information explicitly as the program runs, so that the appropriate type information is made available at runtime [Tol94, TMC+96]. Passing and constructing the type information can affect performance of the system, so type-passing approaches are not always ideal. The other basic approach is to choose a uniform tagged representation for all values that maybe polymorphic [MDCB91, Ler92, Sha97].

Saving constant factors. There are several obvious ways to improve the simple minded collector we have described above. Rather then using character strings to encode type we can use compact bit strings. The functions to copy objects are recursive and therefore need space on the stack to run. There are several approaches that allow the recursive functions to run without an auxiliary stack [SW67, Che70, SF98]. Since we only ever need two allocation arenas at any one time and they need not be of unbounded size, the implementation of our allocation arenas could be made simpler and faster by representing each arena as a single contiguous piece of memory. If our allocation arenas are contiguous, we can inspect the address of a pointer to discover which arena contains the object. Using this check, we can avoid the need for adding a special forwarding-pointer field for many types when building a tagless system. All of these optimizations are commonly used. They are simply more efficient implementations of the underlying abstractions we have so far presented.

2.5 Summary

We have illustrated several memory management techniques used in traditional systems. The first technique requires a global single pointer invaraint. Such an invariant can be enforced using a linear typing discipline that we will discuss in Chapter 7. Maintaining this single pointer invariant can lead to inefficiency because objects must be copied rather than shared. The second technique allocates and deallocates objects in groups called arenas. Arenas are simple to implement and can be used to manage the storage of objects without needing to maintain a single pointer invariant. To use arenas in a safe way requires other invariants be maintained. In Chapter 3 we will discuss a type system that enforces these invariants. The third technique we have discussed is the use of a tracing garbage collector.

Garbage collectors require many complex invariants be maintained. Collectors complicate optimizations of programs, because they violate normal abstractions by performing operations such as examining and modifying the stack frame. All of these complex invariants can be maintained automatically by a compiler, but this just hides the complexity from the programmer and moves it to the compiler writer. In Chapter 4 and Chapter 5 we will discuss how to use type systems to guarantee many important invariants needed by a garbage collector.

Chapter 3

A Simple Region Calculus

3.1 Introduction

In Chapter 2 we described arena-based memory-management techniques. In this chapter we describe a simple type system that provides for safe arena style-memory management. Our type system is based on the region type systems developed first by Tofte and Talpin [TT94]. The original motivation for the region type system was to provide a memory management technique for higher-order languages that had a simpler runtime model than traditional garbage-collection techniques. Their system [TBE+98] compiles the full Standard ML language into programs that run without a garbage collector. Memory is managed by automatically placing objects with similar lifetimes in a *region*. Regions, like arenas, contain several objects and are deallocated as a group in constant time.

Their approach was inspired by the pure stack allocation model of Algol [TH01]. The problem with a pure stack discipline is that stack-allocated values cannot safely escape the lexical scope of their definitions. This prevents the safe use of higher-order functions. In the Tofte and Talpin system values are allowed to escape the lexical scope of their definition. However, the type system associates with each value a region variable which has an associated region scope. Values in the type system are not allowed to escape the scope of the region variables. The scope of a region variable is associated with the dynamic lifetime of the memory used to hold values allocated in that region, just as the lexical scope of variables in Algol is associated with their dynamic lifetime. Their system also support polymorphism over regions which provides the language with extra flexibility not found in the Algol like languages. One initial shortcoming of the region system, which has been addressed by later work, was the LIFO region allocation policy. Like Algol the scopes and lifetimes of regions must be nested in a way that allows the deallocation of only the most recently allocated region.

Their system is based on a *region inference* algorithm, which attempts to automatically place objects in regions so as to minimize the total memory consumption of the program. Unfortunately, the algorithm is sensitive to the structure of the original program in subtle ways that requires "region conscious" [TBE+98] programming. Their region inference system is based on effect inference [TJ92], which allows one to infer strong semantic properties of program based on the type of the programs. In particular one can infer that a program after some point will never access a given region and therefore all values in that region are semantic garbage. However, these values may be reachable via some set of program roots. So the region system creates what might be considered dangling pointers in a traditional system. However, these dangling pointers are safe, since the region system guarantees they will never be dereferenced.

The original region calculus and garbage collectors that use a reachability relation to approximate liveness are incomparable in their ability to reclaim semantic garbage. There are situations where the Tofte and Talpin system will reclaim semantic garbage that would be retained by a traditional garbage collector. Likewise, their are cases where unreachable values are retained by region systems. Combining region type systems and tracing garbage collectors seems to provide the benefits of both systems [Hal99].

A simple region calculus. Since the original Tofte-Talpin calculus there have been many different formulations of various region calculi [AFL95, CWM99, BHR99, Yat99, ZG00, Cal01, CHT01]. For our purposes the region system need not be particularly advanced. We do not need to separate read and write effects, support effect polymorphism, or allow for dangling pointers. All of these features are included in the original Tofte-Talpin region calculus [TT94]. With the exception of [CWM99] most of the various region calculi are formulated as type-and-effect systems. In these calculi every expression has associated with it an inferred set of effects which represents an estimate of the set of regions an expression may access. Rather than infer a set of effects [CWM99] presents a region calculus where an expression is typed with respect to a "capability context" that explicitly restricts the set of regions a given expression may access. The primary difference between these two approaches is that the former is concerned with finding effect annotations which lead to more memory efficient programs that do not require a garbage collector. Capability systems are designed to verify that effects inferred by such systems are correct. We take the capability view of regions since we are primarily interested in validating that a particular effect assignment is valid.

Early deallocation. The original Tofte-Talpin calculus required that regions be allocated and deallocated in a strict LIFO fashion. The most recently allocated region must be the first region that is deallocated. Aiken, Fähndrich, and Levien [AFL95] present a static analysis that allows for *early deallocation*[1] which allows for more flexible non-LIFO policies. The capability calculus of [CWM99] also supports early deallocation. For our type-preserving garbage collector early deallocation is critically important. However, rather than simply using the capability calculus of [CWM99] we derive a simpler and in some respects more expressive calculus that relies on dynamic checking to handle certain aliasing issues that the capability calculus deals with in a purely static way. We will discuss the aliasing issue later when we discuss our static semantics.

3.2 Syntax

Figure 3.1 describes the abstract syntax of a capability-style region calculus. Previous region calculi explicitly annotate every type and term with a region annotation to precisely reflect a particular set of implementation details, such as the fact that most objects do not fit in machine registers and must have auxiliary storage allocated for them. Rather than "baking in" such a decision and cluttering our syntax we introduce two term-level operators and a new type constructor. One term-level operator is $\mathsf{put}[\rho](e)$ that evaluates it argument e and stores the resulting value into region ρ. If the type of the argument e is of type τ the type of the resulting value is $(\tau\ \mathsf{at}\ \rho)$. The type constructor at describes objects of some type τ allocated in region ρ. The operator $\mathsf{get}[\rho](e')$ expects that its argument e' evaluates to a value of type $(\tau\ \mathsf{at}\ \rho)$. It then fetches the value from region ρ and returns a value of type τ. The terms $\mathsf{put}[\rho](e)$ and $\mathsf{get}[\rho](e)$ are inspired by the translation of Banerjee et al. [BHR99], which translates a simplified effect based region calculus into a novel typed lambda calculus. In the translation the effects of the original region calculus are encoded as

[1]Early in the sense of sooner than what a strict LIFO policy would allow.

types	τ	$::=$	unit	
		$\mid$	$\tau_1 \xrightarrow{\Delta} \tau_2$	function type with effect
		$\mid$	$\forall\rho.\tau$	region polymorphic type
		$\mid$	$(\tau\ \mathsf{at}\ \rho)$	region annotated type
		$\mid$	Ans	
terms	e	$::=$	x	
		$\mid$	$\langle\rangle$	
		$\mid$	$(\lambda x\!:\!\tau.e)^{\Delta}$	effect annotated function
		$\mid$	$(e_1\ e_2)$	
		$\mid$	$(\Lambda\rho.e)$	region abstraction
		$\mid$	$(e[\rho])$	region application
		$\mid$	$\mathsf{letr}\ \rho\ \mathsf{in}\ e$	allocate new region
		$\mid$	$\mathsf{put}[\rho](e)$	put value in region
		$\mid$	$\mathsf{get}[\rho](e)$	get value from region
		$\mid$	$\mathsf{only}\ \Delta\ \mathsf{in}\ e$	deallocate regions early
		$\mid$	$(\mathsf{fix} f\!:\!\tau.v)$	
		$\mid$	halt^{τ}	
values	v	$::=$	$\langle\rangle \mid (\lambda x\!:\!\tau.e)^{\Delta} \mid (\Lambda\rho.e) \mid \mathsf{put}[\rho](v)$	
region contexts	Δ	$::=$	$\{\} \mid \{\rho_1, \ldots, \rho_n\}$	

Figure 3.1: Abstract syntax.

term level objects in the target calculus. A value allocated in region ρ is represented by the syntactic term $\mathsf{put}[\rho](v)$. We treat region contexts (Δ) as sets of region variables.

When a function $((\lambda x : \tau . e)^{\Delta})$ captures values in its closure we must account for any regions needed to access those values by annotating the function with the set of needed region variables. Function types are annotated with a corresponding set of region variables needed to evaluate the body of a function. Since we have an explicit region abstraction term ($\Lambda\rho.e$) and a separate fixed-point term in our language we support polymorphic recursion over region variables. The term $\mathsf{only}\ \Delta\ \mathsf{in}\ e$ is our construct to support early deallocation. Our type-preserving garbage collector will contain a mix of "direct style" and CPS-converted expressions. For that reason we include the type Ans which is the return type for continuations, so that we can distinguish between continuations and normal functions.

3.3 Dynamic Semantics

Figure 3.2 describes the dynamic semantics of our calculus in the style of Felleisen and Wright [WF94]. We assume that all bound region variables are unique and that substitution alpha renames variables to preserve this property. We introduce the notion of a region stack, ranged over by R, which are a sequence of nested letr expressions ending in a hole ([ ]). The notation $R[e]$ represents a region stack with the hole of the stack replaced by e. E ranges over control contexts. The notation $E[e]$ represents a control context with the hole of the control context replaced by e. Programs consist of a series of letr bindings establishing an initial region stack surrounding an expression to evaluate. Answers are a subset of program expressions that consist of a single halt^{τ} expression or a region stack enclosing a value.

We define two one step reduction relations. The relation $\mapsto_e$ performs local re-

$$
\begin{array}{rrcl}
\textit{programs} & P & ::= & R[e] \\
\textit{answers} & A & ::= & R[v] \mid \mathsf{halt}^{\tau} \\
\textit{control contexts} & E & ::= & [\,] \\
 & & \mid & (E\ e) \mid (v\ E) \\
 & & \mid & (E[\rho]) \\
 & & \mid & \mathsf{put}[\rho](E) \mid \mathsf{get}[\rho](E) \\
\textit{region stacks} & R & ::= & [\,] \mid \mathsf{letr}\ \rho\ \mathsf{in}\ R
\end{array}
$$

Expression Reductions

$$
\begin{array}{lrcl}
\mathsf{rdsbetav} & ((\lambda x\!:\!\tau.e)^{\Delta}\ v) & \mapsto_e & e[v/x] \\
\mathsf{rdstapp} & ((\Lambda\rho.e)[\rho']) & \mapsto_e & e[\rho'/\rho] \\
\mathsf{rdsfix} & (\mathsf{fix} f\!:\!\tau.v) & \mapsto_e & v[(\mathsf{fix} f\!:\!\tau.v)/f]
\end{array}
$$

Program Reductions

$$
\begin{array}{lrcl}
\mathsf{rdspure} & R[E[e_1]] & \mapsto_P & R[E[e_2]]\ \text{ where } e_1 \mapsto_e e_2 \\
\mathsf{rdsletr} & R[E[\mathsf{letr}\ \rho\ \mathsf{in}\ e]] & \mapsto_P & R[\mathsf{letr}\ \rho\ \mathsf{in}\ E[e]] \\
\mathsf{rdsget} & R[\mathsf{letr}\ \rho\ \mathsf{in}\ R'[E[\mathsf{get}[\rho](\mathsf{put}[\rho](v))]]] & \mapsto_P & R[\mathsf{letr}\ \rho\ \mathsf{in}\ R'[E[v]]] \\
\mathsf{rdsonly} & R[E[\mathsf{only}\ \Delta\ \mathsf{in}\ e]] & \mapsto_P & R'^{\Delta}[e] \\
\mathsf{rdshalt} & R[E[\mathsf{halt}^{\tau'}]] & \mapsto_P & \mathsf{halt}^{\tau}\ \text{ where } E \neq [\,] \\
\mathsf{rdsfree} & R[\mathsf{letr}\ \rho\ \mathsf{in}\ R'[e]] & \mapsto_P & R[R'[e]]\ \text{ where } \vdash R[R'[e]]\ \text{wt}
\end{array}
$$

$$
\begin{array}{rcl}
[\,]^{\Delta} & \stackrel{def}{=} & [\,] \\
(\mathsf{letr}\ \rho\ \mathsf{in}\ R)^{\Delta,\{\rho\}} & \stackrel{def}{=} & (\mathsf{letr}\ \rho\ \mathsf{in}\ R^{\Delta,\{\rho\}})\ \text{where } \rho \notin \Delta
\end{array}
$$

Multi-step Reduction

$$
\frac{}{P \mapsto_P^* P}\ \mathsf{mstprefl}
\qquad
\frac{P_1 \mapsto_P^* P_2 \quad P_2 \mapsto_P^* P_3}{P_1 \mapsto_P^* P_3}\ \mathsf{mstptrans}
\qquad
\frac{P_1 \mapsto_P P_2}{P_1 \mapsto_P^* P_2}\ \mathsf{mstprds}
$$

Figure 3.2: Dynamic semantics.

ductions on expressions, while the relation $\mapsto_P$ performs global reductions on whole programs. The $\mapsto_e$ relation is the standard one step reduction relation for a pure call-by-value lambda calculus. This local reduction relation is used in the definition of the global program reductions by the rule rdspure. The global program reduction rule rdsletr hoists an inner letr binding to the top-level region stack. Since we assume all bound variables are unique, the region in the letr binding is guaranteed to be a fresh variable. The rdsget rule converts a value allocated in region ρ to a pure value, if the region ρ is currently bound by the enclosing region stack. The rdsonly rule throws away the surrounding control context and continues evaluating its body in a new region stack defined by the only expression. The notation R^{Δ} is the smallest region stack binding the variables in Δ.

The rdshalt rule immediately throws away the surrounding control context and region stack and reduces to an answer. We impose the extra side condition on the rdshalt rule that the surrounding control context is nontrivial in order to prevent our abstract machine from repeatedly evaluating a halt^{τ} expression. The rdsfree rule is a nondeterministic rule that removes unneeded region bindings from the region stack. A region binding is unneeded if removing the binding does not prevent rest of the program from remaining well typed. Removing a region binding can only caused a program to be untypable if the region variable previously bound by the binding occurs free in the body of the expression. So an equivalent criteria would be that the region variable is not free in the body of the binding. However, we use the typing judgment as a side condition simply to avoid formalizing the notion of free region variable.

The relation $\mapsto_P^*$ is the reflexive transitive closure of our single step program reduction relation. Notice rdsfree implicitly allows for non-LIFO allocation policy, already providing support for early deallocation, which makes the rdsonly rule seem redundant. However, if we did not include the rdsonly rule we could not immedi-

$$\textit{value environment} \quad \Gamma \quad ::= \quad \{\} \mid \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$$

Judgement	Meaning
$\vdash P$ wt	P is a well typed program.
$\Delta; \Gamma \vdash e : \tau$	e has type τ under $\Delta; \Gamma$. We always implicitly require $\Delta \vdash \Gamma$ wfenv
$\Delta \vdash \tau$ wf	τ is a well formed type with respect to Δ.
$\Delta \vdash \Gamma$ wfenv	Γ is a well formed value environment with respect to Δ.

Figure 3.3: Summary of typing judgements.

ately reclaim regions that are bound in the useless surrounding control context of an expression for which we know will not return.

3.4 Static Semantics

Figure 3.3 summarizes the main typing judgments for our static semantics. Figure 3.4 contains the inference rules for the main typing judgments while Figure 3.5 defines some auxiliary well formedness conditions. We use the notation Δ, Δ' to represent the union of two disjoint set of region variables and the notation $\Gamma, \{x : \tau\}$ to be the extension of an environment mapping the variable x to the type τ where x does not occur already bound in Γ.

The judgment $\vdash P$ wt simply asserts that the closed program P has some valid type. It holds only if the program P is well typed under an empty typing environment. The judgment $\Delta; \Gamma \vdash e : \tau$ asserts that the expression e has type τ under the value environment Γ and the region context Δ. The judgment $\Delta \vdash \tau$ wf asserts that all the free region variables in τ occur in Δ. The judgment $\Delta \vdash \Gamma$ wfenv generalizes this notion for value environments.

In Figure 3.4, to simplify our presentation of the rules for the judgment $\Delta; \Gamma \vdash e : \tau$,

$\boxed{\vdash P \text{ wt}}$

$$\frac{\{\};\{\} \vdash P : \tau}{\vdash P \text{ wt}}\ \mathsf{wte}$$

$\boxed{\Delta;\Gamma \vdash e : \tau}$

$$\frac{}{\Delta;\Gamma,\{x:\tau\} \vdash x:\tau}\ \mathsf{htvar} \qquad \frac{}{\Delta;\Gamma \vdash \langle\rangle : \mathsf{unit}}\ \mathsf{htunit}$$

$$\frac{\Delta \vdash \Gamma \text{ wfenv} \quad \Delta' \vdash \tau_1 \text{ wf} \quad \Delta';\Gamma',\{x:\tau_1\} \vdash e:\tau_2}{\Delta,\Delta';\Gamma,\Gamma' \vdash (\lambda x:\tau_1.e)^{\Delta'} : \tau_1 \xrightarrow{\Delta'} \tau_2}\ \mathsf{htabs}$$

$$\frac{\Delta,\Delta';\Gamma \vdash e_1 : \tau_1 \xrightarrow{\Delta'} \tau_2 \quad \Delta,\Delta';\Gamma \vdash e_2 : \tau_1}{\Delta,\Delta';\Gamma \vdash (e_1\ e_2) : \tau_2}\ \mathsf{htapp}$$

$$\frac{\Delta,\{\rho\};\Gamma \vdash e:\tau}{\Delta;\Gamma \vdash (\Lambda\rho.e) : \forall\rho.\tau}\ \mathsf{httabs} \qquad \frac{\Delta,\{\rho'\};\Gamma \vdash e : \forall\rho.\tau}{\Delta,\{\rho'\};\Gamma \vdash (e[\rho']) : \tau[\rho'/\rho]}\ \mathsf{httapp}$$

$$\frac{\Delta \vdash \tau \text{ wf} \quad \Delta,\{\rho\};\Gamma \vdash e:\tau}{\Delta;\Gamma \vdash (\mathsf{letr}\ \rho\ \mathsf{in}\ e) : \tau}\ \mathsf{htletr}$$

$$\frac{\Delta,\{\rho\};\Gamma \vdash e:\tau}{\Delta,\{\rho\};\Gamma \vdash \mathsf{put}[\rho](e) : (\tau\ \mathsf{at}\ \rho)}\ \mathsf{htput} \qquad \frac{\Delta,\{\rho\};\Gamma \vdash e : (\tau\ \mathsf{at}\ \rho)}{\Delta,\{\rho\};\Gamma \vdash \mathsf{get}[\rho](e) : \tau}\ \mathsf{htget}$$

$$\frac{\Delta \vdash \Gamma \text{ wfenv} \qquad \Delta';\Gamma' \vdash e : \mathsf{Ans}}{\Delta,\Delta';\Gamma,\Gamma' \vdash (\mathsf{only}\ \Delta'\ \mathsf{in}\ e) : \mathsf{Ans}}\ \mathsf{htonly}$$

$$\frac{\Delta \vdash \tau \text{ wf} \quad \Delta;\Gamma,\{f:\tau\} \vdash v:\tau}{\Delta;\Gamma \vdash (\mathsf{fix} f:\tau.v) : \tau}\ \mathsf{htfix} \qquad \frac{\Delta \vdash \tau \text{ wf}}{\Delta;\Gamma \vdash \mathsf{halt}^{\tau} : \tau}\ \mathsf{hthalt}$$

Figure 3.4: Main Typing Judgement

$\boxed{\Delta \vdash \tau \text{ wf}}$

$$\frac{}{\Delta \vdash \mathsf{unit} \text{ wf}} \ \mathsf{wfunit}$$

$$\frac{\Delta \vdash \tau_1 \text{ wf} \quad \Delta \vdash \tau_2 \text{ wf}}{\Delta \vdash \tau_1 \xrightarrow{\Delta} \tau_2 \text{ wf}} \ \mathsf{wfarrow} \qquad \frac{\Delta, \{\rho\} \vdash \tau \text{ wf}}{\Delta \vdash \forall \rho . \tau \text{ wf}} \ \mathsf{wfall}$$

$$\frac{\Delta, \{\rho\} \vdash \tau \text{ wf}}{\Delta, \{\rho\} \vdash (\tau \ \mathsf{at} \ \rho) \text{ wf}} \ \mathsf{wfat}$$

$$\frac{}{\Delta \vdash \mathsf{Ans} \text{ wf}} \ \mathsf{wfAns}$$

$\boxed{\Delta \vdash \Gamma \text{ wfenv}}$

$$\frac{}{\Delta \vdash \{\} \text{ wfenv}} \ \mathsf{wfenvempty} \qquad \frac{\Delta \vdash \Gamma \text{ wfenv} \quad \Delta \vdash \tau \text{ wf}}{\Delta \vdash \Gamma, \{x : \tau\} \text{ wfenv}} \ \mathsf{wfenvbv}$$

Figure 3.5: Auxiliary typing judgements.

we implicitly assume that region variables free in Γ occur in the region context Δ. If we just assume that region variables in Γ occur in Δ for the rules htvar and htunit a simple inductive argument over the derivation of typing rules as presented will show that if $\Delta; \Gamma \vdash e : \tau$ then $\Delta \vdash \Gamma$ wfenv. The only interesting cases are for the rules such as htonly and htabs where we explicitly require the inductive hypothesis ($\Delta \vdash \Gamma$ wfenv) to hold.

The typing rules in Figure 3.4 closely resemble the typing rules for a polymorphic simply-typed lambda calculi, where type polymorphism has been replaced with region polymorphism. The rules htabs differs for the standard rule in that we check the body of the function in a value environment that contains a subset of the enclosing value environment. We check the body in a value environment consisting of those bindings whose free region variables are mentioned in the effect annotation for the function. A similar restriction is placed on the htonly rule. A simple inductive argument will establish the fact that any closed program of type Ans which evaluates to an answer must evaluate to the answer $\mathsf{halt}^{\mathsf{Ans}}$. So informally speaking the requirement that the body of the $\mathsf{only}\ \Delta\ \mathsf{in}\ e$ expression be of type Ans means the body does not return. Our dynamic semantics takes advantage of this fact by throwing away the unneeded control context and region stack.

Because the typing rules htonly and htabs check subexpressions in a nonstandard way the standard substitution lemma for terms does not hold. A more restrictive substitution for values and fix expressions does hold. Since our calculus is call-by-value the more restrictive lemma is sufficient to prove type soundness for our calculus.

3.5 Early Deallocation

A realistic implementation of our abstract semantics must adopt adopt a "region passing" semantics, where region variables are bound to dynamic values at runtime, and therefore region abstractions can not be erased. The only expression takes an arbitrary set of region variables. At runtime we simply note what regions are dynamically bound to the region variables passed to the only expression and safely deallocate any other regions, since they are not needed to evaluate the rest of the program. This dynamic approach to early deallocation of regions is a novel approach, which is simpler than current static approaches to early deallocation and more expressive. The cost of the deallocation operation is at worst linearly related to the number of live regions. Region deallocation primitives in other system are constant time operations. So our more flexible dynamic approach is not without its cost. But for our type-preserving garbage collector this extra cost is negligible, since we can bound the number of live regions to a small constant.

Consider the function

```
fun f[ρa, ρb](x:int at ρb):Ans =
 free_region ρa in (get[ρb](x) ; halt())
```

which uses a new free_region operator that deallocates ρ_a before evaluating its body. At first glance it would seem that region ρ_a is not used in the body of `f` so this early deallocation of ρ_a is safe. However, consider the following calling context for `f`

```
letr ρ1, ρ2 in
 if e then f[ρ1, ρ2](put[ρ1](1))
 else f[ρ1, ρ1](put[ρ1](1))
```

The expression put$[\rho](1)$ stores the integer into the region ρ_1 and returns a reference to the integer. Notice that if we execute the first branch of the conditional then f behaves as expected. However, if we execute the second branch then at runtime the region variables ρ_a and ρ_b are both bound to the same region and the program will attempt to access a region which we have erroneously deallocated. To handle this situation correctly we can use various typing disciplines to prevent programs from deallocating region variables which may be aliased[CWM99]. The static approaches do not incur any runtime overhead, but are relatively complex systems and would disallow us from writing the program above.

Using our dynamic approach we write f as

```
fun f[ρa, ρb](x:int at ρb):Ans =
  only ρb in (get[ρb](x) ; halt())
```

At runtime we determine what region is actually bound to ρ_b, and deallocate the region bound to ρ_a if it is distinct from ρ_b. If ρ_a and ρ_b are bound to distinct regions then we know that it is safe to deallocate the region associated with ρ_a since we do not need it to evaluate the rest of the computation. It is not hard to implement such a system in practice. In our prototype system all of the region primitives are less than 200 lines of C. We simply reserve a “live-bit” for each region and mark all regions bound in the context of the only expression as live. All other unmarked regions can be reclaimed. This particular approach takes time proportional to the number of total allocated regions in order to reclaim all the live regions. An alternative implementation that uses doubly-linked lists of free and allocated regions can be implemented whose runtime is proportional to the number of live regions.

This dynamic approach to region deallocation is similar to the work of Aiken and Gay [GA98]. However, they use a relatively weak region type system and a more expensive reference counting approach that requires updating a reference count for each interregion store. Because our type system provides more guarantees we can safely deallocate regions without needing to maintain any reference counts.

3.6 Safety Properties

The safety properties of our language are standard. We include the full proofs of all the theorems and lemmas in Appendix A. Here we only describe the main theorem and associated lemmas. We begin with the definition of stuck programs

Definition 1 (Stuck Program) *The evaluation of a program P_1 is* **stuck** *if P_1 is not an answer and there is no P_2 such that $P_1 \mapsto_P P_2$.*

The main safety theorem is as follows

Theorem 1 (Type Soundness) *If $\vdash P_1$ wt, then there is no stuck P_2 such that $P_1 \mapsto_P^* P_2$.*

The proof of our main theorem follows directly with a slightly stronger induction hypothesis over the $\mapsto_P^*$ relation and the application of Lemma 1.1 and Lemma 1.2.

Lemma 1.1 (Type Preservation of Programs) *If $\vdash P_1$ wt and $P_1 \mapsto_P P_2$, then $\vdash P_2$ wt.*

Lemma 1.2 (Progress) *If $\vdash P_1$ wt, then there exists P_2 such that $P_1 \mapsto_P P_2$ or P_1 is an answer, i.e. P_1 is not stuck.*

We do not have a general substitution lemma for terms but a restricted form which only holds for values and fix expressions

Lemma 1.3 (Typing Under Term Substitution) *If $\Delta; \Gamma \vdash e : \tau$ and $\Delta; \Gamma, \{x : \tau\} \vdash e' : \tau'$ then $\Delta; \Gamma \vdash e'[e/x] : \tau'$, where $e = v$ or $e = (\mathsf{fix} f : \tau''.v)$.*

The following lemma allows us to throw away unneeded bindings and region variables from our value environment and region context by inspecting the type of values and fix expressions it is need in our proof of the substitution lemma for the htabs case.

Lemma 1.4 (Region Context Strengthening) *If* $\Delta' \vdash \tau$ wf, $\Delta \vdash \Gamma$ wfenv, $\Delta' \vdash \Gamma'$ wfenv, *and* $\Delta, \Delta'; \Gamma, \Gamma' \vdash e\!:\!\tau$ *where* $e = v$ *or* $e = (\mathsf{fix} f\!:\!\tau.v)$ *then* $\Delta'; \Gamma' \vdash e\!:\!\tau$

The remaining lemmas are used in the proof of the previous lemmas and are included for completeness but are not technically interesting

Lemma 1.5 (Type Preservation of Expression) *If* $\Delta; \{\} \vdash e_1 : \tau$ *and* $e_1 \mapsto_e e_2$, *then* $\Delta; \{\} \vdash e_2\!:\!\tau$.

Lemma 1.6 (Redex Decomposition) *If* $\Delta; \{\} \vdash e\!:\!\tau$ *then* e *is a value or* $e = E[r]$ *where* r *is a redux. A redux is any of the following forms:*

1. $((\lambda x\!:\!\tau'.e')^{\Delta''}\ v)$ *where* $\Delta = \Delta', \Delta''$
2. $((\Lambda\rho.e')[\tau'])$
3. $(\mathsf{fix} f\!:\!\tau'.v)$
4. $\mathsf{letr}\ \rho\ \mathsf{in}\ e'$
5. $\mathsf{get}[\rho](\mathsf{put}[\rho](e'))$
6. $\mathsf{only}\ \Delta'\ \mathsf{in}\ e'$
7. $\mathsf{halt}^{\tau'}$

Lemma 1.7 (Canonical Forms) *If* $\Delta; \Gamma \vdash v : \tau$ *then one of the following must be true.*

1. $\tau = \mathsf{unit}$ *iff* $v = \langle\rangle$
2. $\tau = \tau_1 \xrightarrow{\Delta''} \tau_2$ *iff* $v = (\lambda x\!:\!\tau_1.e)^{\Delta''}$ *and* $\Delta = \Delta', \Delta''$
3. $\tau = \forall\rho.\tau'$ *iff* $v = (\Lambda\rho.e)$

4. $\tau = (\tau' \ \mathsf{at}\ \rho)$ *iff* $v = \mathsf{put}[\rho](v')$ *and* $\Delta = \Delta', \{\rho\}$

Lemma 1.8 (Typing Relation Implies Well Formedness) *If* $\Delta; \Gamma \vdash e : \tau$ *then* $\Delta \vdash \tau$ wf.

Lemma 1.9 (Typing Under Region Variable Substitution) *If* $\Delta, \{\rho'\}, \{\rho\}; \Gamma \vdash e : \tau$ *then* $\Delta, \{\rho'\}; \Gamma[\rho'/\rho] \vdash e[\rho'/\rho] : \tau[\rho'/\rho]$.

Lemma 1.10 (Control Context Independence) *If* $\Delta; \Gamma \vdash E[e] : \tau$ *then* $\Delta; \Gamma \vdash e : \tau'$.

Lemma 1.11 (Control Context Replacement) *If* $\Delta; \Gamma \vdash e_1 : \tau$, $\Delta; \Gamma \vdash e_2 : \tau$, *and* $\Delta; \Gamma \vdash E[e_1] : \tau'$ *then* $\Delta; \Gamma \vdash E[e_2] : \tau'$.

Lemma 1.12 (Region Stack Independence) *If* $\Delta; \Gamma \vdash R[e] : \tau$ *then there exists* Δ' *such that* $\Delta, \Delta'; \Gamma \vdash e : \tau$.

Lemma 1.13 (Region Stack Replacement) *There exists* Δ' *such that if* $\Delta, \Delta'; \Gamma \vdash e_1 : \tau$, $\Delta, \Delta'; \Gamma \vdash e_2 : \tau$, *and* $\Delta; \Gamma \vdash R[e_1] : \tau$ *then* $\Delta; \Gamma \vdash R[e_2] : \tau$.

3.7 Summary

We have presented a type system that allows for reasoning about arena style memory management in a safe way. Our calculus is designed to check that a particular assignment of values to regions is safe, unlike the original region type systems which were geared toward region inference. Our system allows for early deallocation, which is a critical feature needed by a garbage collector. We also believe the calculus itself has applications beyond type-preserving garbage collection. We have established the type soundness of our core calculus using standard syntactic techniques.

Tofte and Talpin established the soundness of their calculus through a rule-based co-induction argument [TT94]. Recently there have been several different reformulation that use different proof techniques to establish soundness [CWM99, BHR99, ZG00, CHT01, Cal01]. Our calculus is inspired by the work of Banerjee, Heintze, and Riecke [BHR99]. However, rather than using a denotational model to establish soundness, we use the simpler syntactic techniques seen in other calculi [CWM99, CHT01, Cal01]. Our small-step operational semantics resembles the semantics of the *store-less region calculus* [CHT01] presented by Calcagno, Helsen and Thiemann. However, we include an explicit set of valid regions during reduction to model illegal accesses to dead values in a way similar to the big step semantics presented by Calcagno et al. [Cal01]. Also our deallocation semantics differs significantly from both approaches in that we do not allow for dangling pointers.

Our semantics allows for the early deallocation of regions, a feature found in only two other region calculi [AFL95, CWM99]. Unlike Aiken et al. [AFL95] we guarantee safety using a simple type soundness argument, rather the proving soundness of an auxiliary static analysis which results in the safe insertion of explicit deallocation operators into the original Tofte-Talpin region calculus. Unlike Crary et al. [CWM99] we deal with region aliasing by using runtime checking. Also programs in our calculus can be a mix of CPS and direct-style terms. No region calculus we know supports both direct-style and CPS converted programs efficiently.

Chapter 4

A Type-Preserving Garbage Collector

4.1 Introduction

In Chapter 3 we introduced a simple region calculus and established the soundness of the language. Using a variant of the region calculus presented in Chapter 3, we will show how to construct a garbage collector for a simple list reverse program written in an ML-like language. We will be able to typecheck the resulting program and garbage collector, and no unsafe primitives will be needed in the implementation. Along the way, we will compare our garbage collector with the toy collector we presented in Chapter 2. An important property of our garbage collector is that it is guaranteed to preserve the types of the program being garbage collected.

A functional programing view of garbage collectors. Figure 4.1 illustrates a simple two-space stop-and-copy collector. When the collector is invoked it is passed three variables `from`, `k`, and `roots`, which are the current allocation space, the current

```
fun gc(from, k, roots) =
 if(need_gc(from)) then
  let to = new_space() in
  let roots' = copy(from, to, roots)
  in free_space(from) ; k(to, roots')
 else k(from, roots)
```

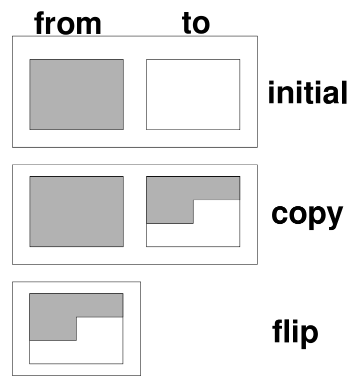


Figure 4.1: Traditional garbage collector.

continuation, and the set of live roots respectively. Heap values are allocated in the current allocation space. The current continuation represents the "rest of the program" and takes as arguments an allocation space and the live roots which point to all the currently reachable data the program may wish to use. All the data reachable from the live roots is allocated in the current allocation space.

The collector uses some heuristic to determine whether a garbage collection should occur. If so, the collector creates a fresh allocation space (`to`) then makes a deep copy of the live roots into the *to-space*. All the data reachable from the new roots (`roots'`) should live in the to-space. The collector can now safely free the old *from-space* and resume the program with the new allocation space and new live roots. Traditionally this operation is called a "flip" because once the from-space is deallocated its storage can immediately be reused as the next to-space, so the roles of the from-space and to-space are reversed. The algorithm we have sketched above is a functional view of the same algorithm presented in Chapter 2. Rather than mutating the roots in place and mutating the allocation arena, our program simply creates fresh copies. In a linear typing framework, these copies can be safely implemented as destructive updates. The continuation `k` represents the return address of the calling function and the variable `roots` represents the call stack.

In order to guarantee that the from-space can be safely deallocated, we must be certain that "the rest of the program" never accesses values allocated in the from-space. If our program is written in continuation-passing style, we can easily enforce this invariant by using the type system we described in Chapter 3. We can assign the current continuation (`k`) a type that guarantees it does not access values in the from-space.

A first-order ML-like language with regions. Our calculus in Chapter 3 can easily be extended to include disjoint unions, records, and recursive types. After these extensions our calculus is sufficiently powerful for implementing a type-preserving garbage collector. The original calculus is higher-order but as we will see the higher-order features of our calculus are not strictly needed in order to implement a type-preserving garbage collector, and make writing a collector more difficult since the garbage collector must copy values captured by the closure of a higher-order function. However, the safety of our higher-order calculus immediately implies the safety of a first-order variant, and the proof techniques for the higher-order calculus are simpler because there are fewer syntactic categories. The higher-order calculus also has applications beyond our type-preserving garbage collector. However, to simplify our presentation, we will use a first-order variant extended into a full ML like language and use more traditional ML syntax in the remainder of our discussions.

Figure 4.2 contains a program that reverses a list of integers. The function `itrev` takes two arguments `l` and `acc` both of type `lst` and returns a value of type `lst`. The argument `l` holds the list to be reversed while `acc` holds the intermediate results. The recursive call to `itrev` is a tail call, so we do not need to allocate a new stack frame for this call. Note when the program first calls `itrev` the call is not a tail call, so we must allocate a trivial stack frame for this call. As the function recursively

```
type lst = Nil | Cons (int, lst)

fun itrev(l:lst, acc:lst):lst =
 case l of Nil ⇒ acc
  | Cons(hd,tl) ⇒
    let acc' = Cons(hd, acc)
    in itrev(tl, acc')

let l = Cons(1, Cons(2, ... )) in
let rl = itrev(l, Nil) (* non-tail call *)
in rl
```

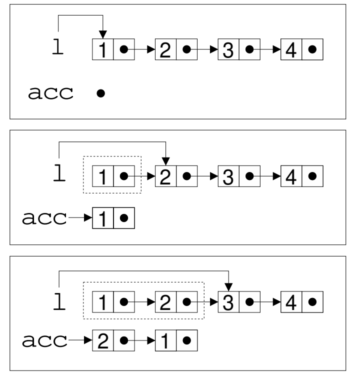


Figure 4.2: Iterative List Reverse

descends l the previous list cells, contained in the dotted box, become unreachable and can be reclaimed. The function therefore, need only retain a constant amount of live data in addition to the list itself.

Region inference would not allow us to immediately free each list cell in l after we have traversed it. A region system would force us to hold onto all the cells of l until the function returns acc. Type systems based on linear logic may give us more fine-grain control over allocation and deallocation and allow us to capture our reasoning for this particular instance, but they are fragile in the presence of aliasing [Bak92a, Bak93, SWM00, WM00].

We will convert the program in Figure 4.2 into an equivalent program that includes a function to garbage-collect dead values. The resulting program will remain well typed. We will need to perform CPS and closure conversion to the program, to make our informal reasoning about the stack and live values explicit. Afterwards, we will perform a simple region annotation to the resulting program to make precise what values live on the heap and when they are allocated. Finally, with this CPS-converted, closure-converted, region-explicit program we can synthesize a function that acts as a garbage collector for the program.

```
type lst = Nil | Cons(int, lst)
fun itrev(k:lst → Ans, l:lst, acc:lst):Ans =
 case l of Nil ⇒ k(acc) (* return *)
  | Cons(hd, tl) ⇒
    let acc' = Cons(hd, acc)
    in itrev(k, tl, acc')
and ret_l(v:lst):Ans =  (* return continuation *)
 let rl = v (* bind return value rl *)
 in rl ; halt() (* exit program *)

let l = Cons(1, ...) in
let k = ret_l
in itrev(k, l, Nil)
```

Figure 4.3: CPS-converted program.

4.2 CPS and Closure Conversion

If we CPS convert [Fis72, Plo75, Ste78, Kra87, App92a, Sab98] our source program reasoning about the control flow of the program becomes explicit. Figure 4.3 is the result of applying a standard CPS conversion to our source program. The `itrev` function now takes a return continuation (`k`), which it calls with the return value of the function. The single return continuation in our program is `ret_l` which accepts the result passed to and exits. Notice that all our functions now return the abstract type `Ans`.

If we were to invoke our garbage collector in the body of `itrev`, the collector would need to examine any data captured by the closure of the return continuation `k`. This analogous to the problem of having a traditional collector examine a stack frame for roots. Our garbage collector must have a description of the closure of `k` in order to copy the closure. We can solve this problem, by performing a typed first-order closure conversion algorithm as outlined by Tolmach and Oliva [TO98]. This form of closure conversion is also known as "defunctionalization" which was described first in

Higher-Order	**First-Order**

`let y = 1 in`
`let f = if` e `then (`λ`x:int.x)`
`        else (`λ`x:int.x + y)`
`in f 1`
	`type clos = C1 | C2(int)`
	`fun apply (f, x) =`
	` case f of C1 ⇒ x`
	`  | C2(y) ⇒ x + y`
	`let y = 1 in`
	`let f = if` e `then C1`
	`        else C2(y)`
	`in apply (f, 1)`

Figure 4.4: First-order closure conversion.

the untyped setting by Reynolds [Rey72].

Figure 4.4 illustrates Tolmach's closure conversion technique, in which the layout of every closure is made explicit. We also encode the result of a very simple control-flow analysis [Shi91, Tol94] in the process of using Tolmach's technique. The control-flow analysis is however not completely free, as we need the whole program at this point. This makes separate compilation difficult. However, with the addition of extensible sums, we can perform closure conversion locally for each compilation unit and merge units with little overhead [TO98].

Figure 4.5 contains our CPS-converted and closure-converted program along with the control-flow graph we obtain from our control-flow analysis. In general each call site of `itrev` will require one new data constructor to represent each distinct return continuation.

Tagless garbage collection algorithms examine the return address of a function stored in the stack frame in order to determine the layout of the stack frames [DMH92]. The transformation we have performed allows us to perform a similar operation. The tag of each data-constructor acts as the return address, the type of the data-constructor describes the stack layout, which is empty in this case. So we can replace

```
type lst = Nil | Cons(int, lst)
type cont = Ret_rl

fun itrev(k:cont, l:lst, acc:lst):Ans = (* B *)
 case l of Nil ⇒ apply(k, acc) (* B1 *)
  | Cons(hd, tl) ⇒             (* B2 *)
    let acc' = Cons(hd, acc)
    in itrev(k, tl, acc')
and apply(k:cont, v:lst):Ans =  (* C *)
 case k of Ret_rl ⇒ (* C1 *)
 let rl = v (* bind return value rl *)
 in rl ; halt() (* exit program *)

let l = Cons(1, ...) in  (* A *)
let k = Ret_rl
in itrev(k, l, Nil)
```

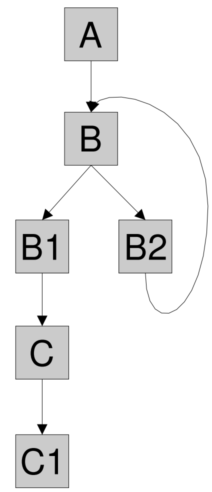


Figure 4.5: CPS-converted and closure-converted program.

low-level table of bitmaps with a set of high-level type declarations.

The chief disadvantage of first-order closure conversion is that it makes separate compilation more difficult. However, providing true separate compilation using standard higher-order techniques [MMH96, MWCG98] that preserve abstraction and have better separate compilation properties is not as simple as it may seem. Even these techniques must have a method of merging type information at link time or force all objects to be uniformly tagged, which is often undesirable. We will discuss these issues in Chapter 8.

4.3 Region Annotated

We have been informally arguing about where and when objects are allocated. Figure 4.6 shows our program with explicit region annotations. Notice that the type `lst` in Figure 4.3 becomes a type constructor `lst[`ρ`]` parameterized by a region in which the list lives. Since we can represent both the empty list and return continuation as single

```
type lst[ρ] = Nil  (* unboxed *)
            | Cons(int, lst[ρ]) at ρ (* boxed *)
type cont[ρ] = Ret_rl (* unboxed *)

fun itrev[ρ_alloc](k:cont[ρ_alloc], l:lst[ρ_alloc], acc:lst[ρ_alloc]):Ans =
 case l of Nil ⇒ apply(k, acc)
  | Cons(hd, tl) ⇒
    let acc' = lst[ρ_alloc].Cons(hd, acc)
    in itrev(k, tl, acc')
and apply[ρ_alloc](k:cont[ρ_alloc], v:lst[ρ_alloc]):Ans = ...

letr ρ_heap in (* initial program heap *)
let l = lst[ρ_heap].Cons(1, ...) in (* heap allocate list *)
let k = cont[ρ_heap].Ret_rl  (* create return continuation *)
in itrev[ρ_heap](k, l, lst[ρ_heap].Nil)
```

Figure 4.6: Program `itrev` after region annotation.

machine words we do not need to allocate space for them. We need to allocate space only when constructing list cells with the `Cons` data-constructor; this is reflected in the type `Cons(int, lst[`ρ`]) at` ρ.

Both the `itrev` and `apply` functions each take a single region parameter (ρ_{alloc}), which corresponds to the allocation pointer in a normal untyped system. When we allocate a new list cell we use the notation `lst[`ρ_{heap}`].Cons(1,...)` which instantiates the region parameter (ρ) of the type constructor `lst` to ρ_{heap} and indicates that the new list cell will be allocated in the region ρ_{heap}. We have assigned regions to types so that values are allocated in one global region, which acts like a traditional heap. When we call `itrev` we instantiate its region parameter ρ_{alloc} to ρ_{heap}. We could apply a more refined local region analysis to avoid heap-allocating an object when the lifetime of the object is locally obvious.

If the return continuation captured some live variables we would heap-allocate the continuation. This approach simplifies the compilation of advanced control features such as exceptions and first class continuations and also simplifies reasoning about

```
type lst[ρ] = Nil | Cons(int, lst[ρ]) at ρ
type cont[ρ] = Ret_rl
type gc_cont[ρ] = Ret_itrev(cont[ρ], lst[ρ], lst[ρ]) at ρ

fun itrev[ρ_alloc](k:cont[ρ_alloc], l:lst[ρ_alloc], acc:lst[ρ_alloc]):Ans =
 if need_gc[ρ_alloc]() then  (*** safe point ***)
  let roots = gc_cont[ρ_alloc].Ret_itrev(k, l, acc)
  in gc[ρ_alloc](roots)
 else ... (* body of original itrev *)
and apply[ρ_alloc](k:cont[ρ_alloc], v:lst[ρ_alloc]):Ans = ...
and gc[ρ_from](roots:gc_cont[ρ_from]):Ans = ...
...
```

Figure 4.7: Program `itrev` with safe point inserted.

safety. However, heap-allocating return continuations could impact performance in an undesirable way. A system extended with linear types, along with a set of simple syntactic restriction would allow us to stack allocate return continuations. In Chapter 7 we will discuss such a system.

4.4 GC Safe Points

Part of the interface between a garbage collector and the compiler is a description of *safe points*. These are locations during the execution of the mutator where it is safe to invoke the garbage collector. At these safe points the compiler usually emits type information describing which values are live at the safe point. Compilers that do optimizations must also be careful not to perform certain optimizations across safe points. It is complicated to characterize precisely which optimizations are and are not allowed [DMH92]. It requires that the compiler understand the special semantics of what happens at a garbage-collection safe point.

In our framework all these issues are handled straightforwardly: since the garbage collector is just a normal function, the compiler does not need to be modified to be

```
type lst[ρ] = Nil | Cons(int, lst[ρ]) at ρ
type cont[ρ] = Ret_rl
type gc_cont[ρ] = Ret_itrev(cont[ρ], lst[ρ], lst[ρ]) at ρ

fun itrev[ρ_alloc](...):Ans = ... and apply[ρ_alloc](...):Ans = ...
and gc[ρ_from](roots:gc_cont[ρ_from]):Ans =
 letr ρ_to in
  let roots' = copy_gc_cont[ρ_from][ρ_to](roots) in
   only ρ_to in (* deallocate ρ_from *)
    case roots' of
     Ret_itrev(k, l, acc) ⇒ itrev[ρ_to](k, l, acc)
and copy_gc_cont[ρ_from, ρ_to](x:gc_cont[ρ_from]):gc_cont[ρ_to] = ...
...
```

Figure 4.8: “Flipping” from and to space.

aware of any special semantics. A garbage collector is just a function that takes some data value. Figure 4.7 shows such a safe point in our program. Depending on some heuristic the code either continues executing or packages the set of current live roots into a return continuation for the garbage collector, described by the type `gc_cont`.

With region types we are able to statically verify that the data value is actually the set of live roots for the entire program. If a buggy compiler or optimizer did not include all possible roots we would catch this error at compile time, since not including a root would result in a scoping error or a violation of the region type system. More importantly, we would be able to easily identify where the error was by examining the code statically. Debugging these sorts of problems in a traditional unsafe system is considerably more difficult, because being able to isolate a bug of this sort in a large program is a serious challenge.

4.5 A Safe Flip

Figure 4.8 contains the code for the garbage collector. It copies the roots into a new region (ρ_{to}) then it implicitly deallocates the old region (ρ_{from}) and resumes the program with the new roots and new region. The term only ρ_{to} in ... requires that the body of the expression does not return, i.e. has type Ans, and can be safely evaluated using only the region dynamically bound to ρ_{to}.

In this case it is obvious that ρ_{to} is distinct from ρ_{from} as it is a new fresh region. A static type system that tracks the uniqueness of regions such as [CWM99] would be sufficient, and our dynamic apporach is not strictly necessary in this case. However, since the extra runtime overhead is negligible, we prefer our simpler dynamic approach to the more complicated static system. We believe that we can integrate the explicit deallocation techniques that use static typing to prevent region aliasing with our implicit approach to give us the benefits of both approaches, so that we resort to this dynamic approach when we are unable statically determine aliasing relationships.

4.6 Safe Copy Function

Figure 4.9 sketches the code for a naive copy function. The type of the copy function guarantees that the function performs a deep copy. The copy function is not written in continuation-passing style so it uses a stack while traversing the list. We could write the copy function in continuation-passing style and heap-allocate all its temporary space in a third region which we could reclaim after we are done. Alternatively if we extend our type system with enough technical machinery so that we can recycle the space used by the continuations we could implement what would amount to the Deutsch-Schorr-Waite pointer reversal algorithm [SW67, Vei76, SF98, WM00].

```
type lst[ρ] = Nil | Cons(int, lst[ρ]) at ρ
type cont[ρ] = Ret_rl
type gc_cont[ρ] = Ret_itrev(cont[ρ], lst[ρ], lst[ρ]) at ρ

fun itrev[ρ_alloc](...):Ans = ... and apply[ρ_alloc](...):Ans = ...
and gc[ρ_from](...):Ans = ...
and copy_gc_cont[ρ_from, ρ_to](x:gc_cont[ρ_from]):gc_cont[ρ_to] =
 case x of Ret_itrev(k, l, acc) ⇒
  let k' = copy_cont[ρ_from, ρ_to](k) in (* walk the "stack" *)
  let l' = copy_lst[ρ_from, ρ_to](l) in
  let acc' = copy_lst[ρ_from, ρ_to](acc)
  in gc_cont[ρ_to].Ret_itrev(k', l', acc')
and copy_lst[ρ_from, ρ_to](x:lst[ρ_from):lst[ρ_to] = ...
and copy_cont[ρ_from, ρ_to](x:cont[ρ_from]):cont[ρ_to] = ...
...
```

Figure 4.9: Copying roots.

Note that the function `copy_cont` performs an operation equivalent to "walking the stack". Since we have CPS-converted and closure-converted our program, the continuation `k` represents the current stack frame. It may be the case that we can adapt the higher-order techniques [MMH96, MWCG98] to provide true abstraction and separate compilation in the presence of a garbage collector by requiring each abstract object to provide a method[1] to copy or trace the object. It is not clear what the software engineering and performance issues are for this technique so we consider it to be future work. A more serious problem with our copy function is that it does not preserve pointer sharing.

4.7 Problems with Sharing

Consider the data structure in the first half of Figure 4.10. If we were to apply our garbage collection technique with a naive copy function it would convert the originally

[1] A closure can be thought of a an object with a single "apply" method.

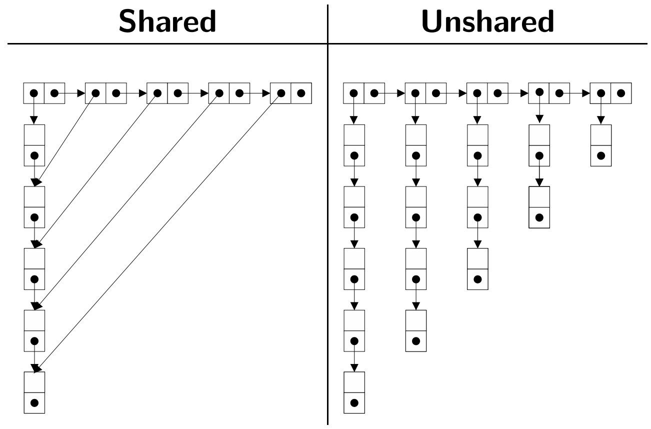


Figure 4.10: Shared vs. unshared values.

shared list of lists into an unshared version which uses more space. In the presence of cyclic data structures our naive copy function would not terminate. Traditional garbage collectors use forwarding pointers to preserve sharing. However, forwarding pointers are not the only mechanism by which to do this.

Figure 4.11 outlines a copy function that uses an auxiliary hash table augmented with one primitive to return the unique pointer address of an object. This approach, while inefficient, demonstrates that the underlying algorithm needed to preserve sharing is not inherently difficult to type. In Chapter 5 we will outline how to encode forwarding pointers in a safe way.

4.8 Summary

Although our approach as presented is not practical for general-purpose systems, we will show how practical systems can be built by extending our current outline. The most important insights are that a general-purpose collector can be built on top of a set of much simpler primitives. If standard type systems are too weak, we can rely on runtime checking or simply add "the right lemma" and encode what amounts to a small proof sublanguage to establish important preconditions needed for any ad-hoc

```
prim objId : [α] α → int
tycon tbl :: Rgn → Typ → Typ → Typ = ...
fun newTbl[ρtbl, α, β](sz:int):tbl[ρtbl, α, β] = ...
fun inTbl[ρtbl, α, β](t:tbl[ρtbl, α, β], key:α):bool = ...
fun getTbl[ρtbl, α, β](t:tbl[ρtbl, α, β], key:α):β = ...
fun addTbl[ρtbl, α, β](t:tbl[ρtbl, α, β], key:α, val:β):unit = ...

type lst[ρ] = Nil | Cons(int, lst[ρ]) at ρ
type cont[ρ] = Ret_rl
type gc_cont[ρ] = Ret_itrev(cont[ρ], lst[ρ], lst[ρ]) at ρ

fun itrev[ρalloc](...):Ans = ...
...
and share_copy_lst[ρtbl, ρfrom, ρto]
  (t:tbl[ρtbl, lst[ρfrom], lst[ρto]],x:lst[ρfrom]):lst[ρto] =
 case x of Nil ⇒ lst[ρto].Nil
 | Cons(hd, tl) ⇒
    if inTbl[ρtbl, lst[ρfrom], lst[ρto]](x) then (* is forwarded? *)
     getTbl[ρtbl, lst[ρfrom], lst[ρto]](x)
     else let hd' = hd in
      let tl' = share_copy_lst[ρtbl, ρfrom, ρto](t,tl) in
      let x' = lst[ρto].Cons(hd',tl')
      in addTbl[ρtbl, lst[ρfrom], lst[ρto]](x,x') ; (* set forwaded *)
         x'
...
```

Figure 4.11: Preserving sharing with a hash table.

reasoning that does not fit into a standard framework.

Chapter 5

Forwarding Pointers

5.1 Introduction

The easiest way to understand how to encode forwarding pointers is to start by encoding as many of the garbage-collector invariants as possible within the type system. We will discover that the type system outlined so far can capture many important invariants, but is not sufficiently expressive to capture them all precisely. However, if we examine our partial solution we will gain enough insight to come up with a full solution by extending our system with a single primitive.

Figure 5.1 sketches one approach to forwarding pointers. Some garbage collectors may overwrite a field of the object, but to simplify our presentation we assume every heap allocated object contains an extra word to hold a forwarding pointer which is either `NULL` or a pointer to an object of the appropriate type in the to-space. Notice that we have two different list types. The `gc_lst` type describes the garbage collector's view of lists. From the garbage collector's standpoint, lists are allocated in a from-space containing forwarding pointers into objects in a to-space. It must be the case that lists allocated in the to-space have forwarding pointers which are always set to

```
tycon ref :: Rgn → Typ → Typ
type gc_lst[ρ_from, ρ_to] = Nil
 | Cons(ref[ρ_from,fwd_ptr[ρ_to]], int, gc_lst[ρ_from, ρ_to]) at ρ_from
and fwd_ptr[ρ_to] = NULL | PTR(lst[ρ_to])
and lst[ρ_to] = Nil
 | Cons(ref[ρ_to, fwd_null], int, lst[ρ_to]) at ρ_to
and fwd_null = NULL
fun itrev[ρ_alloc](...):Ans = ...
...
and share_copy_lst[ρ_from, ρ_to](x:gc_lst[ρ_from, ρ_to]):lst[ρ_to] =
 case x of Nil ⇒ lst[ρ_to].Nil
  | Cons(f, hd, tl) ⇒
    (case deref[ρ_from](f) of NULL ⇒
      let hd' = hd in
      let tl' = share_copy_lst[ρ_from, ρ_to](tl) in
      let l = lst[ρ_to].Cons(mkref[ρ_to](fwd_null.NULL), hd' , tl')
      in f := l; l
     PTR(l) ⇒ l)
```

Figure 5.1: Encoding forwarding pointers.

NULL. The lst type describes lists that the mutator operates on, and maintains the invariant that the forwarding pointer is set to NULL. The fact that the forwarding pointer is a mutable field which the garbage collector will mutate is captured by the use of the ref constructor.

The function share_copy_lst takes objects of type gc_lst and makes a copy of type lst which preserves the underlying pointer sharing in the original gc_lst. This code handles only acyclic lists but can be extended to handle the cyclic case. At first glance this would seem to be a complete solution; unfortunately there is one thorny problem. If the mutator operates on objects of type lst how did we get an object of type gc_lst in the first place?

Ideally, we would like to argue that there is a natural subtyping relationship that allows us to coerce objects of type lst into objects of type gc_lst. For this to work we need the ref constructor to be covariant. However, it is well known that covariant references are unsound. However, Java adopts this rule for arrays[1] and achieves safety by requiring an extra runtime check for every array update. We cannot adopt the approach used by Java. This runtime check would prevent our garbage collector from setting any forwarding pointer to a non-null value.

However, rather than disallowing unsafe updates to an object we can disallow unsafe dereferences, more importantly we can disallow unsafe dereferences in a way that does not require a runtime check for every access. Given a value of type `lst`, if after casting it to a value of type `gc_lst` our program never accesses *any* value of type `lst` this cast is safe.

If our program is written in continuation-passing style, we can enforce this guarantee by making sure that after casting the value of type `lst` to a value of type `gc_lst` we pass the newly cast value immediately to a continuation that never accesses any value of type `lst`. One way to guarantee this condition statically is to type the continuation that receives the cast value in a typing context where the type `lst` is not bound.

Denying access to values of type `lst` after the program has performed a cast is too restrictive to be useful. However, since both the `lst` and `gc_lst` are region-annotated types, we can achieve a similar sort of guarantee and still write useful programs by revoking the right to the access the region where the type `lst` is allocated, using a similar scoping trick. We can do this because after our garbage collector casts a `lst` value to a `gc_lst` value it never needs to examine the original value as a value of type `lst`. After our garbage collector runs, the original `lst` value is garbage, so the mutator never needs to access the region where the `lst` value was allocated. However, if we deny access to the type `lst` by denying access to the region it lives in, where is the value of type `gc_lst` allocated? We solve this problem by introducing a new

[1] A ref cell can be thought of as a one-element array.

"fake" region which is equivalent for the purposes of subtyping to the region we denied access to but for all practical purposes appears to be a distinct fresh region.

To do this we must introduce a nonstandard and ad-hoc form of subtyping on references. This allows for safe covariant references by using region variables to control access to potentially unsafe pointer aliases. Given two types `A` and `B` where `A` is a subtype of `B` and a region ρ the type `ref[`ρ`, A]` is a subtype of `ref[`ρ'`, B]` (where ρ' is a new "fake" region variable) provided that the rest of the program does not access any values in region ρ. This rule is admittedly ad-hoc. Our approach is based on the observation of Crary, Walker, et al. [CWM99] that region variables act like "capabilities". We use this observation to revoke all old references to the object and allow access to the object only through references of the object's supertype. It is important to note that we still must at run time check that ρ is not aliased by any other region variable, so that the new region variable ρ' refers to a unique region. This extra alias check is needed for this approach to be completely sound, but all our alias checks would be unnecessary in the system of Crary, Walker, et al.

Next we formally describe our approach to forwarding pointers by exhibiting a "simple" language that provides safe covariant references. To make any formal safety claims we first must describe a language that has subtyping and region-allocated mutable references. This is best done in stages. We will first describe a system with a trivial subtyping relationship on boolean values, as an extension to our original region calculus. Then we will extend our language to include region-allocated mutable references. Finally, we will describe how to provide safe covariant references by the addition of one new operator. The language we describe here is quite small and impractical for use in a real system. However, the language highlights the key ideas needed to extend the system to include more nontrivial features.

Syntax

$$
\begin{array}{rcll}
\textit{types} & \tau & ::= & \ldots \mid \mathsf{bool}(b) \mid \mathsf{bool} \\
\textit{terms} & e & ::= & \ldots \mid b \mid \quad \mid (\mathsf{if}\ e_1\ e_2\ e_3) \\
\textit{values} & v & ::= & \ldots \mid b \mid \mathsf{mkbool}(b) \\
\textit{booleans} & b & ::= & \mathsf{true} \mid \mathsf{false} \\
\textit{control contexts} & E & ::= & \ldots \mid \quad \mid (\mathsf{if}\ E\ e_1\ e_2)
\end{array}
$$

Expression Reductions

$$
\begin{array}{rrcl}
\mathsf{rdsiftrue} & (\mathsf{if}\ \mathsf{mkbool}(\mathsf{true})\ e_1\ e_2) & \mapsto_e & e_1 \\
\mathsf{rdsiffalse} & (\mathsf{if}\ \mathsf{mkbool}(\mathsf{false})\ e_1\ e_2) & \mapsto_e & e_2
\end{array}
$$

Figure 5.2: Booleans with subtyping.

5.2 Subtyping on Booleans

Figure 5.2 describes the syntax and dynamic semantics needed to provide a very simple form of subtyping over boolean values. We add two new type constructors bool and $\mathsf{bool}(b)$ the intuitive subtyping relationships are

$$\mathsf{bool} \leq \mathsf{bool}(\mathsf{false}) \qquad \mathsf{bool} \leq \mathsf{bool}(\mathsf{true})$$

Figure 5.3 describes the needed extensions to our typing relations. Rather than adding a subsumption rule to our typing relation we introduce a coercion term $\mathsf{mkbool}(e)$ that promotes an expression of type $\mathsf{bool}(b)$ into an expression of type bool. Note that the values of type bool are of the form $\mathsf{mkbool}(b)$ while a value of type $\mathsf{bool}(b)$ is of the form b. The if expression is defined over values of type bool.

This coercive interpretation of subtyping can be extended to include all the standard subtyping relationships [BH98, Cra00]. Relying on explicit coercions rather than subsumption gives us control over what contexts subtyping can be used in. We will

$$\boxed{\Delta; \Gamma \vdash e : \tau}$$

$$\cdots$$

$$\frac{\Delta; \Gamma \vdash e_1 : \mathsf{bool} \quad \Delta; \Gamma \vdash e_2 : \tau \quad \Delta; \Gamma \vdash e_3 : \tau}{\Delta; \Gamma \vdash (\mathsf{if}\ e_1\ e_2\ e_3) : \tau}\ \mathsf{htif}$$

$$\boxed{\Delta \vdash \tau \text{ wf}}$$

$$\cdots$$

$$\frac{}{\Delta \vdash \mathsf{bool}\ \text{wf}}\ \mathsf{wfbool} \qquad \frac{}{\Delta \vdash \mathsf{bool}(b)\ \text{wf}}\ \mathsf{wfibool}$$

Figure 5.3: Typing judgements for booleans with subtyping.

see later that covariant references are only safe in a particular context.

5.3 Region-Allocated References

Figure 5.4 describes the extensions to our semantics for region allocated references. A $(\tau\ \mathsf{ref}\ \rho)$ describes a mutable reference to a value of type τ allocated in region ρ. Rather than introducing a new syntactic category for locations, we simply use variables bound by a new expression letl. Our semantics does not support mutual recursion, but by using letl bound variables to encode locations we avoid the need to add a type for heaps to our typing relation, which allows us to reuse many of our previous results unchanged in proving soundness for this extended calculus. The expression deref dereferences a location into a value. The expression $\mathsf{update}[\rho]\ e_1 :=$

Syntax

$$
\begin{array}{rrcll}
\textit{types} & \tau & ::= & \ldots \mid (\tau\ \mathsf{ref}\ \rho) & \text{region annotated reference} \\
\textit{terms} & e & ::= & \ldots & \\
 & & \mid & \mathsf{letl}\ x\!:\!(\tau\ \mathsf{ref}\ \rho) = e_1\ \mathsf{in}\ e_2 & \text{bind new location} \\
 & & \mid & \mathsf{deref}[\rho](e) & \text{dereference location} \\
 & & \mid & \mathsf{update}[\rho]\ e_1 := e_2; e_3 & \text{update location and continue} \\
\textit{values} & v & ::= & \ldots \mid x & \\
\textit{control contexts} & E & ::= & \ldots & \\
 & & \mid & \mathsf{letl}\ x\!:\!(\tau\ \mathsf{ref}\ \rho) = E\ \mathsf{in}\ e & \\
 & & \mid & \mathsf{deref}[\rho](E) & \\
 & & \mid & \mathsf{update}[\rho]\ E := e_1; e_2 \mid \mathsf{update}[\rho]\ v := E; e & \\
\textit{region stacks} & R & ::= & [\,] \mid \mathsf{letr}\ \rho\ \mathsf{in}\ H & \\
\textit{heaps} & H & ::= & R \mid \mathsf{letl}\ x\!:\!(\tau\ \mathsf{ref}\ \rho) = v\ \mathsf{in}\ H &
\end{array}
$$

Program Reductions

$$
\begin{array}{rcl}
[\,]^{\Delta} & \stackrel{def}{=} & [\,] \\
(\mathsf{letr}\ \rho\ \mathsf{in}\ H)^{\Delta,\{\rho\}} & \stackrel{def}{=} & (\mathsf{letr}\ \rho\ \mathsf{in}\ H^{\Delta,\{\rho\}}) \text{ where } \rho \notin \Delta \\
(\mathsf{letl}\ x\!:\!(\tau\ \mathsf{ref}\ \rho) = v\ \mathsf{in}\ H)^{\Delta,\{\rho\}} & \stackrel{def}{=} & (\mathsf{letl}\ x\!:\!(\rho\ \mathsf{ref}\ \tau) = v\ \mathsf{in}\ H^{\Delta,\{\rho\}}) \text{ where } \rho \notin \Delta
\end{array}
$$

$$
R^{\rho,H,x,\tau,v,H'} \stackrel{def}{=} R[\mathsf{letr}\ \rho\ \mathsf{in}\ H[\mathsf{letl}\ x\!:\!(\tau\ \mathsf{ref}\ \rho) = v\ \mathsf{in}\ H']]
$$

$$
\begin{array}{rrcl}
\mathsf{rdsderef} & R^{\rho,H,x,\tau,v,H'}[E[\mathsf{deref}[\rho](x)]] & \mapsto_P & R^{\rho,H,x,\tau,v,H'}[E[x]] \\
\mathsf{rdsletupd} & R^{\rho,H,x,\tau,v,H'}[E[\mathsf{update}[\rho]\ x := v'; e]] & \mapsto_P & R^{\rho,H,x,\tau,v',H'}[E[e]] \\
\mathsf{rdsletl} & R[\mathsf{letr}\ \rho\ \mathsf{in}\ H[R'[E[\mathsf{letl}\ x\!:\!(\rho\ \mathsf{ref}\ \tau) = v\ \mathsf{in}\ e]]]] & \mapsto_P & R^{\rho,H,x,\tau,v,R'}[E[e]]
\end{array}
$$

Figure 5.4: Regions and references.

$e_2; e_3$ evaluates e_1 to a location allocated in region ρ. It then updates the location with the value of e_2 and continues by evaluating e_3. Since locations are first class values we add them to the set of values also.

In our previous semantics region stacks were simply a set of nested letr bindings. To model references we extend region stacks so that between letr bindings there exists a possibly empty sequence of nested letl bindings which encodes a mapping between locations and values. We redefine the meaning of R^{Δ} appropriately. We also introduce the notation $R^{\rho,H,x,\tau,v,H'}$ which represents a region stack containing a bound region, ρ, which contains a heap, H, containing a location, x, bound to some value, v, of type τ and some arbitrary subheap H'. This definition will be useful when defining our dynamic semantics. The rule rdsderef dereferences locations. The rule rdsletupd updates and existing binding. The rule rdsletl lifts letl bindings up to the appropriate heap.

Figure 5.5 contains the straightforward typing rules for our new constructs. Notice that by using variables bound by letl to encode locations we avoid the need for a separate heap type and can type locations using the standard htvar rule.

5.4 Covariant References

Now that we have a calculus with a very basic form of subtyping and mutable references, we can introduce a new term to our language that allows for safe covariant references. Figure 5.6 describes the needed extensions. The term gclemma evaluates an expression e_1 to a value of type ($\mathsf{bool}(b)$ ref ρ_1) it then coerces that value into a value of type (bool ref ρ_2) where ρ_2 is a new freshly bound region variable. It then binds this value to a variable (x) and evaluates the body (e_2) in a restricted region environment (Δ) and new region stack (R'). The new region stack (R') is the same as

$\boxed{\Delta; \Gamma \vdash e : \tau}$

$$\cdots$$

$$\frac{\Delta \vdash (\tau_1 \textsf{ ref } \rho) \text{ wf} \quad \Delta; \Gamma \vdash e_1 : \tau_1 \quad \Delta; \Gamma, \{x : (\tau_1 \textsf{ ref } \rho)\} \vdash e_2 : \tau_2}{\Delta; \Gamma \vdash (\textsf{letl } x : (\tau_1 \textsf{ ref } \rho) = e_1 \textsf{ in } e_2) : \tau_2} \textsf{htletl}$$

$$\frac{\Delta, \{\rho\}; \Gamma \vdash e : (\tau \textsf{ ref } \rho)}{\Delta, \{\rho\}; \Gamma \vdash \textsf{deref}[\rho](e) : \tau} \textsf{htderef}$$

$$\frac{\Delta, \{\rho\}; \Gamma \vdash e_1 : (\tau_1 \textsf{ ref } \rho) \quad \Delta, \{\rho\}; \Gamma \vdash e_2 : \tau_1 \quad \Delta, \{\rho\}; \Gamma \vdash e_3 : \tau_2}{\Delta, \{\rho\}; \Gamma \vdash (\textsf{update}[\rho] \ e_1 := e_2; e_3) : \tau_2} \textsf{htletupd}$$

$\boxed{\Delta \vdash \tau \text{ wf}}$

$$\cdots$$

$$\frac{\Delta, \{\rho\} \vdash \tau \text{ wf}}{\Delta, \{\rho\} \vdash (\tau \textsf{ ref } \rho) \text{ wf}} \textsf{wfref}$$

Figure 5.5: Typing judgements for regions and references.

Syntax

$$
\begin{array}{rcll}
\textit{terms} & e & ::= & \ldots \\
 & & | & \mathsf{gclemma} \\
 & & & \quad \mathsf{assume}\ (\mathsf{bool}(b)\ \mathsf{ref}\ \rho_1)\ \mathsf{isa}\ (\mathsf{bool}\ \mathsf{ref}\ \rho_2)\ \mathsf{in} \\
 & & & \quad \mathsf{let}\ x = e_1\ \mathsf{in} \\
 & & & \quad \mathsf{only}\ \Delta\ \mathsf{in}\ e_2 \\
\textit{control contexts} & E & ::= & \ldots \\
 & & | & \mathsf{gclemma} \\
 & & & \quad \mathsf{assume}\ (\mathsf{bool}(b)\ \mathsf{ref}\ \rho_1)\ \mathsf{isa}\ (\mathsf{bool}\ \mathsf{ref}\ \rho_2)\ \mathsf{in} \\
 & & & \quad \mathsf{let}\ x = E\ \mathsf{in} \\
 & & & \quad \mathsf{only}\ \Delta\ \mathsf{in}\ e
\end{array}
$$

$$
\begin{array}{rl}
\mathsf{rdsgclemma} & R^{\rho_1,H,x',\mathsf{bool}(b),b,H'}[E[\mathsf{gclemma} \\
 & \qquad \mathsf{assume}\ (\mathsf{bool}(b)\ \mathsf{ref}\ \rho_1)\ \mathsf{isa}\ (\mathsf{bool}\ \mathsf{ref}\ \rho_2)\ \mathsf{in} \\
 & \qquad \mathsf{let}\ x = x'\ \mathsf{in} \\
 & \qquad \mathsf{only}\ \Delta\ \mathsf{in}\ e]] \longmapsto_P R'[E'[\mathsf{only}\ \Delta\ \mathsf{in}\ e[x'/x]]] \\
 & \text{where } R' = (R^{\rho_1,H,x',\mathsf{bool},\mathsf{mkbool}(b),H'})[\rho_2/\rho_1] \\
 & \quad \text{and } E' = (E)[\rho_2/\rho_1] \text{ and } \rho_1 \notin \Delta \\
\mathsf{rdsgclemmaerr} & R[E[\mathsf{gclemma} \\
 & \qquad \mathsf{assume}\ (\mathsf{bool}(b)\ \mathsf{ref}\ \rho_1)\ \mathsf{isa}\ (\mathsf{bool}\ \mathsf{ref}\ \rho_2)\ \mathsf{in} \\
 & \qquad \mathsf{let}\ x = v\ \mathsf{in} \\
 & \qquad \mathsf{only}\ \Delta, \{\rho_1\}\ \mathsf{in}\ e]] \longmapsto_P \mathsf{halt}^{\mathsf{Ans}}
\end{array}
$$

Figure 5.6: Covariant references.

$\boxed{\Delta; \Gamma \vdash e : \tau}$

...

$$\dfrac{\begin{array}{rl} \Delta_1, \{\rho_1\} & \vdash \Gamma_1 \text{ wfenv} \\ \Delta_1, \{\rho_1\}, \Delta_2; \Gamma_1, \Gamma_2 & \vdash e_1 : (\mathsf{bool}(b)\ \mathsf{ref}\ \rho_1) \\ \Delta_2, \{\rho_2\}; \Gamma_2, \{x : (\mathsf{bool\ ref}\ \rho_2)\} & \vdash e_2 : \mathsf{Ans} \end{array}}{\begin{array}{l} \Delta_1, \{\rho_1\}, \Delta_2; \Gamma_1, \Gamma_2 \vdash (\mathsf{gclemma} \\ \qquad \mathsf{assume}\ (\mathsf{bool}(b)\ \mathsf{ref}\ \rho_1)\ \mathsf{isa}\ (\mathsf{bool\ ref}\ \rho_2)\ \mathsf{in} \\ \qquad \mathsf{let}\ x = e_1\ \mathsf{in} \\ \qquad \mathsf{only}\ \Delta_2, \{\rho_2\}\ \mathsf{in}\ e_2) : \mathsf{Ans} \end{array}} \ \mathsf{htgclemma}$$

Figure 5.7: Typing judgements for covariant references.

the original region stack except we have replaced the binding of the particular boolean reference with its supertype and also globally renamed the region variable ρ_1 to ρ_2. We also globally rename the region variable ρ_1 to ρ_2 in the current control context. Both these operations are purely notational and have no real operational effect in a realistic implementation, but are needed in our abstract semantics to guarantee type preservation.

The reduction rule rdsgclemma is safe if and only if x' does not occur in the body of e. We can be assured of this fact when ρ_1 is not in Δ. If x' did occur in e and ρ_1 is not in Δ this would violate the assumption that e is well typed. Figure 5.7 describes the typing rule needed as a precondition for safety. Notice that we statically require that ρ_1 not be a member of $\Delta_2, \{\rho_2\}$, this is a necessary but not sufficient restriction to guarantee safety, because of region aliasing. We must check at runtime that aliasing has not violated this constraint. If it has we must abort our computation using the rdsgclemmaerr rule.

5.5 Summary

In order to preserve pointer sharing in the underlying language efficiently, we must support forwarding pointers. To support forwarding pointers in a safe way we must safely transition between the mutator's and collector's view of a value. Doing so requires some ad-hoc reasoning, which is encodable in our framework. We use regions as capabilities to disallow access to unsafe aliases after we cast a value from the mutator's data-invariant and the collector's data-invariant. Our approach is similar in spirit to the one taken by Monnier et al. [MSS01]. In their approach they use a different encoding of forwarding pointers, and transition between the mutator's and collector's data-invariant using more standard scoping techniques.

Related work. The basic intuitions used in our approach to forwarding pointers and Monnier et al. are the same. There must be some mechanism to disallow aliases to objects with incompatible types. We rely on region annotated references to control the aliasing. Monnier et al. only allow one live value of the appropriate type to exist after their cast. Our approach avoids some technical difficulties encountered by Monnier et al.'s approach. In particular for their type-preservation argument to hold they must define a notion of minimally sufficient heap. That is a heap that contains just enough bindings for an expression to be well typed. They basically, need to show that they can strengthen their typing context by throwing away needless bindings without destroying typability. We use a slightly more general lemma in our soundness result for our region calculus in order to properly type only expressions. Type-preservation in our framework reduces to the fact that all our typing rules hold under alpha renaming.

Our approach is also slightly more expressive as we allow more than one value to survive across a cast. We also allow values to live in several different regions before and after a cast. This extra flexibility requires an additional runtime check. However, we believe that in order to support the integration of type-preserving garbage collection and region based memory management we must support this more general mechanism.

Chapter 6

Performance Evaluation

6.1 Introduction

The type-preserving collectors we have described so far have the same asymptotic complexity as existing trusted garbage collectors. However, we should not neglect constant factors and other important pragmatic issues. We have built an experimental system that includes a type-preserving garbage collector in order to understand the performance issues involved. In this chapter we will describe our initial experience with our system and identify potential performance penalties with our approach. We will first compare the performance of our type-preserving garbage collector to a traditional collector using a set of simple microbenchmarks. Then we will validate the observations we make with a larger macrobenchmark.

Input programs. For our microbenchmarks we have chosen several programs seen in the literature on region-based memory management. They are the following:

ackermann Ackermann's function ($n = 3$, $m = 6$) [TT94]

fib Recursive Fibonacci ($n = 33$)

sumit Tail recursive sum of the first n integers (n = 1000) [TT94]

sum Recursive sum of the first n integers ($n = 1000$) [TT94]

hsum Sum the value in a heap-allocated list ($n = 1000$) [TT94]

appel1 Program designed to demonstrate issues of space complexity ($n = 1000$) [TT94]

appel2 Program designed to demonstrate issues of space complexity ($n = 1000$) [TT94]

inline Inline variant of appel1 ($n = 1000$) [TT94]

quicksort Quicksort randomly generated list ($n = 1000$) [TT94]

itrev Recursively build a list of integers and reverse the result using a tail recursive list reverse function ($n = 10,000$)

share-copy Build and reverse a shared list of lists ($n = 10,000$)

These programs are not a representative workload. However, they are sufficient for a preliminary evaluation. It is important to note that our safe collector for **appel1**, **appel2**, and **inline** uses asymptotically less space than approaches based on region inference. Our safe collector is also more robust in that both **appel1** and **inline** have similar space characteristics which is not the case in the original Tofte-Talpin system. For our macrobenchmark we have chosen a slightly modified version of `life` [App92a, page 179]. We will discuss the details of our modifications when we discuss the results of our macrobenchmark.

Compiler. We have modified the back end of the `MLton` [MLT] compiler to accept source programs that include a type-preserving garbage collector. The `MLton` compiler emits C code that is then processed by the system C compiler to produce a runnable program. `MLton` has a breath-first-search (BFS) precise copying collector. The compiler also stack-allocates the environments of return continuations.

The `MLton` front end translates ML programs into a first-order monomorphic intermediate language. The back end then performs optimizations on a CPS-based representation of the front end's intermediate language. Finally, it emits code to be compiled by the system C compiler. The resulting program is linked with a library of primitives that includes a garbage collector. We have designed a first-order intermediate language which has a region type system. Our language can be used as the target language of the `MLton` front end as well as the source language for the back end. So our extensions sit between the front end and back end of `MLton`. This is not the most ideal situation – it would be better to work with the optimized output of the back end – but it simplifies some engineering details.

As outlined in Chapter 4 we CPS-convert and closure-convert the original program. We then region-annotate the resulting program and build a specialized type-preserving garbage collector for it. After every phase we type-check the result to guarantee that the phase has not violated the type safety of the system. However, to minimize the need to modify the back end of `MLton` we transform all region-type variables into expression-level variables. So a function of the form

```
fun f[ρ](...):Ans = ...
```

is transformed into

```
fun f'[](r:rhandle,...):Ans = ...
```

Our translation replaces the normal `MLton` allocation primitives with allocation primitives that take a region handle as an extra argument. After this translation we are still able to type-check the resulting program, but since we have eliminated all of our region-typing constructs and converted them into a term-level objects, type checking no longer guarantees safety. The main goal of these experiments is to understand the performance impact of our approach and not to build a system with a minimal trusted computing base. Therefore, we have not invested the effort needed to add a region type system throughout all phases of the compiler. We consider this to be an area for future work.

Compiling source programs. The source programs for all our microbenchmarks are monomorphic and either already first-order or easily converted to a first-order form. Rather than passing these programs through the `MLton` front end, which may transform our programs in uncontrollable ways, we feed the source programs directly into our first-order intermediate language with regions, using our own trivial front end. These programs are then fed to the `MLton` back end after a type-preserving garbage collector has been added. Our type-preserving garbage collector is written as a depth-first-search (DFS) copy function that uses a stack to manage the auxiliary storage needed for the recursive traversal. For our macrobenchmark we use the output of the `MLton` frontend as the input to the phases that add a type-preserving garbage collector.

For each source programs we collect data for the following variants:

orig Original program passed directly to `MLton`, run with `MLton`'s unsafe collector

cps Program run through CPS transform and first-order closure conversion, run with `MLton`'s unsafe collector

	orig	cps	gc-tbl	gc-fwd
`ackermann`	9	1,530	1,017	1,524
`fib`	9	137	98	131
`sumit`	9	9	2	3
`sum`	9	3,009	2,002	3,003
`hsum`	3,009	3,009	3,005	4,007
`appel1`	2,009	5,013	4,006	7,008
`appel2`	2,009	5,013	4,006	7,008
`inline`	2,009	3,012	2,005	3,007
`quicksort`	3,025	3,025	2,004	3,005
`itrev`	30,025	30,025	20,007	30,008
`share-copy`	50,025	50,025	40,012	60,020
Total	92,147	103,807	78,164	118,725

Table 6.1: Maximum number of live words in the heap.

gc-tbl Same as **cps** using a type-preserving collector with hash table to preserve sharing

gc-fwd Same as **cps** using a type-preserving collector and forwarding pointers which require an extra word of space for each object

6.2 Microbenchmarks

Retention of heap data. For our microbenchmarks we arrange for the garbage collector to be invoked at every safe point and instrument the collector to keep track of the maximum number of four-byte words copied by the collector. The first column (**orig**) of Table 6.1 shows that for programs that are not tail recursive, stack allocating return continuations results in less heap data. We are using a simple flat-closure representation; more advanced closure representation techniques can significantly reduce the amount of allocation [AS00]. For tail-recursive programs, such as **sumit** and **itrev**, there is little or no difference between the first column and the others.

If we compare the second column (**cps**) with the remaining columns we see that type-preserving collectors, with a few exceptions, do not retain any extra live data. There is a small constant difference between the `MLton` collector and the other type-preserving collectors because the `MLton` collector is copying a constant amount of auxiliary system data. All the objects in the **gc-tbl** case are smaller by one word, because these objects do not need either type tags or space reserved for a forwading pointer. Programs in the **gc-tbl** case retain fewer live words because objects are smaller.

The type-preserving collector that uses forwarding pointers also uses an extra word for a forwarding pointer, and because forwarding pointers are mutable values certain unboxing and flattening optimizations the `MLton` backend can apply to both the **cps** and **gc-tbl** programs are not performed for **gc-fwd**. Therefore, for programs such as `appel1` and `appel2` the versions compiled with forwarding pointers allocate more objects. If we generated the type-preserving garbage collector after the unboxing and flattening optimization were performed, this difference should disappear. Ignoring the inefficiency caused by disabling certain optimizations, it seems our type-preserving collectors are equally efficient in reclaiming unused space.

Performance of collectors. Because each approach results in different allocation patterns and object sizes, simply comparing program execution time is not a good way to measure the efficiency of the garbage collector. For example if a program allocates a small amount of data on the heap the overall performance of the program will be largely independent of the garbage collector's efficiency. Moreover, we can always reduce the impact of garbage collection time by allowing a program to use more memory so that the collector is invoked less frequently. Approaches which allow for smaller objects such as the **gc-tbl** case may speed up some programs even though

	cycles/object (std. err.)	cycles/word (std. err.)
cps	103 (2.11)	18 (0.09)
gc-tbl	112 (4.09)	31 (0.18)
gc-fwd	9 (4.56)	31 (0.20)

Table 6.2: Estimated cycle costs for copying.

using a hash table makes copying an object an expensive operation. The space savings in object size may result in fewer invocations of the collector and thus better overall performance.

Rather than measuring program execution time, we will estimate the copying performance our collectors, by assuming the following model:

$$\textit{gc time} = c_1 \cdot \textit{objects collected} + c_2 \cdot \textit{words collected}$$

This assumes that total garbage collection time is simply the sum of time spent collecting each object and that the time spent collecting each object is some constant amount plus the cost collecting each word of the object. We have estimated the per-object and per-word costs by artificially varying both the object size and number of objects collected for our set of programs and then performing a least-squares fit over the data.

We manipulate object sizes by padding objects with needless words. We increase the number of objects collected by varying how often the collector is invoked. We increase the number of collections by simply forcing collector to run after the program has crossed a fixed number of safe points. We use a special hardware register to measure the number of elapsed machine cycles. We also track the number of objects and words copied during each collection. We exclude `fib` and `sumit` since they retain so little live data. Table 6.2 is the result of applying a least squares fit to our data.

We also report the standard error of each parameter. We omit numbers for the **orig** case since it is using exactly the same unsafe collector as **cps**.

Since the unsafe collector is interpreting tags it has a high per-object cost. Our tagless scheme allows us to avoid any tag-interpretation overhead, so the per-object cost for **gc-tbl** reflects the cost of the hash table lookup. For **gc-fwd** our per-object cost is significantly lower than the two previous collectors.

The unsafe collector in **cps** is using a system-optimized `memcpy` function which allows it to have a much smaller per-word cost. Our safe collector is copying objects with a series of naive loads and stores. For objects smaller than eight words our safe collector using forwarding pointers is more efficient than the default unsafe `MLton` collector. We must add a caveat that with such small programs we are ignoring important caching and paging effects in our analysis.

6.3 Macrobenchmark

Our microbenchmarks suggest that the raw copying performance of a type-preserving collector using forwarding pointers is faster than the default unsafe collector for small objects. We also observe that stack-allocating the closures of return continuations can reduce the workload on the collector significantly. We want to check whether these observations generalize to a larger and more realistic program. Therefore, we have measured the performance of a modified version of `life` [App92a, page 179]. We removed the use of polymorphic equality for this benchmark as well as removing the routine that prints the resulting board, but have kept the computational core of the program. Since our prototype system does not yet handle exceptions, we also have removed all exceptions from the program.

We take the modified SML source code of `life` and use the `MLton` front end to

	program allocated		collector copied	
	objects	words	objects	words
orig	2,535,270	7,416,127	222,755	654,320
cps	40,929,057	176,361,759	8,760,603	23,273,826
gc-tbl	54,262,922	151,630,608	11,051,256	19,562,814
gc-fwd	54,261,068	205,885,881	9,810,287	27,185,673

Table 6.3: Allocation and collector workload for `life`.

	program			collector		
	real ms	cpu ms	sys ms	real ms	cpu ms	sys ms
orig	819	730	89	118	110	8
cps	9680	8405	1275	4140	3856	284
gc-tbl	14179	13357	822	7204	6463	741
gc-fwd	8108	7926	182	1664	1605	59

Table 6.4: Execution times for `life` benchmark.

	collector		
	measured ms	estimated ms	measured/estimated
orig	118	69.89	1.41
cps	4140	2659.43	1.36
gc-tbl	7204	3667.66	1.49
gc-fwd	1664	1855.37	0.88

Table 6.5: Accuracy of model.

transform the program into a monomorphic first-order program suitable for our type-preserving garbage collector. We compile the program in the same way we compiled our microbenchmarks. We use the same default dynamic heap resizing policy of `MLton` for all the collectors. Table 6.3 and 6.4 shows the results of our experiments.

Allocation and collector workload. Table 6.3 shows that the **orig** case allocates much less heap data. This significantly reduces the workload placed on the collector. The **cps**, **gc-tbl**, and **gc-fwd** cases all allocate a different number of objects because of the interactions with the `MLton` optimizer. Objects in the **gc-tbl** case are smaller by one word which is reflected in the number of words allocated. Because all the collectors are using a heap resizing policy based on the ratio of live data and the current heap size, the **gc-tbl** case is actually running with a smaller heap. However, this smaller heap causes the collector to be invoked more frequently resulting in more work for the collector. This is reflected in the larger number of words copied by the **gc-tbl** collector. If we ran the programs with a fixed heap size or adjusted the ratio for **gc-tbl** we may see **gc-tbl** actually perform better than the **cps** and **gc-fwd** cases.

Program and collector execution times. Table 6.4 shows that the reduced collector workload and other secondary benefits of stack allocation results in a significantly faster runtime for **orig** than for the other programs. This suggests that stack allocation is an important technique we must be able to support. The **gc-tbl** has the worst runtime, because of the overhead of using a hash table and higher frequency of collection. The **gc-fwd** case is roughly 15% faster than the **cps** case. Notice also that **cps** seems to be spending significantly more time dealing with system interrupts. Examining the paging behavior of these programs we discover that **cps** is incurring nine times as many page faults as **gc-fwd**. We believe this is due to our DFS garbage collection algorithm compared to the BFS algorithm used by **cps**. DFS base collection algorithms have been shown to result in better locality properties compared to BFS algorithms [Jon96, pp. 132– 133]. Our `life` benchmark seems to confirm the basic qualitative predictions of our microbenchmarks.

Accuracy of our quantitative model. Table 6.5 shows the difference between the measured real time of the various garbage collectors and the estimated real time based on the performance model and inferred parameters from Table 6.2. Unfortunately, our performance model seems to be very inaccurate for all of the different collectors. However, observe that the direction of the errors suggest that our performance model is overly optimistic for all of the collectors except for the **gc-fwd** case in where our model seems to be pessimistic.

The most plausible explanation for the difference between our model and reality would be locality-related effects. Both the **orig** and **cps** collectors have bad locality properties because of their BFS algorithm. The **gc-tbl** collector exhibits bad locality because of the use of a hash table. Our **gc-fwd** collector because of its DFS algorithm should have quite good locality, and may even exhibit some benefits from prefetching.

6.4 Summary

Our type-preserving collectors also do not retain more data than traditional collectors. However, they interact badly with certain unboxing and flattening optimizations, so it is better to generate a type-preserving collector after representation optimizations. Our experiments suggest that being able to support stack allocation is an important requirement. Our tagless type-preserving collector using forwarding pointers seems to have better raw copying performance than a traditional collector which interprets tags, if object sizes are small. We also have some evidence that suggests such a type-preserving collector has better locality properties. In Chapter 7 we will discuss how we can support stack allocation without destroying our safety guarantees. Our comparisons are not entirely fair since our type-preserving collectors are using extra auxiliary space during garbage collection. The default `MLton` collector runs in constant auxiliary space. However, we believe we can adopt ideas in the existing literature to make our collectors safely run in constant space [SW67, Che70, SF98].

Chapter 7

Stack Allocation

7.1 Introduction

Our preliminary results show that heap allocating return continuations can cause some programs to allocate significantly more data on the heap. This heap-allocated data must be garbage collected, placing an extra workload on our type-preserving collector. This workload is not placed on garbage collected systems which stack allocate return continuations. If we heap allocate return continuations we can implement advanced control features such as first-class continuations so that both creating and invoking a continuation is a constant time operation [HDB90]. Heap allocated continuations make implementing advanced control structures such as coroutines and exception handlers simple and efficient. However, for programming languages that do not need all these advanced control structures, heap allocating return continuations may not be an ideal solution [App87, MR94, AS96].

Previously, we have placed every return continuation into our global allocation region. However, unlike ordinary heap-allocated objects we know that once a return continuation is created it will be invoked exactly once. The original CPS-based compilers performed simple escape analysis to discover which continuations could be safely stack allocated [Ste78, Kra87]. Other systems used dynamic checking to prevent "one-shot" continuations from being invoked more than once [BWD96].

Others have already studied the semantics of treating continuations as linear objects [Fil92, PP00, BORT00]. Some systems provide explicit type-safe support for stack allocation [MCGW98]. All of these approaches have treated the environment of a return continuation as a private abstract value. However, for the purpose of a type-preserving garbage collector we must deal with the explicit representation of continuations and their environment so that the garbage collector may examine and trace roots accessible from the variables captured in the environment of the continuation. So we treat return continuations as data structures which happen to have an associated `apply` function that interprets the meaning of the continuation.

Review of CPS-based compilation. Figure 7.1 illustrates the compilation of a simple program that computes a list of integers. Observe that there are two function calls, both of which are nontail calls. When the nontail call labeled `A` in the source program is executed there are no variables that need to be preserved across the call. When the nontail call labeled `B` is executed we must save the value of `n` and the previous return address of the function across the call. After CPS converting the source program these observations about what values must be preserved across the call become more apparent.

In the CPS-converted code the return address for the function is represented explicitly as the return continuation that is now an extra argument to the `mklst` function. The return continuation bound to the variable `ret` simply halts the program. Notice that this continuation captures no free variables in its body. The second return continuation bound to the variable `k'` in the body of `mklst` creates a new list cell and

Source Program

```
type lst = Nil | Cons (int, lst)
fun mklst(n:int):lst =
 if n = 0 then Nil
 else let l = mklist(n-1) (* non-tail call B *)
     in Cons(n,l)
(* main program *)
let l = mklst(10)         (* non-tail call A *)
in l
```

CPS Converted

```
type lst = Nil | Cons (int, lst)
type cont = lst → Ans
fun mklst(k:cont, n:int):Ans =
 if n = 0 then (k Nil)
 else let k' = (fn l ⇒ k (Cons(n,l)))
     in mklist(k', n-1)
(* main program *)
let ret = (fn l ⇒ l ; halt())
in mklst(ret, 10)
```

Heap Allocated Closures

```
type lst  = Nil | Cons (int, lst)
type cont = Ret | MkLstK (int, cont)
fun mklst(k:cont, n:int):Ans =
 if n = 0 then apply(k,Nil)
 else let k' = MkLstK(n,k)
     in mklst(k',n-1)
and apply(k:cont, l:lst):Ans =
 case k of
    MkLstK(n,k) ⇒ apply(k,Cons(n,l))
  | Ret ⇒ l ; halt()
(* main program *)
let ret = Ret
in mklst(ret, 10)
```

Figure 7.1: Compilation of `mklst` program.

invokes the return continuation of `mklst`. Notice that this continuation contains two free variables `n` and `k`. This makes our observation about what values must be preserved across the call explicit. Any value that occurs free in the return continuation of the CPS-translated program must be preserved across a function call in the source program. After we perform a first-order closure conversion the set of free variables in every return continuation becomes even more obvious.

After first-order closure conversion each of our return continuations is represented as a data-structure and a block of code in a global dispatch function (`apply`). The type `cont` in the closure-converted code includes a constructor for every unique return continuation. The free variables needed by that return continuation are the values carried by each constructor. It is this `cont` data-structure that a collector must trace at runtime to locate live values. If we performed closure conversion using standard higher-order techniques [MMH96, MWCG98] the collector would not be able to explicitly trace roots in the return continuation.

Special properties of return continuations. In a language without exceptions or first-class continuations, after CPS and first-order closure conversion the type used to represent the closures of return continuations will be of the following form

```
type cont =
    K0
  | K1 (..., cont)
  | K2 (..., cont)
    ...
  | Kn (..., cont)
```

Specifically each constructor of a return continuation type will contain at most one value whose type is used to represent a continuation. This is because there is never more than one return continuation live at any time [DDP00]. In the body of every function a continuation is either passed to another function unchanged, used in the creation of another continuation value or invoked by passing the value to a function such as `apply`. Functions like `apply` completely deconstruct the continuation value.

Since every continuation value occurs at most once in every constructor the type that describes continuation closures looks very much like a list. The head of each cell of our "continuation list" contains the set of free variables captured by the continuation and the tail is simply the previously captured continuation value. Because there is never more than one continuation value live at any point in the program, every cell in our continuation list is pointed to exactly once. This property allows us to represent a continuation as an unrolled data structure so that it can be allocated in a contiguous sequence of memory cells. This means we can "cdr-code" [Han69, CG77, SRA94] our continuation values.

Figure 7.2 shows a type declaration for a set of first-order return continuations and a particular continuation value, along with two possible representations for that continuation value. The linked representation corresponds to a naive heap-based representation. The unrolled representation corresponds to a more efficient representation based on the special properties of return continuations. Notice that the unrolled representation uses a contiguous sequence of memory locations and does not need to include pointers to the next cell in the chain. The next cell can be determined by examining the tag of the current cell and then computing the location of the address of the next cell based on the size of the current cell. The unrolled representation therefore uses less space, because it does not have to include pointers to the next cell. It also will likely have better cache locality because it is using a contiguous sequence of memory locations.

Return Continuations

```
type cont =
   K0
 | K1 (int, cont)
 | K2 (int, int, cont)
 | K3 (int, int, cont)

K1(0, K3(1, 2, K2(3, 4, K0)))
```

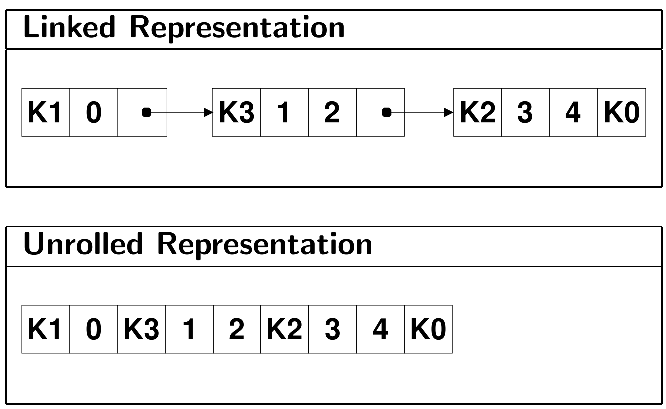


Figure 7.2: Representation of return continuations.

Operations on unrolled lists. Each unrolled continuation list can be represented as a contiguous array of words with an index indicating which is the first unallocated word in the array. When a new cell is added we simply increment the index appropriately checking for overflow. If we do not have enough space in our array to allocate a new word, we simply allocate a fresh array and copy the old contents into our freshly allocated array. Since there are no pointers into the interior of our array we can immediately reclaim the space from our old array. Deconstructing an element from our unrolled list is implemented as a simple pointer decrement. In short, our unrolled list is implemented as a stack and adding a new element is equivalent to a push operation while removing an element from our list is equivalent to a pop operation. However, for this implementation to work properly we are relying on the fact that there are no pointers into the interior of our unrolled list. In the next section we will discuss how to use linear type systems to guarantee this.

7.2 Linear Type Systems

Linear type systems are based on ideas from linear logic developed by Girard [Gir87]. Linear logics are resource-conscious logics. Traditional logics allow unrestricted duplication and discarding of hypotheses. In particular the "structural rules" of contraction and weakening are implicit in the formulation of nonlinear logics. Typically these rules are implicitly introduced by treating the logical context (Γ) as a set of formulas. If we treat the logical context as an unordered sequence of formulas, i.e. a multiset, the structural rules of weakening and contraction would be

$$\frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}\,\text{Contraction} \qquad \frac{\Gamma \vdash B}{\Gamma, A \vdash B}\,\text{Weakening}$$

The contraction rule can be read informally as "Two of the same hypothesis are just as good as one". The weakening rule can be read informally as "An extra hypothesis never hurts." In linear logics these rules do not hold in general. In linear logics two hypothesis are not just as good as one, and adding an extra unneeded hypothesis will prevent certain propositions from being proven.

Linear logics control the use of hypotheses in proofs explicitly. Linear logics encode standard nonlinear logics by adding an "of course" modality ($!A$). Linear logics only allow the rules of contraction and weakening to be applied to nonlinear hypotheses which are indicated by the use of the ($!A$) modality. The rules of contraction and weakening become explicit axioms in linear logic.

From logic to types. Using the proposition-as-types correspondence one can interpret the ideas of linear logic as a type system used to reason about resource consumption in a programming language. This idea is not particularly new and has been studied in depth by others [Laf88, LM92, Abr93, TW99]. Linear type systems have been put forward as one method of avoiding the need for garbage collectors [Laf88, Bak92a, Hof00]. Linear type systems provide an explicit method of managing storage for objects. By carefully controlling the use of values, linear type systems allow programmers to manage memory explicitly. Unfortunately, the semantics of programming languages based on linear type system often require objects be completely copied in order to maintain a "single-pointer property". This requirement limits the efficiency of pure linear programming languages [Laf88, Bak93].

Programming languages based on linear type systems are no replacement for general purpose garbage collection. However, for our particular application treating return continuations as linear values is sufficient. This is because at no point in the program should there ever be more than one reference to it. In languages with first-class continuations there are situations where we must promote the current continuation to a nonlinear value. The operational semantics we will adopt for this promotion will force us to completely copy the continuation so that we can maintain the single pointer invariant. These copy operations are exactly the same copy operations performed in traditional systems that support stack allocation and first-class continuations [HDB90]. So the copies needed in our linear framework merely reflect the existing state of the art.

A linear assembly-language. To make our discussion concrete we will specify a simplified linear assembly-language[1]. Our language is a CPS-converted linear lambda calculus, with nonlinear integer values and linear lists. We use linear lists as a stand-in for return continuations. To keep our discussion focused, our language does not provided any constructs for recursion, but adding first-order functions is a straightforward extension.

Figure 7.3 describes the abstract syntax of our linear assembly language. Although we have a type for linear integers (int), we only provide operations and constants for nonlinear integers ($!\mathsf{int}$). We also include polymorphic lists ($\mathsf{L}(\tau)$) which we treat linearly. A program consists of a triple of memory (M), registers (R), and a term to be evaluated (e). Figure 7.4 describes both the static and dynamic semantics of our language. In our semantics a list is represented as an address in registers which refers indirectly to a list value in memory. The transition $\mathsf{step}_{\mathsf{cons}}$ is the only rule that allocates new values in our memory. The other list operations perform in-place updates of previously allocated lists in memory. Notice that the rule $\mathsf{step}_{\mathsf{casenil}}$ explicitly deallocates the list.

Figure 7.4 has the corresponding typing rules for our language. It is important

[1] Our assembly language is actually much more like a fragment of Static Single-Assignment form.

Syntax

$$
\begin{array}{rrcl}
\textit{variables} & x, y, z & ::= & \ldots \\
\textit{addresses} & a, b, c & ::= & \ldots \\
\textit{types} & \tau & ::= & \mathsf{int} \mid \mathsf{L}(\tau) \mid \; !\tau \\
\textit{terms} & e & ::= & \mathsf{done} \\
 & & \mid & \mathsf{let}\ x = !i\ \mathsf{in}\ e \\
 & & \mid & \mathsf{let}\ x = y - !1\ \mathsf{in}\ e \\
 & & \mid & (\mathsf{if0}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) \\
 & & \mid & \mathsf{let}\ x = \mathsf{nil}\ \mathsf{in}\ e \\
 & & \mid & \mathsf{let}\ x = \mathsf{cons}(y,\ z)\ \mathsf{in}\ e \\
 & & \mid & (\mathsf{case}\ x\ \mathsf{of}\ \mathsf{nil} \Rightarrow e_1,\ \mathsf{cons}(y,\ z) \Rightarrow e_2) \\
\textit{values} & v & ::= & a \mid i \\
\textit{heap values} & hv & ::= & \mathsf{nil} \mid \mathsf{cons}(v,\ hv) \\
\textit{value env} & \Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \\
\textit{memory} & M & ::= & \cdot \mid M[a{:=}hv] \\
\textit{registers} & R & ::= & \cdot \mid R[x{:=}v] \\
\textit{programs} & P & ::= & (M,\ R,\ e)
\end{array}
$$

Figure 7.3: Syntax for Linear Assembly Langauge.

Dynamic Semantics

$$
\begin{array}{lrcl}
\mathsf{step_{done}} & (M,\ R,\ \mathsf{done}) & \mapsto & (M,\ R,\ \mathsf{done}) \\
\mathsf{step_{i}} & (M,\ R,\ \mathsf{let}\ x = !i\ \mathsf{in}\ e) & \mapsto & (M,\ R[x{:=}i],\ e) \\
\mathsf{step_{dec}} & (M,\ R[y{:=}i],\ \mathsf{let}\ x = y{-}!1\ \mathsf{in}\ e) & \mapsto & (M,\ R[y{:=}i][x{:=}i-1],\ e) \\
\mathsf{step_{if0}} & (M,\ R[x{:=}0],\ (\mathsf{if0}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)) & \mapsto & (M,\ R[x{:=}0],\ e_1) \\
\mathsf{step_{ifi}} & (M,\ R[x{:=}i],\ (\mathsf{if0}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)) & \mapsto & (M,\ R[x{:=}i],\ e_2)\ \ i \neq 0 \\
\mathsf{step_{nil}} & \multicolumn{3}{l}{(M,\ R,\ \mathsf{let}\ x = \mathsf{nil}\ \mathsf{in}\ e) \mapsto (M[a{:=}\mathsf{nil}],\ R[x{:=}a],\ e)\ \ a \notin dom(M)} \\
\mathsf{step_{cons}} & \multicolumn{3}{l}{(M[a{:=}hv],\ R[y{:=}v][z{:=}a],\ \mathsf{let}\ x = \mathsf{cons}(y,\ z)\ \mathsf{in}\ e)} \\
 & \multicolumn{3}{r}{\mapsto (M[a{:=}\mathsf{cons}(v,\ hv)],\ R[y{:=}v][x{:=}a],\ e)} \\
\mathsf{step_{casenil}} & \multicolumn{3}{l}{(M[a{:=}\mathsf{nil}],\ R[x{:=}a],\ (\mathsf{case}\ x\ \mathsf{of}\ \mathsf{nil} \Rightarrow e_1,\ \mathsf{cons}(y,\ x) \Rightarrow e_2))} \\
 & \multicolumn{3}{r}{\mapsto (M,\ R,\ e_1)} \\
\mathsf{step_{casecons}} & \multicolumn{3}{l}{(M[a{:=}\mathsf{cons}(v,\ hv)],\ R[x{:=}a],(\mathsf{case}\ x\ \mathsf{of}\ \mathsf{nil} \Rightarrow e_1,\ \mathsf{cons}(y,\ x) \Rightarrow e_2))} \\
 & \multicolumn{3}{r}{\mapsto (M[a{:=}hv],\ R[y{:=}v][z{:=}a],\ e_2)}
\end{array}
$$

Static Semantics

$$\boxed{\Gamma \vdash e}$$

$$\frac{}{\Gamma \vdash \mathsf{done}}\ \mathsf{wf_{done}}$$

$$\frac{\Gamma, x{:}!\mathsf{int} \vdash e}{\Gamma \vdash \mathsf{let}\ x = !i\ \mathsf{in}\ e}\ \mathsf{wf_{i}} \qquad \frac{\Gamma, y{:}!\mathsf{int}, x{:}!\mathsf{int} \vdash e}{\Gamma, y{:}!\mathsf{int} \vdash \mathsf{let}\ x = y{-}!1\ \mathsf{in}\ e}\ \mathsf{wf_{dec}}$$

$$\frac{\Gamma, x{:}!\mathsf{int} \vdash e_1 \quad \Gamma, x{:}!\mathsf{int} \vdash e_2}{\Gamma, x{:}!\mathsf{int} \vdash (\mathsf{if0}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)}\ \mathsf{wf_{if0}}$$

$$\frac{\Gamma, x{:}\mathsf{L}(\tau) \vdash e}{\Gamma \vdash \mathsf{let}\ x = \mathsf{nil}\ \mathsf{in}\ e}\ \mathsf{wf_{nil}} \qquad \frac{\Gamma, x{:}\mathsf{L}(\tau) \vdash e}{\Gamma, y{:}\tau, z{:}\mathsf{L}(\tau) \vdash \mathsf{let}\ x = \mathsf{cons}(y,\ z)\ \mathsf{in}\ e}\ \mathsf{wf_{cons}}$$

$$\frac{\Gamma \vdash e_1 \quad \Gamma, y{:}\tau, z{:}\mathsf{L}(\tau) \vdash e_2}{\Gamma, x{:}\mathsf{L}(\tau) \vdash (\mathsf{case}\ x\ \mathsf{of}\ \mathsf{nil} \Rightarrow e_1,\ \mathsf{cons}(y,\ z) \Rightarrow e_2)}\ \mathsf{wf_{casel}}$$

Figure 7.4: Semantics for Linear Assembly Langauge.

Stack-allocated Continuations

```
type lst  = Nil | Cons (int, lst)
lintype cont = Ret | MkLstK (int, cont)
fun mklst(k:cont, n:int):Ans =
 if n = 0 then apply(k,Nil)
 else let k' = MkLstK(n,k)
      in mklst(k',n-1)
and apply(k:cont, l:lst):Ans =
 lincase k of
    MkLstK(n,k) ⇒ apply(k,Cons(n,l))
  | Ret ⇒ l ; halt()
(* main program *)
let ret = Ret
in mklst(ret, 10)
```

Figure 7.5: Stack-allocated continuations.

to remember that our value environments do not have implicit rules for contraction and weakening of values of type $!\mathsf{int}$. Later we will add explicit rules and term level operators to provide for contraction and weakening. As is standard, we assume that all variables are unique within our value environment. It is worth comparing the rules $\mathsf{wf}_{\mathsf{dec}}$ and $\mathsf{wf}_{\mathsf{cons}}$ to understand the difference between rules for linear and nonlinear values. In the rule $\mathsf{wf}_{\mathsf{dec}}$ we type the subterm of the decrement expression in a value environment where we have extended the original environment with a new binding $(x : !\mathsf{int})$. In the rule $\mathsf{wf}_{\mathsf{cons}}$ we type the subterm in an environment where we add a binding $(x : \mathsf{L}(\tau))$ and remove two bindings $(y : \tau$ and $z : \mathsf{L}(\tau))$. The removal of bindings prevents the subterm from accessing pointers to the older version of the list.

Stack allocation of return continuation. Figure 7.5 is a CPS-converted and closure-converted version of our original `mklst` program. The program is identical to the version that heap allocates return continuations except that the continuation type is declared to be a linear type with the lintype declaration. The apply function now uses lincase to pattern match against the linear continuation. Because the continuation type `cont` is treated linearly we can represent it as an unrolled value. Since the storage for linear values is managed explicitly, linear values do not need to be heap allocated.

7.3 Compiling Advanced Control Structures

Supporting first-class continuation. If we wish to support advanced control structures such as call-with-current-continuation we must be able to treat our linear lists which represent return continuations as nonlinear values. Figure 7.6 describe the needed extensions to support nonlinear values. The first two new terms copy and kill are general operators on nonlinear values. They represent explicit contraction and weakening rules similar to the terms found in Benton et al [BBdH93]. The kill term can be thought of as a static hint that a particular variable is dead. Notice that copy does not in fact perform a copy of a list value, but merely copies its address.

Benton et al. include two operators called "promote" and "derelict" which are used to convert linear values to nonlinear values and vice versa. Their constructs operate on arbitrary values. A general promotion rule would require that the promoted value contain no free linear variables. However, in our assembly language the only values are free variables. Instead of standard promotion and dereliction terms we introduce two nonstandard terms freeze and thaw whose operational behavior is similar to the primitives used in existing systems that support first-class continuations and are limited only to lists. The freeze operator is basically a no-op[2]. However, in our static semantics we have converted a linear value into a nonlinear value. Since we have no

[2] It is actually a needless register to register move which a coalescing register allocator should remove.

Syntax

$$
\begin{array}{llll}
\textit{terms} & e & ::= & \ldots \\
 & & \mid & \textsf{copy}\ x\ \textsf{as}\ y, z\ \textsf{in}\ e \\
 & & \mid & \textsf{kill}\ x\ \textsf{in}\ e \\
 & & \mid & \textsf{let}\ x = \textsf{freeze}(y)\ \textsf{in}\ e \\
 & & \mid & \textsf{let}\ x = \textsf{thaw}(y)\ \textsf{in}\ e
\end{array}
$$

Dynamic Semantics

$$
\begin{array}{lrcl}
\textsf{step}_{\textsf{copy}} & (M,\ R[x:=v],\ \textsf{copy}\ x\ \textsf{as}\ y,z\ \textsf{in}\ e) & \mapsto & (M,\ R[y:=v][z:=v],\ e) \\
\textsf{step}_{\textsf{killi}} & (M,\ R[x:=i],\ \textsf{kill}\ x\ \textsf{in}\ e) & \mapsto & (M,\ R,\ e) \\
\textsf{step}_{\textsf{killa}} & (M[a:=hv],\ R[x:=a],\ \textsf{kill}\ x\ \textsf{in}\ e) & \mapsto & (M,\ R,\ e) \\
\textsf{step}_{\textsf{freeze}} & (M[a:=hv],\ R[y:=a],\ \textsf{let}\ x = \textsf{freeze}(y)\ \textsf{in}\ e) & & \\
 & & & \mapsto (M[a:=hv],\ R[x:=a],\ e) \\
\textsf{step}_{\textsf{thaw}} & (M[a:=hv],\ R[y:=a],\ \textsf{let}\ x = \textsf{thaw}(y)\ \textsf{in}\ e) & & \\
 & \multicolumn{3}{r}{\mapsto (M[a:=hv][b:=hv],\ R[y:=a][x:=b],\ e)\quad b \notin dom(M)}
\end{array}
$$

Static Semantics

$\boxed{\Gamma \vdash e}$

$$
\frac{\Gamma, y:!\tau, z:!\tau \vdash e}{\Gamma, x:!\tau \vdash \textsf{copy}\ x\ \textsf{as}\ y,z\ \textsf{in}\ e}\ \textsf{wf}_{\textsf{copy}}
\qquad
\frac{\Gamma \vdash e}{\Gamma, x:!\tau \vdash \textsf{kill}\ x\ \textsf{in}\ e}\ \textsf{wf}_{\textsf{kill}}
$$

$$
\frac{\Gamma, x:!\textsf{L}(\tau) \vdash e}{\Gamma, y:\textsf{L}(\tau) \vdash \textsf{let}\ x = \textsf{freeze}(y)\ \textsf{in}\ e}\ \textsf{wf}_{\textsf{freeze}}
\qquad
\frac{\Gamma, x:\textsf{L}(\tau) \vdash e}{\Gamma, y:!\textsf{L}(\tau) \vdash \textsf{let}\ x = \textsf{thaw}(y)\ \textsf{in}\ e}\ \textsf{wf}_{\textsf{thaw}}
$$

Figure 7.6: Extended Linear Assembly Langauge.

Source Program

```
type lst = Nil | Cons (int, lst)
fun mklst(uk:lst cont,n:int):lst =
 if n < 0 then throw(uk,Nil)
 else if n = 0 then Nil
 else let l = mklist(uk,n-1)
     in Cons(n,l)
(* main program *)
let l = letcc uk in mklst(uk,10)
in l
```

CPS Converted

```
type lst = Nil | Cons (int, lst)
type cont = lst → Ans
type ucont = cont
fun mklst(k:cont, ucont:uk, n:int):Ans =
 if n < 0 then (uk Nil)
 else if n = 0 then (k Nil)
 else let k' = (fn l ⇒ k (Cons(n,l)))
     in mklist(k', uk, n-1)
(* main program *)
let ret = (fn l ⇒ l ; halt())  in
let uk = ret
in mklst(ret, uk, 10)
```

Figure 7.7: Compilation of `mklst` with first-class continuations.

primitive operators on nonlinear lists the only thing we can do with a nonlinear list is to duplicate or store its address in other values. Finally, our thaw term converts our “frozen” list back into a linear value by creating a copy and returning a unique pointer to the copy.

Compiling first-class continuations. Figure 7.7 contains a variation of our `mklst` program that uses first-class continuations and the same program after CPS conversion with explicit control semantics. Our modified `mklst` function takes a user continuation (`uk`) as a parameter. In the body of the function we invoke the user continuation with the value `Nil` when the integer passed to it is less than zero. We invoke `mklst` with a user continuation provided by a letcc binding which binds the current continuation to the variable `uk`. The CPS-converted version of the program makes the semantics of our source program explicit. Notice that our CPS-converted version of the program takes two continuations, `k`, which was inserted by the CPS transform, and `uk`, which is a user continuation from our source program. Notice how the letcc construct is compiled. We bind `uk` to the return continuation (`ret`) created by the CPS transform. Now that we have made the desired control semantics explicit, we need to choose a method to represent the continuations.

Figure 7.8 contains two approaches. The first uses heap-allocated linked continuations. In this model all continuations are heap allocated and capturing and invoking a continuation are both constant time operations. Unfortunately, as we have pointed out, before heap allocating all continuations increases the workload of the garbage collector and is not an ideal solution if the use of first-class continuations is not frequent. The other approach treats return continuations as linear values so that they can be allocated in a stack like way. User continuations are linear values that have been turned into nonlinear values. We distinguish between linear types that have been converted into nonlinear types with the "!" operator from types that were never linear in the first place. So we have two different incompatible kinds of nonlinear type. This distinction allows us to represent objects that were never linear in the first place as linked data-structures, while treating linear values promoted to nonlinear values as simply pointers to an underlying linear object which maybe freely duplicated. The version that uses linear continuations is basically the same as the heap allocated version except that there are explicit freeze and thaw coercions inserted

Heap Allocated Continuations

```
type lst  = Nil | Cons (int, lst)
type cont = Ret | MkLstK (int, cont)
fun mklst(k:cont, uk:cont, n:int):Ans =
 if n < 0 then apply (uk,Nil)
 else if n = 0 then apply(k,Nil)
 else let k' = MkLstK(n,k)
      in mklst(k',n-1)
and apply(k:cont, l:lst):Ans =
 case k of
    MkLstK(n,k) ⇒ apply(k,Cons(n,l))
  | Ret ⇒ l ; halt()
(* main program *)
let ret = Ret in
let uk = ret
in mklst(ret, uk, 10)
```

Stack Allocated Continuations

```
type lst  = Nil | Cons (int, lst)
lintype cont = Ret | MkLstK (int, cont)
lintype ucont = !cont
fun mklst(k:cont, uk:ucont, n:int):Ans =
 if n < 0 then let k = thaw uk in apply (k,Nil)
 else if n = 0 then apply(k,Nil)
 else let k' = MkLstK(n,k)
      in mklst(k',n-1)
and apply(k:cont, l:lst):Ans =
 lincase k of
    MkLstK(n,k) ⇒ apply(k,Cons(n,l))
  | Ret ⇒ l ; halt()
(* main program *)
let ret = Ret in
let uk = freeze ret in
let ret = thaw uk in
in mklst(ret, uk, 10)
```

Figure 7.8: Implementing first-class continuations.

Source Program

```
type lst = Nil | Cons (int, lst)
fun mklst(n:int):lst =
 if n < 0 then raise Fail
 else if n = 0 then Nil
 else let l = mklist(n-1)
     in Cons(n,l)
(* main program *)
let l = mklst(10) handle Fail ⇒ Nil
in l
```

CPS Converted

```
type lst = Nil | Cons (int, lst)
type cont = lst → Ans
type econt = unit → Ans
fun mklst(k:cont, eh:econt, n:int):Ans =
 if n < 0 then eh ()
 else if n = 0 then k Nil
 else let k' = (fn l ⇒ k (Cons(n,l)))
     in mklist(k',eh,n-1)
(* main program *)
let ret = (fn l ⇒ l ; halt()) in
let eh = (fn () ⇒ ret Nil)
in mklst(ret, eh, 10)
```

Figure 7.9: Compilation of `mklst` program with exceptions.

that copy and restore our representation of the control stack appropriately. The copy makes the work needed to capture and invoke a continuation proportional to the size of the control stack. We believe our calculus is expressive enough to encode hybrid schemes [HDB90] that bound the amount of copying needed to a constant factor.

Compiling exceptions. First-class continuations are powerful control structures that can be used to implement many other control structures. However, many languages do not fully support them because they are hard to implement efficiently when return continuations are stack allocated. Exception-handling mechanisms are more common since they are easy to implement efficiently in the presence of stack-allocated return continuations.

Figure 7.9 shows a modified version of `mklst` that uses exceptions. The CPS-converted version of `mklst` makes the control semantics of exceptions explicit. For simplicity we assume there is only one exception (`Fail`) that can be raised. In the CPS-converted version the `mklst` function is passed two continuation arguments. The first is the standard return continuation (`k`). The other continuation is the exception handler (`eh`) and is invoked to signal an error. Notice that the initial exception handler for the program invokes the initial return continuation (`ret`) to return a value. In general an exception handler will always either invoke the return continuation for the expression that it handles or invoke the current exception handler.

Figure 7.10 describes a straightforward implementation using heap-allocated continuations. Notice that the initial return continuation (`ret`) is passed to `mklst` and used by initial exception handler. This prevents us from treating return continuation as linear objects, using the linear type system we have presented so far. Implementing exceptions in this way is quite simple, but again puts a larger workload on the garbage collector. One positive aspect is that installing a handler and invoking the handler are both constant time operations.

Stack unwinding. Figure 7.11 presents a different implementation scheme that allows for stack allocation of return continuations. It exploits the fact that there is a one-to-one mapping between exception handlers and return continuations. We can uniquely identify an exception handler by associating it with the return continuation of the expression which it handles. Using this fact we can stack allocate return

Heap-allocated Continuations

```
type lst = Nil | Cons (int, lst)
type cont = Ret | MkLstK (int, cont)
type econt = HandleTop (cont)
fun mklst(k:cont, eh:econt, n:int):Ans =
 if n < 0 then eapply eh
 else if n = 0 then apply(k,Nil)
 else let k' = MkLstK(n,k)
     in mklist(k',eh,n-1)
and apply(k:cont, l:lst):Ans =
 case k of
    MkLstK(n,k) ⇒ apply(k,Cons(n,l))
  | Ret ⇒ l ; halt()
and eapply(e:econt):Ans =
 case k of
  HandleTop k ⇒ apply(k,Nil)
(* main program *)
let ret = Ret in
let eh = HandleTop(ret)
in mklst(ret, eh, 10)
```

Figure 7.10: Heap-allocated continuations.

Stack Unwinding

```
type lst = Nil | Cons (int, lst)
lintype cont = Ret | MkLstK (int, cont)
fun mklst(k:cont, n:int):Ans =
 if n < 0 then eapply k
 else if n = 0 then apply(k,Nil)
 else let k' = MkLstK(n,k)
      in mklist(k',n-1)
and apply(k:cont, l:lst):Ans =
 lincase k of
    MkLstK(n,k) ⇒ apply(k,Cons(n,l))
  | Ret ⇒ l ; halt()
and eapply(e:cont):Ans =
 lincase k of
    MkLstK(n,k) ⇒ eapply(k)
  | Ret ⇒ apply (Ret, Nil)
(* main program *)
let ret = Ret
in mklst(ret, 10)
```

Figure 7.11: Stack unwinding.

continuations and invoke exception handlers by simply unwinding the current chain of return continuations until we find a return continuation that has an exception handler associated with it. The function `eapply` is the stack unwinding function that determines what exception handler is invoked by unwinding the exception handler. Our type system will allow us to use one of these pointers but not both simultaneously.

Figure 7.12 sketches how such a system would work. Every function now takes a control state argument which is an additive pair of either the return continuation or the exception handler. Notice that when we build the initial control state (`cs`) we build an additive pair whose first component and second component both refer to the same linear value `ret`. Just as in the logical rule for additive pairs our type

Stack Cutting

```
type lst = Nil | Cons (int, lst)
lintype cont = Ret | MkLstK (int, cont)
lintype econt = HandleTop (cont)
lintype cstate = cont & econt
fun mklst(cstate:cs, n:int):Ans =
 if n < 0 then eapply #2(cs)
 else if n = 0 then apply(#1(cs),Nil)
 else let cs' = (MkLstK(n,#1(cs)) & #2(cs))
     in mklist(cs',n-1)
and apply(k:cont, l:lst):Ans =
 lincase k of
    MkLstK(n,k) ⇒ apply(k,Cons(n,l))
  | Ret ⇒ l ; halt()
and eapply(e:econt):Ans =
 lincase k of
  HandleTop k ⇒ apply(k,Nil)
(* main program *)
let ret = Ret in
let cs  = (ret & HandleTop(ret))
in mklst(cs, 10)
```

Figure 7.12: Stack cutting.

systems allows us to use the same resource twice. To invoke an exception we select one component of the pair and either return or invoke an exception handler. Also, note that when we build a new control state (`cs'`) we are allowed to decompose the old additive pair twice, because constructing a new pair allows us to duplicate the linear resources in the current typing context. We must place a few extra restrictions on additive pairs to make sure our operational model is sound, just as we have with freeze and thaw which are specialized versions of well-known linear logical operators.

7.4 Summary

Using the ideas from linear logic we can encode many of the important resource constraints needed to verify the safety of many advanced control features. We have shown how to encode the major techniques for first-class continuations and exception handlers. User-level thread packages which explicitly manage stacks on a per thread basis are also encodeable with the framework we have outlined. What is more important is that our system is flexible enough so that one can choose different techniques for implementing first-class continuations, exception handling, stack allocation, and threads simultaneously. We have provided the basic abstractions from which more complicated structures can be built in a way that allows us to maintain the underlying type-safety guarantees. The idea of using linear continuations to implement many advanced control structures efficiently is not new. Ramsey and Peyton Jones [RJ00] outline a similar idea in a low-level compiler intermediate language. Their system is not type safe and they do not enforce the linearity constraints required. Our type system is not particularly novel either. However, our decision to represent the control stack as an explicit data-structure provides clean semantics for previously low-level operations of runtime systems. Since our system is type-safe we give the compiler more implementation flexibility in implementing advanced control structures without compromising safety or efficiency.

Chapter 8

Separate Compilation

8.1 Introduction

The type-preserving garbage collector that is outlined in Chapter 3, unfortunately, assumes the compiler has access to the whole program at compile time. There are many whole-program compilation systems that perform aggressive optimizations which require access to the entire program [Dea96, CJW00]. However, whole-program compilation is not acceptable in many situations. In this chapter we will describe how to relax the whole-program assumption by adapting existing approaches to support a more modular compilation scheme.

Cardelli [Car97] presents one of the few formal accounts of separate compilation. Unfortunately, Cardelli abstracts the problem of separate compilation to the problem of typechecking *program fragments* in isolation. In Cardelli's model program fragments are programs with free identifiers. Cardelli formally describes how to compute typing assignments to free identifiers and proves some interesting properties about his model of separate compilation. Cardelli's model of separate compilation hides many important implementation issues. We wish to understand the issues of separate com-

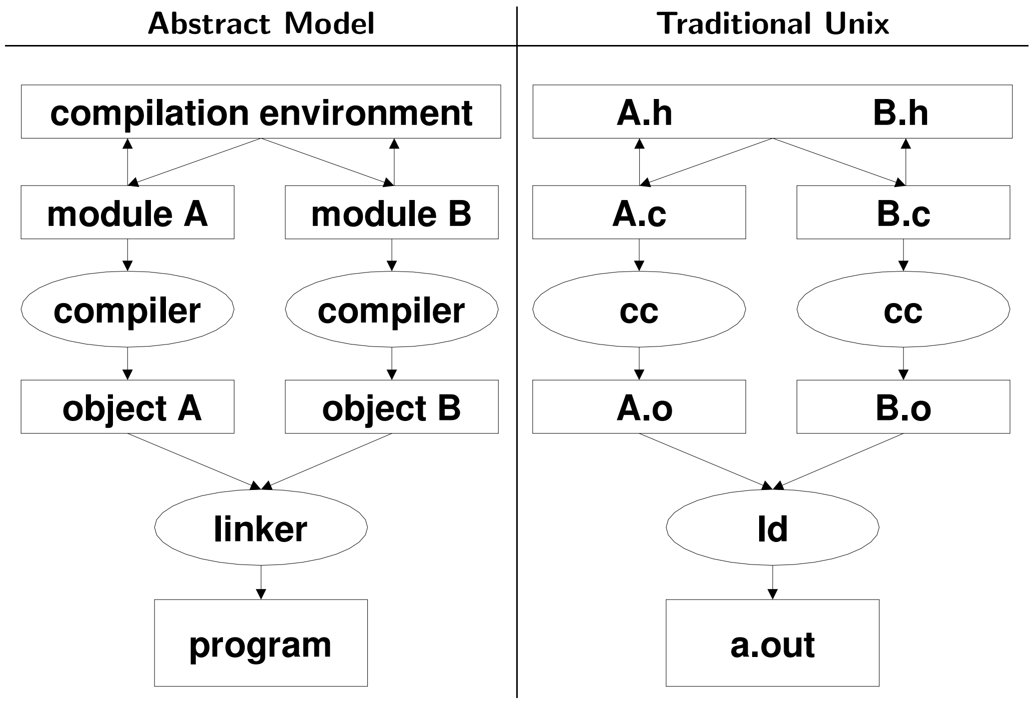


Figure 8.1: Separate-compilation architectures.

pilation at a level that exposes many more pragmatic issues.

Separate compilation architectures. Figure 8.1 depicts our abstract model of how program fragments are combined to form a whole program and a particular instantiation of our model found in a traditional Unix environment. We refer to program fragments as *modules*. For modules to be compiled separately they must import information from a *compilation environment* which is shared across modules. The compilation environment is built from information exported by modules into the environment. Each module is converted to an *object* by use of a *compiler*. Objects are linked to create a complete program by a *linker*. Although we use traditional terms to suggest their roles in the process of separate compilation it is important to think of them in completely abstract ways. A compiler is any function that converts program fragments to objects which themselves are abstract values consumed by a linker to produce a program.

We will refer to a specific choice of compilation environment, module, compiler, object, linker, and program as a *separate compilation architecture*. Architectures differ primarily in how work is divided between the compiler and linker, as well as what information is contained in the compilation environment.

For example in a traditional Unix environment modules export `.h` files that define the compilation environment. Modules themselves are simply C programs. The compiler is the just the standard C compiler (`cc`) which produces object files. Object files are machine code with relocation and symbol information. The linker is the normal Unix linker (`ld`) which merges object files by relocating machine code and patching symbol information to produce a runnable program. In this architecture, the compiler does the majority of the work while the linker's job is relatively simple, and only type information is shared in the compilation environment.

Properties of separate compilation architectures. Figure 8.2 describes two different separate compilation architectures that can be built from the standard Unix tool set. They both demonstrate certain bad properties we wish to avoid. One of the architectures is *tardy*, since it delays too much work until link time. The other architecture is *fragile* since it reveals so much information in the compilation environment that changes to a module may require the recompilations of other modules. In the tardy architecture the compiler simply produces objects by leaving the module untouched. The compiler does no work and forces the linker to do the actual compilation.

Some tardy architectures. While tardy architecture seems useless, it is very close to the separate compilation architecture used by mobile-code systems such as

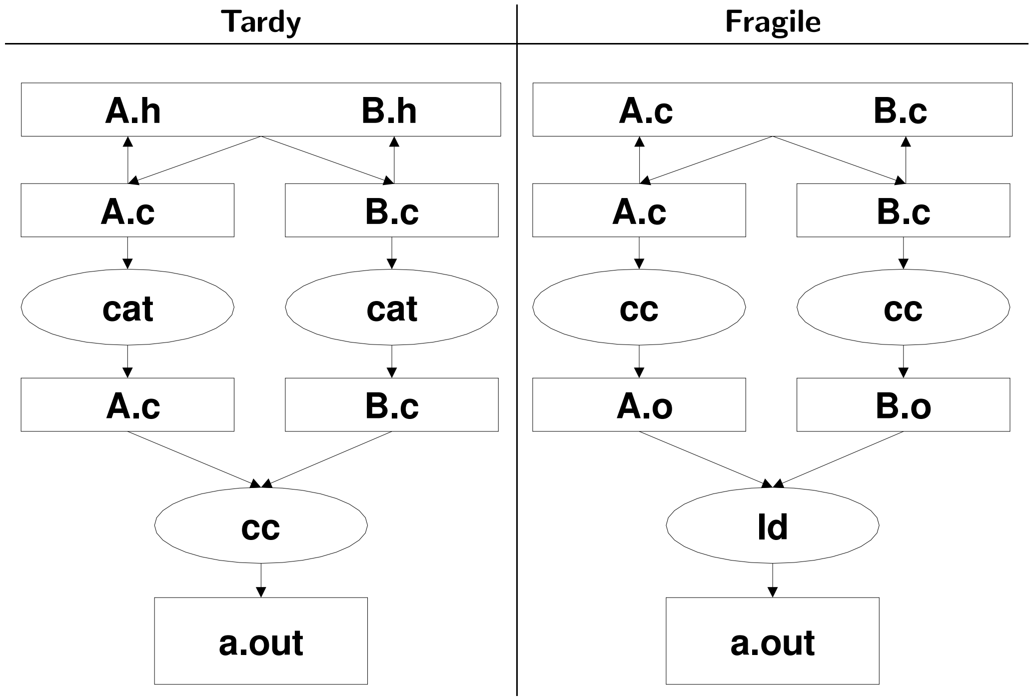


Figure 8.2: Tardy and fragile architectures.

Java. The Java bytecode compiler does a minimal amount of work when it converts Java source code to Java bytecode. Java bytecode, for all practical purposes, is preparsed Java source code. The Java "Just In Time" compiler plays the role of a linker by combining several Java bytecode files into an executable program. Many other systems have similar architectures [SW92, Fer95]. Delaying work until link time has the advantage of enabling link-time optimizations that could not be performed by the compiler [Fer95, Dea96]. So some amount of tardiness is not a bad thing.

Fragile architectures. In general any work done at compile time can be moved to link time, since at link time more information is available to the linker than the compiler. We can always avoid some tardiness in our architecture by exporting more information in the compilation environment. We would like to move a minimal amount of information into the compilation environment, since revealing too much information to other modules destroys the modularity of our system. The fragile architecture in Figure 8.2 represents an extreme case where the compilation environment contains the actual source code of every module.

In this scenario, the compiler can examine the code for `B.c` while compiling `A.c`. It produces an object file only for `A.c`, but having access to `B.c` allows the compiler to perform some optimizations such as inlining calls from `B.c` that could only previously be done at link time. To guarantee some amount of sanity the compiler should include a checksum of the compilation environment it used when compiling `A.c` and `B.c`. At link-time the linker should check that the checksum for the compilation environment used to compile the modules separately is the same. If they differ linking should fail. Unfortunately, a small change to `B.c` may require that `A.c` be recompiled if we wish for linking to proceed.

Fragile compilation architectures are found in languages such as C++ which force users to export the actual source code of small functions they wish to inline across modules at compile time. More transparent systems that export small functions across modules exists for languages like Standard ML [BA97].

Opportunities for optimization. The amount of information in the compilation environment and the amount of work we are willing to perform at link time constrains the set of optimizations we can perform on programs and the overall quality of the code produced. At compile time the set of optimizations we can perform is limited primarily by the amount of information available to the compiler from the compilation environment. At link time we have access to the whole program so in theory any valid optimization can be performed. However, if we wish linking to be “fast” then the set of optimization we perform is limited by the amount of time available. For example, if our notion of a “fast” linker is any linker that runs in linear time with respect to its

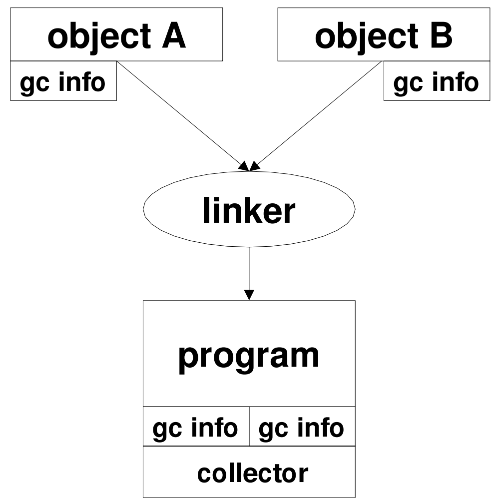


Figure 8.3: Linking architectures and garbage collection.

input[1] then we cannot perform quadratic optimizations such as graph-coloring based global register allocation.

Separate compilation architectures and garbage collection. So far we have been talking about separate compilation architectures in a general way. Garbage collectors add additional complications. If we wish our garbage collector to precisely locate all the program roots and be able to trace the program heap it must be provided with global type information. The problem a garbage collector faces is similar to the problems faced by debuggers.

Compilers typically include information needed for debugging in the object files they produce. Linkers merge the information into a global table so that at run time a debugger has access to information about all the program modules that provide debugging information. A garbage collector needs approximately the same information as a debugger, but it must have it for every module in the program, not just those which are willing to provide it. Figure 8.3 depicts the situation in terms of our

[1]We must assume some resonable complexity bounds on the compiler's output for our notion of fast to be useful.

previous linking architecture. Typically the garbage collection information cannot be exported into the shared compilation environment, since much of the information the garbage collector needs, such as the precise data layout of values and stack frames is available only after compilation, especially if the compiler is performing optimizations [DMH92, JRR99].

In the presence of optimization compilers omit debugging information because maintaining that information becomes difficult. Compilers that support garabage collection cannot omit information because that would cause the collector to fail. The compiler must preserve the information in the presence of optimization or deliberately disable optimizations [DMH92, Cop94].

Preserving abstraction. Debuggers inspect program state that is hidden from normal programs. For example, a function typically cannot examine its calling context at runtime and determine what function is its caller. A function that violates this abstraction boundary can behave in unexpected ways [App96]. However, debuggers can provide a backtrace of function invocations. In this respect the debugger violates the abstractions provided by the language. Garbage collectors also examine the calling context and other data in a way that violates the source abstractions of the language.

Informally one can think of an abstraction gurantee as a gurantee of data privacy. Data privacy guarantees are useful to preserve during the process of compilation. Many typed-based compilation techniques seem to provide strong abstraction guarantees [MMH96, MWCG99, MCGW98, Gle00]. However, none of these techniques have properly considered the role of the garbage collector, since the collector, like a debugger, is outside the model they consider. Now that our collector is part of the model of our langauge we must decide what to do about abstraction guarantees. We can either make the fact the garbage collector violates these abstractions explicit or

	Linker work	Fragile changes
WP	Compile program and build copy function	Any modification
ES	Build jump tables and merge copy function	Dispatch function signature
ITA	Merge type information link with collector	New type constructors

WP = Whole program with first-order closure conversion
ES = Extensible Sums with first-order closure conversion
ITA = Intensional Type Analysis with higher-order closure conversion

Table 8.1: Comparison of various linking approaches.

attempt to preserve the abstraction guarantees in the presence of garbage collection.

Three different architectures. Table 8.1 describes different architectures for supporting separate compilation in the presence of garbage collection. For each architecture we describe what work must be done at link time and what changes may cause the recompilation of every module in the system.

Whole program compilation with first-order closure conversion is the technique we have described in Chapter 3. The "linker" (which is actually the compiler) must compile the entire program and build a global copy function needed by the garbage collector at "link time". Potentially any change to a single module file may require recompilation of all modules. Because of the first-order closure conversion the source level abstractions are not preserved.

All of the architectures described in the table have been used in realistic systems to support separate compilation in higher-order languages such as ML. We will focus our discussion primarily on how to extend these existing techniques for type-preserving garbage collection. We have already described how to extend type-preserving garbage collection techniques to whole-program compilation in Chapter 3. So we will focus on the remaining two approaches.

compilation environment
`type lst = Nil | Cons (int, lst)` `val mklst : int → lst (* exported by module A *)`

module A	module B

`fun mklst(n:int):lst =`
` if n = 0 then Nil`
` else Cons(n,mklist(n-1))`
	`fun itrev(l:lst, acc:lst):lst =`
	`   ...`
	`(* program entry point *)`
	`let l = mklst(10) in`
	`let rl = itrev(l, Nil)`
	`in rl`

Figure 8.4: Separate-compilation example.

8.2 Extensible Sums

Figure 8.4 describes two modules. The first module A defines and exports a function (`mklst`) into the compilation environment while module B calls the function (`mklst`). These two modules will serve as our running example throughout this section. We will describe how to separately compile each module using extensible sums and first-order closure-converstion techniques.

What are extensible sums? An extensible sum is like an ordinary tagged union except that new constructors can be added in a modular and local way. Exceptions in ML are one such example of an extensible sum. An exception declaration creates a new constructor that injects values into a single universal exception type. Extensible sums are an instance of more general mechanisms commonly seen in object-oriented langauges. They can be generalized to allow for a hierarchy of extension [RR96, Gle99]. For our purpose we can describe our technique using nothing more complicated then ML-style exceptions as extensible sums. We will in fact

```
object A
imports
  type lst = Nil | Cons (int, lst)
  type cont = exn
  val topApply    : (cont, lst) → Ans
exports
  val mklst       : (cont, int) → Ans
  val mklst_apply : (cont, lst) → Ans
  exception MkLst_K(int, cont)
is
  fun mklst(k:cont, n:int):Ans =
   if n = 0 then topApply(k,Nil)
   else let k' = MkLst_K(n,k)
        in mklist(k',n-1)
  and mklst_apply(k:cont, l:lst):Ans =
    case k of
     MkLst_K(n,k) ⇒ topApply(k,Cons(n,l))
```

Figure 8.5: Object for module A.

use a more restrictive form of exceptions which are not generative[2] and can only be declared globally. Adding these restrictions allows us to choose a representation of exceptions at link time that allows for efficient dispatch.

Extensible sums for separate compilation. Figure 8.5 is the result of our compiler applying a CPS and first-order closure conversion to the source of module A. It represents an object in our architecture. Our object is very much like the *linkset* described by Cardelli [Car97]. The object consists of a set of import and export declarations as well as the code that implements the exported declarations. All objects will be in CPS and closure-converted form. Such a form is very close to machine code [Kel95, App98].

Object A imports the two type definitions that describe the representations of

[2]In ML each exception declaration introduces a dynamically unique constructor, requiring a fresh tag to be created at runtime.

```
object B
imports
  type lst = Nil | Cons (int, lst)
  type cont = exn
  val topApply    : (cont, lst) → Ans
  val mklst       : (cont, int) → Ans
exports
  val itrev_apply : (cont, lst) → Ans
  exception ItRev_K1
  exception ItRev_k2
is
  fun itrev(k:cont, l:lst, acc:lst):Ans =
   case l of Nil ⇒ topApply(k, acc)
    | Cons(hd, tl) ⇒
      let acc' = Cons(hd, acc)
      in itrev(k, tl, acc')
  and itrev_apply(k:cont, v:lst):Ans =
   case k of
      ItRev_K1 ⇒
      let k = ItRev_K2 in
      let l = v
      in itrev(k,l,Nil)
    | ItRev_K2 ⇒
      let rl = v
      in rl ; halt()
  mklst(ItRev_K1, 10)
```

Figure 8.6: Object for module B.

```
program
  type lst = Nil | Cons (int, lst)
  type cont = exn
  exception MkLst_K(int, cont)
  exception ItRev_K1
  exception ItRev_k2
  fun mklst (...):Ans =  ...
  and itrev(...):Ans = ...
  and mklst_apply (...):Ans =  ...
  and itrev_apply(...):Ans = ...
  and topApply(k:cont,v:lst):Ans =
   case k of
      MkLst_K  ⇒ mklst_apply(k,v)
    | ItRev_K1 ⇒ itrev_apply(k,v)
    | ItRev_K2 ⇒ itrev_apply(k,v)
```

Figure 8.7: Merged program.

lists and that describes the representation of continuation closures as exceptions. It imports one function `topApply` which is a global dispatch function. This function will be supplied by the linker. It will examine its continuation argument and invoke an appropriate local dispatch function. It exports the CPS version of the `mklst` function and a local dispatch function `mklst_apply`. Notice that the `mklst` function calls `topApply` to invoke its continuation parameter. If we could locally determine all the call sites of `mklst` we could replace the `topApply` function with a specialized version. Tolmach describes this and other potential optimizations that use local control flow information [TO98].

Figure 8.6 contains the code for the compiled object for module B. Notice that both objects must agree on the type signature for `topApply` and the `lst` type or linking will fail. If we changed our compilation technique so that the type of the global dispatch function `topApply` changed we would have to recompile all modules. Notice that in general we will need a different `topApply` for every distinct return type in the original source program. Therefore there needs to be an agreed-upon protocol for naming the global apply function based on the type of its argument. This is no different from the "name mangling" linker conventions found for languages that support overloading.

Figure 8.7 contains the full program created by merging the definitions. The linker in the process must build code that implements `topApply`, which calls the appropriate module specific dispatch function. At link time when all the exception constructors for program are known we can use unique integers as tags for the constructors and implement dispatch on constructors as a jump table. Given the stylized representation of each dispatch function a single global jump table can be built by merging the local dispatch functions. We could reduce this to the cost of an indirect jump by choosing a different tagging scheme and relying on the fact that after merging the dispatch function `topApply` is the only function that deconstructs exceptions, but later when the garbage collector must also deconstruct exceptions our scheme will break. So we must resort to a slightly slower approach.

Extension need for type-preserving garbage collection. Extending our approach for type-preserving garbage collection is not particularly difficult. The garbage collector needs the ability to copy every value. So just as with the `topApply` function we break the problem into small modular pieces requiring each module to implement a function that handles the types defined in that module. Figure 8.8 describes how such a scheme would be implemented.

As with our previous example we assume our compiler produces an object which has been CPS and closure converted. Additionally, the code has also been region annotated so safe points have been inserted that call our garbage collector. At each safe point we must package up the set of currently live roots. Here we use an exception packet to represent an abstract root set. Note that exceptions are allocated in a

```
object A
imports
  type lst[ρ] = Nil | Cons (int, lst[ρ]) at ρ
  type cont[ρ] = exn at ρ
  type gc_cont[ρ] = exn at ρ
  val topApply : ∀ ρ.(cont[ρ], lst[ρ]) → Ans
  val need_gc  : ∀ ρ.() → Ans
  val gc       : ∀ ρ.gc_cont[ρ] → Ans
  val gcCopy   : ∀ ρ1,ρ2.gc_cont[ρ1] → gc_cont[ρ2]
exports
  val mklst       : ∀ ρ.(cont[ρ], int[ρ]) → Ans
  val mklst_apply : (cont[ρ], lst[ρ]) → Ans
  val mklst_copy  : ∀ ρ1,ρ2.(exn[ρ1],exn[ρ2]) → Ans
  exception MkLst_K[ρ](int, cont[ρ])
  exception Ret_MkLst[ρ](cont[ρ], int)
is
  fun mklst[ρ](k:cont[ρ],n:int):Ans =
    (* safe point *)
    if need_gc[ρ]() then
      gc[ρ](gc_cont[ρ].Ret_MkLst(k,n)
    else ...
  and mklst_apply[ρ](k:cont[ρ],l:lst[ρ]):Ans = ...
  and mklst_copy[ρ1][ρ2](k:cont[ρ1]):cont[ρ2] =
    case k of
       MkLst_K(n,k) ⇒
       let n' = n in
       let k' = gcCopy[ρ1][ρ2](k)
       in exn[ρ2].MkLst_K(n',k')
     | Ret_MkLst(k,n) ⇒
       let k' = gcCopy[ρ1][ρ2](k) in
       let n' = n
       in exn[ρ2].Ret_MkLst(k',n')
```

Figure 8.8: Object for module A.

program

```
type lst[ρ] = Nil | Cons (int, lst[ρ]) at ρ
type cont[ρ] = exn[ρ]
exception MkLst_K[ρ](int, cont[ρ])
exception Ret_MkLst[ρ](cont[ρ], int)
exception ItRev_K1[ρ]
exception ItRev_k2[ρ]
exception Ret_ItRev[ρ](cont[ρ], lst[ρ], lst[ρ])
fun mklst[ρ](...):Ans =  ...
and itrev[ρ](...):Ans = ...
and mklst_apply[ρ](...):Ans =  ...
and itrev_apply[ρ](...):Ans = ...
and topApply[ρ](...):Ans = ...
and mklst_copy[ρ1][ρ2](k:cont[ρ1]):cont[ρ2] =  ...
and itrev_copy[ρ1][ρ2](k:cont[ρ1]):cont[ρ2] =  ...
and lst_copy[ρ1][ρ2](l:lst[ρ1]):lst[ρ2] =  ...
and gc[ρfrom](roots:gc_cont[ρfrom]):Ans =
 letr ρ1 in
  let roots' = gcCopy[ρfrom][ρto](roots) in
   only ρto in (* deallocate ρfrom *)
    case roots' of
      Ret_MkLst(k, n) ⇒ mklst[ρto](k, n)
    | Ret_Itrev(k, l, acc) ⇒ itrev[ρto](k, l, acc)
and gcCopy[ρ1][ρ2](k:cont[ρ1]):cont[ρ2] =
  case k of
     ItRev_K1 ⇒ itrev_copy[ρ1][ρ2](k)
   | ItRev_K2 ⇒ itrev_copy[ρ1][ρ2](k)
   | MkLst_K  ⇒ mklst_copy[ρ1][ρ2](k)
```

Figure 8.9: Merged program.

particular region. Also note that the declaration

```
exception MkLst_K[ρ](int, cont[ρ])
```

is our notation for an exception constructor parameterized by ρ and that the region variable occurring immediately to the right of the constructor name is a binding occurrence of the region variable.

The code in Figure 8.8 assumes the existence of three new functions. The `need_gc` function is a simple predicate to determine whether a collection should take place. If a collection should take place the current live roots are packaged up in an exception packet and the `gc` function is called. The `gc` function will be provided at link time as well as the `gcCopy` function, which will locally be used here to implement a copy function that handles only the locally defined types. Just as with the case for `topApply` there also needs to be a protocol for "name mangling" so that there is a canonical copy function to call for every type.

Figure 8.9 shows the merged program which includes our type-preserving garbage collector. At link time the linker must merge the various copy functions into a global copy function and emit code for the garbage collector which must examine the exception packets used to package the roots to determine what function to return to after completing a collection. This information can be encoded in a naming convention or simply as extra data passed to the linker in the object.

Problems with extensible sums approach. Since we have encoded many values by using the `exn` type as a univeral value our type-preservation guarantee says less about the overall correctness of the original program. By using hierarchical extensible sums, we can regain many of the typing guarantees we have lost by injecting values into a universal type. Also with hierarchical sums at link time we can determine which pattern matches are potentially non-exhaustive and reject programs that may fail because of non-exhaustive matches at link time. However, all these are minor modifications of the underlying approach.

The extensible sum approach to separate compilation relies on relatively simple and well understood typing constructs. The fact that the linker must merge and build a garbage collector at link time reflects the fact that in many garbage collected systems type information must be merged at link time to provide a consistent global view. It, unfortunately, does not preserve all the source language abstraction guarantees we would like, but those violations of abstraction are precisely the same violations of abstraction need by traditional tracing garbage collectors and debuggers. At worst the merging of code at link time and abstraction violations seen in the extensible sum approach merely reflect existing problems in any system that claims to support tagless garbage collection and separate compilation [JRR99]. We have simply made those problems explicit.

One significant limitation of our approach is that it does not allow for the efficient separate compilation of polymorphic functions. Separately compiling polymorphic functions efficiently has been an active area of research. For languages that do not support parametric polymorphism we believe the extensible sum approach is more than adequate.

8.3 Intensional Type Analysis

Separate compilation and polymorphism. There are a variety of solutions for compiling polymorphism. They can be broadly divided into two approaches, *heterogenous* or *homogeneous* [OW97]. In the heterogenous approach each use of a polymorphic function at a given type results in the specialization of that function at that type. In the homogeneous approach there is simply one version of the function used at all types. For example, C++ templates require heterogenous semantics. ML style polymorphism allows for either heterogenous or homogeneous approaches, but most ML systems use homogeneous encodings.

The heterogenous approach. To achieve separate compilation in a system that adopts a heterogenous approach, such as C++, requires that exported generic[3] functions must be included in the compilation environment so that clients of the function can expand and compile the code. This means generic functions may have the same problems of fragility that explicitly globally inlined functions do. Depending on the sophistication of the system, a change to a generic function may require the full recompilation of every module that depends on it. At the very least each specalized copy of the code must be recompiled. Another problem is that often a generic function is applied several times to the same type across modules. Systems that use the heterogenous approach usually provide some mechanism to share the code of generic functions typically via a cache of previously compiled functions. However, if a generic function is instantiated at many different distinct types, code blow-up is still a problem.

The chief advantage of a heterogenous approach is that it allows for a maximum amount of code optimization. Since the compiler knows at compile time precisely what type a generic function is applied to and knows precisely what the generic function is it can produce an optimized copy of the function specalized for a specific type.

[3]We use the term generic to mean polymorphic which is more consistent with C++ terminology.

The homogenous approach. Traditionally, homogeneous systems must compile polymorphic functions by choosing a universal representation for all objects passed to polymorphic functions. This has good separate compilation properties. Polymorphic functions can be compiled exactly once. The compiled code can be shared by all modules that wish to use the function at any type. The source code for the function need not be exported to other modules in the compilation environment. Changes are local to the definition of the functions and do not need to be propagated to clients of the function. However, code quality may suffer since the universal representation is not optimal for all types. For example some ML systems force floating point values to be passed by reference rather than passing the values directly in the floating point registers of the machine.

There is quite a deal of literature about how to effectively compile polymorphic functions in a homogenous fashion. The simplest approach is to choose a universal representation for all values and incur a performance penalty [MDCB91]. There are various coercion-based approaches that ameliorate the performance penalty of the uniform representation approach by using efficient specialized representations when possible and coercing to and from a universal representation only when calling unknown polymorphic functions [Ler92, Sha97].

The homogenous approach and runtime type dispatch. Finally, there has been a great deal of research into handling polymorphism via runtime type dispatch. While the previously homogeneous approaches have attempted to make polymorphic functions independent of the runtime representation of values, approaches that use runtime type dispatch solve the problem by making polymorphic functions representation-aware. Polymorphic values carry runtime type information with them so functions can determine the appropriate type-specific behavior at runtime. Again

```
val flat_i          : int → int
val flat_pii        : (int × int) → (int × int)
val flat_ppii       : ((int × int) × int) → (int × int × int)
val flat_pipi       : (int × (int × int)) → (int × int × int)
val flat_ppiipii    : ((int × int) × (int × int)) → (int × int × int × int)
val flat_pppiiipii : (((int × int) × int) × (int × int)) →
                         (int × int × int × int × int)
...
```

Figure 8.10: Flattening pairs.

```
  Typerec Append[...][...] = ...
  Typerec Flat[(α × β)] = Append[Flat[α]][Flat[β]]
        | Flat[α → β] = Flat[α] → Flat[β]
        | Flat[α] = ...

val flat : α → Flat[α]
```

Figure 8.11: Flattening pairs with ITA.

these type-dispatch techniques incur an extra cost for polymorphic functions that can be eliminated with link-time specialization. There are several approaches to runtime type dispatch. Intensional Type Analysis (ITA) [HM95, Wei01] is perhaps the most sophisticated. The most important feature of ITA is that at compile time one can verify the type correctness of code that performs runtime type dispatch. There are various flavors of ITA that lift restrictions found in the original systems [CWM98, CW99, TSS00, STS00].

Assigning types to type-dispatching functions. Another important feature of ITA is that it allows the expression of typing constraints not possible with normal parametric polymorphism by providing for inductive reasoning at the level of types. Such reasoning lets us encode certain "metatheorems" about types that could not be expressed directly before.

One such metatheorem is the fact that a set of nested pairs can be converted into a single flat tuple. One could attempt to encode this metatheorem in our language by exhaustively writting down a set of functions. (See Figure 8.10) We cannot ever hope to complete such a list since there are an countably infinite set of possible types. However, a very simple inductive proof on the structure of types can show this fact. It is perhaps not too surprising that in a language that allows for runtime type dispatch we can write a function that can generically flatten an arbitrary series of nested pairs into a flat tuple. With ITA not only can we write such a function but we can directly express its type, which is a bit surprising since as one can see by inspecting Figure 8.10 the result type of each function is related to the argument type of the function in a rather complex way.

In ITA such relationships are represented as primitive recursive functions over the structure of primitive types. In Figure 8.11 the function `Flat` defines an inductive function from primitive types to primitive types that performs case analysis on the structure of the type to compute the flatten version of an arbitrary type. It calls an auxiliary inductively defined function (Append) from types to types. Given this type function we can now assign a type to a flatten function (`flat`) that takes a value of any type (α) and returns a value whose type is the result of applying our type function to the input type of our function $\texttt{Flat}[\alpha]$. We do not wish to discuss all the formalities of ITA in detail and refer the reader to the literature for a more formal treatment [HM95, TSS00, Wei01]. The simple intuition that ITA allows for the definition of types that manipulate types will be sufficient for our discussions.

Just as the type $\mathsf{int} \to \mathsf{int}$ is an abstract description of a function that takes an integer and returns an integer, the type $\alpha \to \texttt{Flat}[\alpha]$ abstractly describes the behavior of a function that takes a value and produces another value whose type structure is related to the type structure of its argument in a predetermined way. Typecheck-

```
Typerec SubstR[ρ][(α × β)]  = (SubstR[ρ][α] × SubstR[ρ][β])
      | SubstR[ρ][α → β] = (SubstR[ρ][α] → SubstR[ρ][β])
      | SubstR[ρ][α at _] = (SubstR[ρ][α]) at ρ
      | SubstR[ρ][int] = int

val deep_copy : ∀ ρ1,ρ2.SubstR[ρ1][α] → SubstR[ρ2][α]
```

Figure 8.12: Substituting region variables with ITA.

```
Typerec Id[(α × β)] = (Id[α] × Id[β])
      | Id[α → β] = Id[α] → Id[β]
      | Id[int] = int
```

Figure 8.13: Identity Typerec.

ing in systems that support ITA is a decidable problem. ITA allows polymorphic functions to exploit specialized representations and avoids the need to treat values passed to polymorphic functions in a uniform way. It allows for the definition of type transforming functions.

ITA and garbage collection. Given the expressiveness of ITA it is not surprising that it can be used implement and typecheck the tracing function of a garbage collector which simply must be able to traverse a value of any type at runtime. If we extend our ITA system to handle regions we can now properly describe the type of the copy function used by a type-preserving garbage collector.

Figure 8.12 describes the `SubstR` type constructor. The inductive definition substitutes every occurrence of every region variable with another determined by its argument. We can understand the type of the function `deep_copy` to mean that given an arbitrary object whose components are allocated in region ρ_1 return an object with the same structure but allocated in region ρ_2. The use of ITA gurantees that the function has completely copied the value out of the region.

In ITA there are many subtleties with the representation of runtime type information in particular type representations must be computed at runtime. While all the subtleties can be addressed they add a great deal of extra complexity to the system. It is also not clear what parts of an ITA system can be easily formally verified and thus remain outside of the trusted computing base. Even if we ignore all these implementation details there are also a few subtleties with using ITA to implement type-preserving garbage collectors.

Figure 8.13 describes a simple inductive definition `Id` which simply maps a type back to itself. Under the definition of type equivalence in many ITA systems the `Id` function applied to a primitive constructor behaves in a natural way. However, when applied to an uninstantiated type variable the standard type equivalence rules do not allow us to treat $\mathtt{Id}[\alpha]$ as identical to α. The difficulty is caused because ITA systems treat $\mathtt{Id}[\alpha]$ as a normal form. From this fact we can derive the following set of equations

$$\begin{array}{l}\mathtt{Id}[\alpha \times \beta] \equiv \mathtt{Id}[\alpha] \times \mathtt{Id}[\beta] \\ \mathtt{Id}[\alpha \rightarrow \beta] \equiv \mathtt{Id}[\alpha] \rightarrow \mathtt{Id}[\beta] \\ \mathtt{Id}[\mathsf{int}] \equiv \mathsf{int} \\ \mathtt{Id}[\alpha] \not\equiv \alpha\end{array}$$

So far we have only been performing type analysis on ground types. A garbage collector must be able to perform case analysis on existentially quantified types used to hide the representation of closures. Extending ITA to handle quantified types has been examined by others [CW99, TSS00, STS00]. However, even with an ITA system extended to handle quantified types we still require some extra machinery. The last equation above is essence of the following problematic equation

$$\mathtt{SubstR}[\rho_1][\mathtt{SubstR}[\rho_2][\alpha]] \not\equiv \mathtt{SubstR}[\rho_1][\alpha]$$

which prevents us from handling existentially quantified type variables needed in the copying of closures. Monnier, Saha et al [MSS01] describe a solution to this problem by stratifying the type system into a language of types and tags. Tags only contain information about the structure of a value and do not provide information about where values are allocated. They use a specialized ITA constructor $S_\rho(\tau)$ that maps tags into region-annotated types. They discuss other pragmatic details needed to use ITA for garbage collection. Their system allows for traditional separate compilation so that the linker does almost no extra work.

Although ITA is a powerful idea it requires nontrivial techniques to formalize as well as complex and delicate primitives to implement. While it violates fewer abstraction guarantees than our extensible sum approach it also violates data privacy which can be troubling when type systems are use to enforce security guarantees.

8.4 Summary

We have outlined some of the issues in understanding precisely what tradeoffs there are in various different separate compilation architectures. Realistically, a programming environment can provide different tradeoffs by choosing to delay some work until link time or make the system more fragile. We have described one separate compilation scheme to support type-preserving garbage collectors that uses extensible sums and linker support. The extensible sums approach is sufficient for compiling monomorphic languages. For polymorphic languages sophisticated type systems that support intensional type analysis are sufficient. However, intensional type analysis is not without its complexity. Both these approaches violate important data-abstraction guarantees which we would like to preserve.

Chapter 9

Conclusions and Future Work

We have shown how to use new type systems to solve old problems related to memory management. Type systems allow for explicit memory management in ways that statically guarantee the integrity of the system. These integrity guarantees are important for secure mobile code systems. As well as reducing the amount of code that must be trusted in a secure system, we have made the system more flexible since there is a smaller set of inflexible assumptions in the system design. Programmers can also be given more explicit control over memory management policy. Debugging the compiler interface to the runtime system becomes easier.

We have shown how to build efficient garbage collectors on top of simpler, less flexible memory management services. We developed type systems that guarantee the simpler services are used in ways that do not violate program integrity. Our system improves on previous approaches by using a dynamic runtime approach to deal with certain soundness problems that have previously been addressed with complex static type systems. In the process of building a copying garbage collector, certain issues such as the violation of program abstraction become explicit. Our approach suggests ways to freely mix explicit memory management techniques with automatic techniques in a safe and flexible way.

The prototype system we built has identified important performance problems, in particular the need to support forwarding pointers and stack allocation. We have shown how to extend our basic approach to solve these performance problems. As well as describing how to extend our technique to provide better separate compilation properties.

Extending to other languages. The core idea of building a garbage collector from simpler primitives is not dependent on the programming language involved. The issues we encounter in handling higher-order functions are similar to issues that must be addressed when handling object-oriented languages. If we extend our core region calculus to handle existentially quantified types and polymorphism, we can adapt our ideas to almost any programming language.

Extending to other garbage collection algorithms. At a high-level, garbage collection algorithms move objects from one abstract set to another. Particular garbage collection algorithms differ in how these abstract sets of objects are implemented. In our type-preserving collector each abstract set of objects corresponds to a region. Our technique is not dependent on any particular implementation of the region primitives.

In the past regions have been implemented as contiguous allocation arenas. If we implement regions as doubly-linked lists of objects rather than contiguous allocation arenas, we can build a "fake copying" collector [Wan89]. The "fake copying" scheme forms the basis for incremental techniques such as Baker's Treadmill [Bak92b]. We may be be able to use this observation as the basis for building safe incremental collectors.

The mark bits used in mark-sweep and mark-compact collectors can also be seen as simple set membership bits. We believe that with an appropriate implementation of the underlying region primitives, mark-sweep and mark-compact collection schemes could be implemented. There has already been work to extend our result to generational garbage collection schemes [MSS01]. However, if we integrate existing region-based techniques with our system, we may not need the full complexity of a generational garbage collection system to achieve good performance [Hal99].

Building realistic systems and proofs. We have sketched many extensions needed for a realistic system, but their still is a great deal of work to be done in order to incorporate all these ideas into a coherent system. The challenge, however, is not to build a system, but to build a *simple* system. Constructing a soundness proof for such a system will be quite challenging. Ideally we would like to use the machine checkable semantic techniques developed by for Foundational Proof-carrying Code [App01] to construct rigorous machine checkable proofs for our system.

Preserving abstraction. We have tried to emphasize, but have not fully explored, issues related to abstraction. Can we guarantee that all the source level data privacy guarantees are maintained even in the presence of a garbage collector? How can two systems that each come with their own storage management strategies interoperate? We believe these are fundamental questions that may lead to new insights in how to construct large, efficient, and robust systems.

One approach to these problems is based on dictionary passing [WB89b]. What makes this approach attractive is that it preserves many more data-privacy guarantees than the previous approaches discussed. In this approach abstract objects are compiled into self copying values, so that a type-preserving garbage collector can copy an object by simply invoking a function provided by the value. The contents of the value remain abstract to the collector. Formalizing the needed type systems for this approach is not particularly difficult. However, it is not clear what the runtime performance of such a system would be, since copying every object now involves a function call. This may degrade the performace of the system in an unacceptable way.

Appendix A

Proof of Safety Properties

Theorem 1 (Type Soundness) *If* $\vdash P_1$ wt*, then there is no stuck* P_2 *such that* $P_1 \mapsto_P^* P_2$*.*

Proof. By structural induction on derivations of $P \mapsto_P^* P'$ and Lemma 1.1 (Type Preservation of Programs) and Lemma 1.2 (Progress)

Induction Hypothesis: We need the following stronger induction hypothesis: *If* $\vdash P_1$ wt *and* $P_1 \mapsto_P^* P_2$ *then* P_2 *is not stuck and* $\vdash P_2$ wt*.*

case $\boxed{\dfrac{}{P \mapsto_P^* P}\ \textsf{mstprefl}}$

By assumption $\vdash P$ wt and by Lemma 1.2 P is not stuck.

case $\boxed{\dfrac{P_1 \mapsto_P^* P_2 \quad P_2 \mapsto_P^* P_3}{P_1 \mapsto_P^* P_3}\ \textsf{mstptrans}}$

1. $\vdash P_1$ wt	By assumption
2. $P_1 \mapsto_P^* P_2$	By assumption
3. $P_2 \mapsto_P^* P_3$	By assumption
4. $\vdash P_2$ wt	By IH with (1) and (2)
It follows that $\vdash P_3$ wt and P_3 is not stuck	By IH with (4) and (3)

case $\boxed{\dfrac{P_1 \mapsto_P P_2}{P_1 \mapsto_P^* P_2}\ \textsf{mstprds}}$

1. $\vdash P_1$ wt — By assumption
2. $P_1 \mapsto_P P_2$ — By assumption
3. $\vdash P_2$ wt — By Lemma 1.1 with (1) and (2)
4. P_2 is not stuck — By Lemma 1.2 and (3)

   It follows that $\vdash P_2$ wt and P_2 is not stuck — By (3) and (4)

Lemma 1.1 (Type Preservation of Programs) *If* $\vdash P_1$ wt *and* $P_1 \mapsto_P P_2$*, then* $\vdash P_2$ wt*.*

Proof. By case analysis of $P_1 \mapsto_P P_2$

Consdier the cases of $P_1 \mapsto_P P_2$

case $\boxed{\textsf{rdspure}\ \ R[E[e_1]] \mapsto_P R[E[e_2]] \text{ where } e_1 \mapsto_e e_2}$

1. $\vdash R[E[e_1]]$ wt — By assumption
2. $e_1 \mapsto_e e_2$ — By assumption
3. $\{\};\{\} \vdash R[E[e_1]] : \tau$ — By (1) and inversion of $\textsf{wte}$
4. $\{\},\Delta';\{\} \vdash E[e_1] : \tau$ — By Lemma 1.12 and (3)
5. $\{\},\Delta';\{\} \vdash e_1 : \tau'$ — By Lemma 1.10 and (5)
6. $\{\},\Delta';\{\} \vdash e_2 : \tau'$ — By Lemma 1.5 with (5) and (2)
7. $\{\},\Delta';\{\} \vdash E[e_2] : \tau$ — By Lemma 1.11 with (5), (6), and (4)
8. $\{\};\{\} \vdash R[E[e_2]] : \tau$ — By Lemma 1.13 with (4), (7), and (3)

   Therefore $\vdash R[E[e_2]]$ wt — By $\textsf{wte}$ and (8)

case $\boxed{\textsf{rdsletr}\ \ R[E[\textsf{letr}\ \rho\ \textsf{in}\ e]] \mapsto_P R[\textsf{letr}\ \rho\ \textsf{in}\ E[e]]}$

1. $\vdash R[E[\textsf{letr}\ \rho\ \textsf{in}\ e]]$ wt — By assumption
2. $\{\};\{\} \vdash R[E[\textsf{letr}\ \rho\ \textsf{in}\ e]] : \tau$ — By (1) and inversion of $\textsf{wte}$
3. $\{\},\Delta';\{\} \vdash E[\textsf{letr}\ \rho\ \textsf{in}\ e] : \tau$ — By Lemma 1.12 and (2)
4. $\{\},\Delta';\{\} \vdash \textsf{letr}\ \rho\ \textsf{in}\ e : \tau'$ — By Lemma 1.10 and (3)
5. $\{\},\Delta',\{\rho\};\{\} \vdash e : \tau'$ — By (4) and inversion of $\textsf{htletr}$
6. $\{\},\Delta',\{\rho\};\{\} \vdash \textsf{letr}\ \rho\ \textsf{in}\ e : \tau'$ — By (4) and weakening
7. $\{\},\Delta',\{\rho\};\{\} \vdash E[\textsf{letr}\ \rho\ \textsf{in}\ e] : \tau$ — By (3) and weakening
8. $\{\},\Delta',\{\rho\};\{\} \vdash E[e] : \tau$ — By Lemma 1.11 with (6), (5), and (7)
9. $\{\},\Delta' \vdash \tau$ wf — By (3) and Lemma 1.8

10. $\{\}, \Delta'; \{\} \vdash \mathsf{letr}\ \rho\ \mathsf{in}\ E[e] : \tau$ — By htletr with (9) and (8)
11. $\{\}; \{\} \vdash R[\mathsf{letr}\ \rho\ \mathsf{in}\ E[e]] : \tau$ — By Lemma 1.13 with (3), (10), and (2)

Therefore $\vdash R[\mathsf{letr}\ \rho\ \mathsf{in}\ E[e]]$ wt — By wte and (11)

case $\boxed{\mathsf{rdsget}\quad R[\mathsf{letr}\ \rho\ \mathsf{in}\ R'[E[\mathsf{get}[\rho](\mathsf{put}[\rho](v))]]] \mapsto_P R[\mathsf{letr}\ \rho\ \mathsf{in}\ R'[E[v]]]}$

1. $\vdash R[\mathsf{letr}\ \rho\ \mathsf{in}\ R'[E[\mathsf{get}[\rho](\mathsf{put}[\rho](v))]]]$ wt — By assumption
2. $\{\}; \{\} \vdash R[\mathsf{letr}\ \rho\ \mathsf{in}\ R'[E[\mathsf{get}[\rho](\mathsf{put}[\rho](v))]]] : \tau$ — By (1) and inversion of wte
3. $\{\}, \Delta'; \{\} \vdash E[\mathsf{get}[\rho](\mathsf{put}[\rho](v))] : \tau$ — By Lemma 1.12 and (2)
4. $\{\}, \Delta'; \{\} \vdash \mathsf{get}[\rho](\mathsf{put}[\rho](v)) : \tau'$ — By Lemma 1.10 and (3)
5. $\{\}, \Delta'; \{\} \vdash \mathsf{put}[\rho](v) : (\tau'\ \mathsf{at}\ \rho)$ — By (4) and inversion of htget
6. $\{\}, \Delta'; \{\} \vdash v : \tau'$ — By (5) and inversion of htput
7. $\{\}, \Delta'; \{\} \vdash E[v] : \tau$ — By Lemma 1.11 with (4), (6), and (3)
8. $\{\}; \{\} \vdash R[\mathsf{letr}\ \rho\ \mathsf{in}\ R'[E[v]]] : \tau$ — By Lemma 1.13 with (3), (7), and (2)

Therefore $\vdash R[\mathsf{letr}\ \rho\ \mathsf{in}\ R'[E[v]]]$ wt — By wte and (8)

case $\boxed{\mathsf{rdsonly}\quad R[E[\mathsf{only}\ \Delta''\ \mathsf{in}\ e]] \mapsto_P R'^{\Delta''}[e]}$

1. $\vdash R[E[\mathsf{only}\ \Delta''\ \mathsf{in}\ e]]$ wt — By assumption
2. $\{\}; \{\} \vdash R[E[\mathsf{only}\ \Delta''\ \mathsf{in}\ e]] : \tau$ — By (1) and inversion of wte
3. $\{\}, \Delta; \{\} \vdash E[\mathsf{only}\ \Delta''\ \mathsf{in}\ e] : \tau$ — By Lemma 1.12 with (2)
4. $\{\}, \Delta; \{\} \vdash \mathsf{only}\ \Delta''\ \mathsf{in}\ e : \tau'$ — By Lemma 1.10 with (3)
5. $\{\}, \Delta', \Delta''; \{\} \vdash \mathsf{only}\ \Delta''\ \mathsf{in}\ e : \mathsf{Ans}$ — Because by inspection of htonly $\Delta = \Delta', \Delta''$
6. $\Delta''; \{\} \vdash e : \mathsf{Ans}$ — By (5) inversion of htonly
7. $\{\}; \{\} \vdash R'^{\Delta''}[e] : \mathsf{Ans}$ — By definition of $R'^{\Delta''}$

Therefore $\vdash R'^{\Delta''}[e]$ wt — By wte with (7)

case $\boxed{\mathsf{rdshalt}\quad R[E[\mathsf{halt}^{\tau'}]] \mapsto_P \mathsf{halt}^{\tau} \text{ where } E \neq [\,]}$

1. $\vdash R[E[\mathsf{halt}^{\tau'}]]$ wt — By assumption
2. $\{\}; \{\} \vdash R[E[\mathsf{halt}^{\tau'}]] : \tau$ — By (1) and inversion of wte
3. $\{\} \vdash \tau$ wf — By Lemma 1.8 with (2)
4. $\{\}; \{\} \vdash \mathsf{halt}^{\tau} : \tau$ — By hthalt with (3)

Therefore $\vdash \mathsf{halt}^{\tau}$ wt — By wte with (4) case $\boxed{\ \mathsf{rdsfree}\ \ R[\mathsf{letr}\ \rho\ \mathsf{in}\ R'[e]] \mapsto_P R[R'[e]] \text{ where } \vdash R[R'[e]] \text{ wt}}$

Trivial since by assumption $\vdash R[R'[e]]$ wt

Lemma 1.2 (Progress) *If* $\vdash P_1$ wt*, then there exists* P_2 *such that* $P_1 \mapsto_P P_2$ *or* P_1 *is an answer, i.e.* P_1 *is not stuck.*

Proof. Because $\vdash P_1$ wt impiles $\{\};\{\} \vdash R[e]:\tau$. By Lemma 1.12 $\Delta;\{\} \vdash e:\tau$ so by case analysis on the conclusions of Lemma 1.6

Consdier the cases of Lemma 1.6

case $\boxed{e = v}$

If e is a value than P_1 is an answer.

case $\boxed{e = E[r] \text{ where } r = ((\lambda x:\tau'.e')^{\Delta'}\ v)}$

P_1 reduces to P_2 using $\mathsf{rdspure}$ with $\mathsf{rdsbetav}$.

case $\boxed{e = E[r] \text{ where } r = ((\Lambda\rho.e')[\rho'])}$

P_1 reduces to P_2 using $\mathsf{rdspure}$ with $\mathsf{rdstapp}$.

case $\boxed{e = E[r] \text{ where } r = (\mathsf{fix} f:\tau'.v')}$

P_1 reduces to P_2 using $\mathsf{rdspure}$ with rdsfix.

case $\boxed{e = E[r] \text{ where } r = \mathsf{letr}\ \rho\ \mathsf{in}\ e'}$

P_1 reduces to P_2 using $\mathsf{rdsletr}$.

case $\boxed{e = E[r] \text{ where } r = \mathsf{get}[\rho](\mathsf{put}[\rho](v))}$

P_1 reduces to P_2 using rdsget.

case $\boxed{e = E[r] \text{ where } r = \mathsf{only}\ \Delta'\ \mathsf{in}\ e'}$

P_1 reduces to P_2 using $\mathsf{rdsonly}$.

case $\boxed{e = E[r] \text{ where } r = \mathsf{halt}^{\tau'}}$

If $E \neq [\,]$ then P_1 reduces to P_2 using $\mathsf{rdshalt}$ with rdsfix. If $E = [\,]$ then P_1 is an answer.

Lemma 1.3 (Typing Under Term Subsitution) *If* $\Delta;\Gamma \vdash e:\tau$ *and* $\Delta;\Gamma,\{x:\tau\} \vdash e':\tau'$ *then* $\Delta;\Gamma \vdash e'[e/x]:\tau'$*, where* $e = v$ *or* $e = (\mathsf{fix} f:\tau''.v)$.

Proof. By structural induction on the derivations of $\Delta;\Gamma,\{x:\tau\} \vdash e':\tau'$.

Induction Hypothesis: If $\Delta;\Gamma \vdash e:\tau$ *and* $\Delta;\Gamma,\{x:\tau\} \vdash e':\tau'$ *then* $\Delta;\Gamma \vdash e'[e/x]:\tau'$, *where* $e = v$ *or* $e = (\mathsf{fix} f:\tau''.v)$.

case $\boxed{\dfrac{}{\Delta;\Gamma,\{x:\tau\} \vdash x:\tau}\ \mathsf{htvar}}$

Because $e' = x$, $\tau' = \tau$ and $x[e/x] = e$ therefore $\Delta;\Gamma \vdash e:\tau$ by assumption

case $\boxed{\dfrac{}{\Delta;\Gamma,\{x:\tau\},\{y:\tau'\} \vdash y:\tau'}\ \mathsf{htvar}}$

Because $e' = y$, $y[e/x] = y$ therefore $\Delta;\Gamma,\{y:\tau'\} \vdash y:\tau'$ by htvar

case $\boxed{\dfrac{}{\Delta;\Gamma,\{x:\tau\} \vdash \langle\rangle:\mathsf{unit}}\ \mathsf{htunit}}$

Because $e' = \langle\rangle$, $\tau' = \mathsf{unit}$, and $\langle\rangle[e/x] = \langle\rangle$ therefore $\Delta;\Gamma \vdash \langle\rangle:\mathsf{unit}$ by htunit

case $\boxed{\dfrac{\Delta' \vdash (\Gamma',\{x:\tau\})\ \text{wfenv} \quad \Delta'' \vdash \tau_1\ \text{wf} \quad \Delta'';\Gamma'',\{y:\tau_1\} \vdash e'':\tau_2}{\Delta',\Delta'';(\Gamma',\{x:\tau\}),\Gamma'' \vdash (\lambda y:\tau_1.e'')^{\Delta''}:\tau_1 \xrightarrow{\Delta''} \tau_2}\ \mathsf{htabs}}$

Because $\Gamma = \Gamma',\Gamma''$, $\Delta = \Delta',\Delta''$, $e' = (\lambda y:\tau_1.e'')^{\Delta''}$, $\tau' = \tau_1 \xrightarrow{\Delta''} \tau_2$, and $(\lambda y:\tau_1.e'')^{\Delta''}[e/x] = (\lambda y:\tau_1.e''[e/x])^{\Delta''}$

1. $\Delta'';\Gamma'',\{y:\tau_1\}\tau \vdash e'':\tau_2$ — By assumption
2. $e''[e/x] = e''$ — Because (1) implies that x is not free in e''

Therefore $\Delta',\Delta'';\Gamma',\Gamma'' \vdash (\lambda y:\tau_1.e'')^{\Delta''}:\tau_1 \xrightarrow{\Delta''} \tau_2$ — By assumption

case $\boxed{\dfrac{\Delta' \vdash \Gamma'\ \text{wfenv} \quad \Delta'' \vdash \tau_1\ \text{wf} \quad \Delta'';(\Gamma'',\{x:\tau\}),\{y:\tau_1\} \vdash e'':\tau_2}{\Delta',\Delta'';\Gamma',(\Gamma'',\{x:\tau\}) \vdash (\lambda y:\tau_1.e'')^{\Delta''}:\tau_1 \xrightarrow{\Delta''} \tau_2}\ \mathsf{htabs}}$

Because $\Gamma = \Gamma',\Gamma''$, $\Delta = \Delta',\Delta''$, $e' = (\lambda y:\tau_1.e'')^{\Delta''}$, $\tau' = \tau_1 \xrightarrow{\Delta''} \tau_2$, and $(\lambda y:\tau_1.e'')^{\Delta''}[e/x] = (\lambda y:\tau_1.e''[e/x])^{\Delta''}$

1. $\Delta',\Delta'';\Gamma',\Gamma'' \vdash e:\tau$ — By assumption
2. $\Delta' \vdash \Gamma'$ wfenv — By assumption
3. $\Delta'' \vdash \tau_1$ wf — By assumption
4. $\Delta'';\Gamma'',\{y:\tau_1\},\{x:\tau\} \vdash e'':\tau_2$ — By assumption and exchange
5. $\Delta'' \vdash \Gamma''$ wfenv — By assumption implicit in (4)

6. $\Delta''; \Gamma'' \vdash e : \tau$ — By Lemma 1.4 with (3), (2), (5), and (1) N.B. here is where we use the fact that $e = v$ or $e = (\mathsf{fix} f : \tau'.v')$
7. $\Delta''; \Gamma'', \{y : \tau_1\} \vdash e''[e/x] : \tau_2$ — By IH with (6) and (4)

   Therefore $\Delta', \Delta''; \Gamma', \Gamma'' \vdash (\lambda y : \tau_1.e''[e/x])^{\Delta''} : \tau_1 \xrightarrow{\Delta''} \tau_2$ — By htabs with (2), (3) and (7)

case $\boxed{\dfrac{\Delta', \Delta''; \Gamma, \{x : \tau\} \vdash e_1 : \tau_1 \xrightarrow{\Delta''} \tau_2 \quad \Delta', \Delta''; \Gamma, \{x : \tau\} \vdash e_2 : \tau_1}{\Delta', \Delta''; \Gamma, \{x : \tau\} \vdash (e_1\ e_2) : \tau_2}\ \mathsf{htapp}}$

Because $\Delta = \Delta', \Delta''$, $e' = (e_1\ e_2)$, $\tau' = \tau_2$, and $(e_1\ e_2)[e/x] = (e_1[e/x]\ e_2[e/x])$

1. $\Delta', \Delta''; \Gamma \vdash e : \tau$ — By assumption
2. $\Delta', \Delta''; \Gamma, \{x : \tau\} \vdash e_1 : \tau_1 \xrightarrow{\Delta''} \tau_2$ — By assumption
3. $\Delta', \Delta''; \Gamma, \{x : \tau\} \vdash e_2 : \tau_1$ — By assumption
4. $\Delta', \Delta''; \Gamma \vdash e_1[e/x] : \tau_1 \xrightarrow{\Delta''} \tau_2$ — By IH with (1) and (2)
5. $\Delta', \Delta''; \Gamma \vdash e_2[e/x] : \tau_1$ — By IH with (1) and (3)

   Therefore $\Delta', \Delta''; \Gamma \vdash (e_1[e/x]\ e_2[e/x]) : \tau_2$ — By htapp with (4) and (5)

case $\boxed{\dfrac{\Delta, \{\rho\}; \Gamma, \{x : \tau\} \vdash e'' : \tau''}{\Delta; \Gamma, \{x : \tau\} \vdash (\Lambda\rho.e'') : \forall\rho.\tau''}\ \mathsf{httabs}}$

Because $e' = (\Lambda\rho.e'')$, $\tau' = \forall\rho.\tau''$, and $(\Lambda\rho.e'')[e/x] = (\Lambda\rho.e''[e/x])$

1. $\Delta; \Gamma \vdash e : \tau$ — By assumption
2. $\Delta, \{\rho\}; \Gamma, \{x : \tau\} \vdash e'' : \tau''$ — By assumption
3. $\Delta, \{\rho\}; \Gamma \vdash e''[e/x] : \tau''$ — By IH with (1) and (2)

   Therefore $\Delta; \Gamma \vdash (\Lambda\rho.e''[e/x]) : \forall\rho.\tau''$ — By httabs and (3)

case $\boxed{\dfrac{\Delta, \{\rho'\}; \Gamma, \{x : \tau\} \vdash e'' : \forall\rho.\tau''}{\Delta, \{\rho'\}; \Gamma, \{x : \tau\} \vdash (e''[\rho']) : \tau''[\rho'/\rho]}\ \mathsf{httapp}}$

Because $e' = (e''[\rho'])$, $\tau' = \tau''[\rho'/\rho]$, and $(e''[\rho'])[e/x] = (e''[e/x]\ \rho')$

1. $\Delta; \Gamma \vdash e : \tau$ — By assumption
2. $\Delta \vdash \rho'$ wf — By assumption
3. $\Delta; \Gamma, \{x : \tau\} \vdash e'' : \forall\rho.\tau''$ — By assumption
4. $\Delta; \Gamma \vdash e''[e/x] : \forall\rho.\tau''$ — By IH with (1) and (3)

Therefore $\Delta; \Gamma \vdash (e''[e/x][\rho']) : \tau''[\rho'/\rho]$ By httapp with (2) and (4)

case $\boxed{\dfrac{\Delta \vdash \tau' \text{ wf} \quad \Delta, \{\rho\}; \Gamma, \{x:\tau\} \vdash e'' : \tau'}{\Delta; \Gamma, \{x:\tau\} \vdash (\mathsf{letr}\ \rho\ \mathsf{in}\ e'') : \tau'}\ \mathsf{htletr}}$

Because $e' = \mathsf{letr}\ \rho\ \mathsf{in}\ e''$ and $\mathsf{letr}\ \rho\ \mathsf{in}\ e''[e/x] = \mathsf{letr}\ \rho\ \mathsf{in}\ e''[e/x]$

1. $\Delta; \Gamma \vdash e : \tau$ By assumption
2. $\Delta \vdash \tau' \text{ wf}$ By assumption
3. $\Delta, \{\rho\}; \Gamma, \{x:\tau\} \vdash e'' : \tau'$ By assumption
4. $\Delta, \{\rho\}; \Gamma \vdash e''[e/x] : \tau'$ By IH with (1) and (3)

Therefore $\Delta; \Gamma \vdash \mathsf{letr}\ \rho\ \mathsf{in}\ e''[e/x] : \tau'$ By letr with (2) and (4)

case $\boxed{\dfrac{\Delta', \{\rho\}; \Gamma, \{x:\tau\} \vdash e'' : \tau''}{\Delta', \{\rho\}; \Gamma, \{x:\tau\} \vdash \mathsf{put}[\rho](e'') : (\tau''\ \mathsf{at}\ \rho)}\ \mathsf{htput}}$

Because $\Delta = \Delta', \{\rho\}$, $e' = \mathsf{put}[\rho](e'')$, $\tau' = (\tau''\ \mathsf{at}\ \rho)$, and $\mathsf{put}[\rho](e'')[e/x] = \mathsf{put}[\rho](e''[e/x])$

1. $\Delta', \{\rho\}; \Gamma \vdash e : \tau$ By assumption
2. $\Delta', \{\rho\}; \Gamma, \{x:\tau\} \vdash e'' : \tau''$ By assumption
3. $\Delta', \{\rho\}; \Gamma \vdash e''[e/x] : \tau''$ By IH with (1) and (2)

Therefore $\Delta', \{\rho\}; \Gamma \vdash \mathsf{put}[\rho](e''[e/x]) : (\rho\ \mathsf{at}\ \tau'')$ By htput with (3)

case $\boxed{\dfrac{\Delta', \{\rho\}; \Gamma, \{x:\tau\} \vdash e'' : (\tau'\ \mathsf{at}\ \rho)}{\Delta', \{\rho\}; \Gamma, \{x:\tau\} \vdash \mathsf{get}[\rho](e'') : \tau'}\ \mathsf{htget}}$

Because $\Delta = \Delta', \{\rho\}$, $e' = \mathsf{get}[\rho](e'')$, $\mathsf{get}[\rho](e'')[e/x] = \mathsf{get}[\rho](e''[e/x])$

1. $\Delta', \{\rho\}; \Gamma \vdash e : \tau$ By assumption
2. $\Delta', \{\rho\}; \Gamma, \{x:\tau\} \vdash e'' : (\rho\ \mathsf{at}\ \tau')$ By assumption
3. $\Delta', \{\rho\}; \Gamma \vdash e''[e/x] : (\rho\ \mathsf{at}\ \tau')$ By IH with (1) and (2)

Therefore $\Delta', \{\rho\}; \Gamma \vdash \mathsf{get}[\rho](e''[e/x]) : \tau'$ By htget with (3)

case $\boxed{\dfrac{\Delta' \vdash \Gamma', \{x:\tau\} \text{ wfenv} \qquad \Delta''; \Gamma'' \vdash e'' : \mathsf{Ans}}{\Delta', \Delta''; \Gamma', \{x:\tau\}, \Gamma'' \vdash (\mathsf{only}\ \Delta''\ \mathsf{in}\ e'') : \mathsf{Ans}}\ \mathsf{htonly}}$

Because $\Delta = \Delta', \Delta''$, $\Gamma = \Gamma', \Gamma''$, $e' = \mathsf{only}\ \Delta''\ \mathsf{in}\ e''$ and $\tau' = \mathsf{Ans}$

1. $\Delta''; \Gamma'' \vdash e'' : \mathsf{Ans}$ By assumption
2. $e''[e/x] =$ e” Because (1) implies x is not free in e''

Therefore $\Delta', \Delta''; \Gamma', \Gamma'' \vdash \mathsf{only}\ \Delta''\ \mathsf{in}\ e'' : \mathsf{Ans}$ By assumption

case $\boxed{\dfrac{\Delta' \vdash \Gamma' \text{ wfenv} \qquad \Delta''; \Gamma'', \{x : \tau\} \vdash e'' : \mathsf{Ans}}{\Delta', \Delta''; \Gamma', \Gamma'', \{x : \tau\} \vdash (\mathsf{only}\ \Delta''\ \mathsf{in}\ e'') : \mathsf{Ans}}\ \mathsf{htonly}}$

Because $\Delta = \Delta', \Delta''$, $\Gamma = \Gamma', \Gamma''$, $e' = \mathsf{only}\ \Delta''\ \mathsf{in}\ e''$, $\mathsf{only}\ \Delta''\ \mathsf{in}\ e''[e/x] = \mathsf{only}\ \Delta''\ \mathsf{in}\ e''[e/x]$, and $\tau' = \mathsf{Ans}$

1. $\Delta', \Delta''; \Gamma', \Gamma'' \vdash e : \tau$ By assumption
2. $\Delta''; \Gamma'', \{x : \tau\} \vdash e'' : \mathsf{Ans}$ By assumption
3. $\Delta' \vdash \Gamma'$ wfenv By assumption
4. $\Delta'' \vdash \Gamma'', \{x : \tau\}$ wfenv By implicit assumption
5. $\Delta'' \vdash \tau$ wfenv By (4) and inversion of wfenvbv
6. $\Delta''; \Gamma'' \vdash e : \tau$ By Lemma 1.4 with (5), (3), (4), and (1)
7. $\Delta''; \Gamma'' \vdash e''[e/x] : \mathsf{Ans}$ By IH with (6) and (2)

   Therefore $\Delta', \Delta''; \Gamma', \Gamma'' \vdash \mathsf{only}\ \Delta''\ \mathsf{in}\ e''[e/x] : \mathsf{Ans}$ By htonly with (3) and (7)

case $\boxed{\dfrac{\Delta \vdash \tau' \text{ wf} \quad \Delta; \Gamma, \{\tau : x\}, \{f : \tau'\} \vdash v : \tau'}{\Delta; \Gamma, \{\tau : x\} \vdash (\mathsf{fix} f : \tau'.v) : \tau'}\ \mathsf{htfix}}$

Because $e' = (\mathsf{fix} f : \tau'.v)$ and $(\mathsf{fix} f : \tau'.v)[e/x] = (\mathsf{fix} f : \tau'.v[e/x])$

1. $\Delta; \Gamma \vdash e : \tau$ By assumption
2. $\Delta \vdash \tau'$ wf By assumption
3. $\Delta; \Gamma, \{f : \tau'\}, \{x : \tau\} \vdash v : \tau'$ By assumption of htfix
4. $\Delta; \Gamma, \{f : \tau'\} \vdash v[e/x] : \tau'$ By IH with (1) and (3)

   Therefore $\Delta; \Gamma \vdash (\mathsf{fix} f : \tau'.v[e/x]) : \tau'$ By htfix with (2) and (4)

case $\boxed{\dfrac{\Delta \vdash \tau' \text{ wf}}{\Delta; \Gamma, \{x : \tau\} \vdash \mathsf{halt}^{\tau'} : \tau'}\ \mathsf{hthalt}}$

Because $e' = \mathsf{halt}^{\tau'}$ and $\mathsf{halt}^{\tau'}[e/x] = \mathsf{halt}^{\tau'}$ by assumption $\Delta \vdash \tau'$ wf therefore $\Delta; \Gamma \vdash \mathsf{halt}^{\tau'} : \tau'$ by hthalt

Lemma 1.4 (Region Context Strengthening) *If* $\Delta' \vdash \tau$ wf*,* $\Delta \vdash \Gamma$ wfenv*,* $\Delta' \vdash \Gamma'$ wfenv*, and* $\Delta, \Delta'; \Gamma, \Gamma' \vdash e : \tau$ *where* $e = v$ *or* $e = (\mathsf{fix} f : \tau.v)$ *then* $\Delta'; \Gamma' \vdash e : \tau$

Proof. By induction on derivations of $\Delta, \Delta'; \Gamma, \Gamma' \vdash e : \tau$.

Induction Hypothesis: If $\Delta' \vdash \tau$ wf, $\Delta \vdash \Gamma$ wfenv, $\Delta' \vdash \Gamma'$ wfenv, and $\Delta, \Delta'; \Gamma, \Gamma' \vdash e : \tau$ where $e = v$ or e = $(\mathsf{fix} f : \tau . v)$ then $\Delta'; \Gamma' \vdash e : \tau$

case $\boxed{\dfrac{}{\Delta, \Delta'; \Gamma, \Gamma' \vdash \langle\rangle : \mathsf{unit}}\ \mathsf{htunit}}$

Trivial since $\Delta' \vdash \Gamma'$ wfenv therefore $\Delta'; \Gamma' \vdash \langle\rangle : \mathsf{unit}$ by htunit

case $\boxed{\dfrac{\Delta \vdash \Gamma' \text{ wfenv} \quad \Delta' \vdash \tau_1 \text{ wf} \quad \Delta'; \Gamma, \{x : \tau_1\} \vdash e' : \tau_2}{\Delta, \Delta'; \Gamma', \Gamma \vdash (\lambda x : \tau_1 . e')^{\Delta'} : \tau_1 \xrightarrow{\Delta'} \tau_2}\ \mathsf{htabs}}$

Because $\tau = \tau_1 \xrightarrow{\Delta'} \tau_2$,

1. $\Delta' \vdash \tau_1$ wf — By assumption
2. $\Delta'; \Gamma', \{x : \tau_1\} \vdash e' : \tau_2$ — By assumption
3. $\{\} \vdash \{\}$ wfenv — By $\mathsf{wfenvenpty}$

   Therefore $\{\}, \Delta'; \{\}, \Gamma' \vdash (\lambda x : \tau_1 . e')^{\Delta'} : \tau_1 \xrightarrow{\Delta'} \tau_2$ By htabs with (3), (1), and (2)

case $\boxed{\dfrac{\Delta, \Delta', \{\rho\}; \Gamma, \Gamma' \vdash e' : \tau'}{\Delta, \Delta'; \Gamma, \Gamma' \vdash (\Lambda \rho . e') : \forall \rho . \tau'}\ \mathsf{httabs}}$

1. $\Delta' \vdash \forall \rho . \tau'$ wf — By assumption
2. $\Delta \vdash \Gamma$ wfenv — By assumption
3. $\Delta' \vdash \Gamma'$ wfenv — By assumption
4. $\Delta, \Delta', \{\rho\}; \Gamma, \Gamma' \vdash e' : \tau'$ — By assumption
5. $\Delta', \{\rho\} \vdash \tau'$ wf — By (1) and inversion of wfall
6. $\Delta', \{\rho\} \vdash \Gamma'$ wfenv — By (3) and weakening
7. $\Delta', \{\rho\}; \Gamma' \vdash e' : \tau'$ — By IH with (5), (2), (6), and (4)

   Therefore $\Delta'; \Gamma' \vdash (\Lambda \rho . e') : \forall \rho . \tau'$ — By httabs with (7)

case $\boxed{\dfrac{\Delta, \Delta'', \{\rho\}; \Gamma, \Gamma' \vdash v'' : \tau'}{\Delta, \Delta'', \{\rho\}; \Gamma, \Gamma' \vdash \mathsf{put}[\rho](v'') : (\tau' \ \mathsf{at}\ \rho)}\ \mathsf{htput}}$

Because $\Delta' = \Delta'', \{\rho\}$ and $\tau = (\tau' \ \mathsf{at}\ \rho)$

1. $\Delta'', \{\rho\} \vdash (\tau' \ \mathsf{at}\ \rho)$ wf — By assumption
2. $\Delta \vdash \Gamma$ wfenv — By assumption

3. $\Delta'', \{\rho\} \vdash \Gamma'$ wfenv — By assumption
4. $\Delta, \Delta'', \{\rho\}; \Gamma, \Gamma' \vdash v' : \tau'$ — By assumption
5. $\Delta'', \{\rho\} \vdash \tau'$ wf — By (1) and inversion of wfat
6. $\Delta'', \{\rho\}; \Gamma' \vdash v' : \tau'$ — By IH with (5), (3), (2), and (4)

   Therefore $\Delta'', \{\rho\}; \Gamma' \vdash \mathsf{put}[\rho](v') : (\tau' \ \mathsf{at}\ \rho)$ — By htput with (6)

case $\boxed{\dfrac{\Delta, \Delta' \vdash \tau \text{ wf} \quad \Delta, \Delta'; \Gamma, \Gamma', \{f : \tau\} \vdash v' : \tau}{\Delta, \Delta'; \Gamma, \Gamma' \vdash (\mathsf{fix} f : \tau . v') : \tau}\ \mathsf{htfix}}$

1. $\Delta' \vdash \tau$ wf — By assumption
2. $\Delta \vdash \Gamma$ wfenv — By assumption
3. $\Delta' \vdash \Gamma'$ wfenv — By assumption
4. $\Delta' \vdash \Gamma', \{f : \tau\}$ wfenv — By $\mathsf{wfenvbv}$ with (3) and (1)
5. $\Delta, \Delta'; \Gamma, \Gamma', \{f : \tau\} \vdash v' : \tau$ — By assumption
6. $\Delta'; \Gamma', \{f : \tau\} \vdash v' : \tau$ — By IH with (1), (2), (4) and (5)

   Therefore $\Delta'; \Gamma' \vdash (\mathsf{fix} f : \tau . v') : \tau$ — By htfix with (1) and (6)

Lemma 1.5 (Type Preservation of Expression) *If $\Delta; \{\} \vdash e_1 : \tau$ and $e_1 \mapsto_e e_2$, then $\Delta; \{\} \vdash e_2 : \tau$.*

Proof. By case analysis of $e_1 \mapsto_e e_2$

Consdier the cases of $e_1 \mapsto_e e_2$

case $\boxed{\mathsf{rdsbetav} \quad ((\lambda x : \tau_1 . e)^{\Delta''}\ v) \mapsto_e e[v/x]}$

Because $\Delta = \Delta', \Delta''$ and $\{\} = \{\}, \{\}$

1. $\Delta', \Delta''; \{\}, \{\} \vdash ((\lambda x : \tau_1 . e)^{\Delta''}\ v) : \tau_2$ — By assumption
2. $\Delta', \Delta''; \{\}, \{\} \vdash (\lambda x : \tau_1 . e)^{\Delta''} : \tau_1 \xrightarrow{\Delta''} \tau_2$ — By (1) and inversion of htapp
3. $\Delta', \Delta''; \{\}, \{\} \vdash v : \tau_1$ — By (1) and inversion of htapp
4. $\Delta''; \{\}, \{x : \tau_1\} \vdash e : \tau_2$ — By (2) and inversion of htabs
5. $\Delta', \Delta''; \{\}, \{\}, \{x : \tau_1\} \vdash e : \tau_2$ — By (4) and weakening

   It follows that $\Delta', \Delta''; \{\}, \{\} \vdash e[v/x] : \tau_2$ — By Lemma 1.3 with (3) and (5)

case $\boxed{\mathsf{rdstapp} \quad ((\Lambda \rho . e)[\rho']) \mapsto_e e[\rho'/\rho]}$

Because $\Delta = \Delta', \{\rho'\}$

1. $\Delta', \{\rho'\}; \{\} \vdash ((\Lambda\rho.e)[\rho']) : \tau'[\rho'/\rho]$ — By assumption
2. $\Delta', \{\rho'\}; \{\} \vdash (\Lambda\rho.e) : \forall\rho.\tau'$ — By (1) and inversion of httapp
3. $\Delta', \{\rho'\}, \{\rho\}; \{\} \vdash e : \tau'$ — By (2) and inversion of httabs
4. $\Delta', \{\rho'\}; \{\}[\rho'/\rho] \vdash e[\rho'/\rho] : \tau'[\rho'/\rho]$ — By Lemma 1.9 with (2)

   It follows that $\Delta', \{\rho'\}; \{\} \vdash e[\rho'/\rho] : \tau'[\rho'/\rho]$ — Because $\{\}[\rho'/\rho] = \{\}$

case $\boxed{\mathsf{rdsfix} \quad (\mathsf{fix} f : \tau.v) \mapsto_e v[(\mathsf{fix} f : \tau.v)/f]}$

1. $\Delta; \{\} \vdash (\mathsf{fix} f : \tau.v) : \tau$ — By assumption
2. $\Delta; \{\}, \{f : \tau\} \vdash v : \tau$ — By (1) and inversion of htfix

   It follows that $\Delta; \{\} \vdash v[(\mathsf{fix} f : \tau.v)/f] : \tau$ — By Lemma 1.3 with (1) and (2)

Lemma 1.6 (Redex Decomposition) *If $\Delta; \{\} \vdash e : \tau$ then e is a value or $e = E[r]$ where r is a redux. A redux is any of the following forms:*

1. $((\lambda x : \tau'.e')^{\Delta''}\ v)$ *where* $\Delta = \Delta', \Delta''$
2. $((\Lambda\rho.e')[\tau'])$
3. $(\mathsf{fix} f : \tau'.v)$
4. $\mathsf{letr}\ \rho\ \mathsf{in}\ e'$
5. $\mathsf{get}[\rho](\mathsf{put}[\rho](e'))$
6. $\mathsf{only}\ \Delta'\ \mathsf{in}\ e'$
7. $\mathsf{halt}^{\tau'}$

Proof. By structural induction on well typed closed e

Induction Hypothesis: If $\Delta; \{\} \vdash e : \tau$ then e is a value or $e = E[r]$ where r is a redux.

case $\boxed{e = \langle\rangle}$

e is a value

case $\boxed{e = (\lambda x : \tau'.e')^{\Delta''} \text{ where } \Delta = \Delta', \Delta''}$

e is a value

case $\boxed{e = (\Lambda\rho.e')}$

e is a value

case $\boxed{e = \mathsf{put}[\rho](v)}$

e is a value

case $\boxed{e = (v_1\ v_2)}$

Since e is well typed by inversion of htapp v_1 has type $\tau_1 \xrightarrow{\Delta''} \tau_2$ where $\Delta = \Delta', \Delta''$. From Lemma 1.7 we conclude that $v_1 = (\lambda x : \tau'.e')^{\Delta''}$. Therefore $E = [\,]$ and $r = ((\lambda x{:}\tau'.e')^{\Delta''}\ v_2)$.

case $\boxed{e = (v\ e')}$

By IH $e' = E'[r']$. Therefore $E = (v\ E')$ and $r = r'$

case $\boxed{e = (e'\ e'')}$

By IH $e' = E'[r']$. Therefore $E = (E'\ e'')$ and $r = r'$

case $\boxed{e = (v[\tau'])}$

Since e is well typed by inversion of httapp v has type $\forall\rho.\tau'$. From Lemma 1.7 we conclude that $v = (\Lambda\rho.e')$. Therefore $E = [\,]$ and $r = ((\Lambda\rho.e')[\tau'])$

case $\boxed{e = (e[\tau'])}$

By IH $e = E'[r']$. Therefore $E = (E[\tau'])$ and $r = r'$

case $\boxed{e = \mathsf{letr}\ \rho\ \mathsf{in}\ e'}$

Let $E = [\,]$ and $r = \mathsf{letr}\ \rho\ \mathsf{in}\ e'$

case $\boxed{e = \mathsf{put}[\rho](e')}$

By IH $e' = E'[r']$. Therefore $E = \mathsf{put}[\rho](E')$ and $r = r'$

case $\boxed{e = \mathsf{get}[\rho](v)}$

Since e is well typed by inversion of htget v has type $(\tau'\ \mathtt{at}\ \rho)$ From 1.7 we conclude that $v = \mathsf{put}[\rho](v')$. Therefore $E = [\,]$ and $r = \mathsf{get}[\rho](\mathsf{put}[\rho](v'))$

case $\boxed{e = \mathsf{get}[\rho](e')}$

By IH $e' = E'[r']$. Therefore $E = \mathsf{get}[\rho](E')$ and $r = r'$

case $\boxed{e = \mathsf{only}\ \Delta'\ \mathsf{in}\ e'}$

Let $E = [\,]$ and $r = \mathsf{only}\ \Delta'\ \mathsf{in}\ e'$

case $\boxed{e = (\mathsf{fix} f : \tau'.v')}$

Let $E = [\,]$ and $r = (\mathsf{fix} f : \tau'.v')$

case $\boxed{e = \mathsf{halt}^{\tau}}$

Let $E = [\ ]$ and $r = \mathsf{halt}^{\tau}$

Lemma 1.7 (Canonical Forms) *If $\Delta;\Gamma \vdash v : \tau$ then one of the following must be true.*

1. $\tau = \mathsf{unit}$ *iff* $v = \langle\rangle$
2. $\tau = \tau_1 \xrightarrow{\Delta''} \tau_2$ *iff* $v = (\lambda x : \tau_1.e)^{\Delta''}$ *and* $\Delta = \Delta', \Delta''$
3. $\tau = \forall \rho.\tau'$ *iff* $v = (\Lambda\rho.e)$
4. $\tau = (\tau' \ \mathsf{at}\ \rho)$ *iff* $v = \mathsf{put}[\rho](v')$ *and* $\Delta = \Delta', \{\rho\}$

Proof. By inspection of the typing judgments for τ

case $\boxed{\tau = \mathsf{unit}}$

Follows from inversion of htunit

case $\boxed{\tau = \tau_1 \xrightarrow{\Delta''} \tau_2}$

Follows from inversion of htabs

case $\boxed{\tau = \forall \rho.\tau'}$

Follows from inversion of httabs

case $\boxed{\tau = (\tau' \ \mathsf{at}\ \rho)}$

Follows from inversion of htput

Lemma 1.8 (Typing Relation Implies Well Formedness) *If $\Delta;\Gamma \vdash e : \tau$ then $\Delta \vdash \tau$ wf.*

Proof. By structural induction on derivations of $\Delta;\Gamma \vdash e : \tau$.

Induction Hypothesis: If $\Delta;\Gamma \vdash e : \tau$ then $\Delta \vdash \tau$ wf.

case $\boxed{\dfrac{}{\Delta;\Gamma,\{x:\tau\} \vdash x:\tau}\ \mathsf{htvar}}$

$\Delta \vdash \Gamma, \{x : \tau\}$ wf by implicit assumption. Therefore $\Delta \vdash \tau$ wf by $\Delta \vdash \Gamma, \{x : \tau\}$ wf and inversion of $\mathsf{wfenvbv}$

case $\boxed{\dfrac{}{\Delta;\Gamma \vdash \langle\rangle : \mathsf{unit}}\ \mathsf{htunit}}$

$\Delta \vdash \mathsf{unit}$ wf by wfunit

case $\boxed{\dfrac{\Delta' \vdash \Gamma' \text{ wfenv} \quad \Delta'' \vdash \tau_1 \text{ wf} \quad \Delta''; \Gamma'', \{x : \tau_1\} \vdash e' : \tau_2}{\Delta', \Delta''; \Gamma', \Gamma'' \vdash (\lambda x : \tau_1 . e')^{\Delta''} : \tau_1 \xrightarrow{\Delta''} \tau_2} \textsf{htabs}}$

Because $\Delta = \Delta', \Delta''$ and $\Gamma = \Gamma', \Gamma''$

1. $\Delta'' \vdash \tau_1$ wf — By assumption
2. $\Delta''; \Gamma'', \{x : \tau_1\} \vdash e' : \tau_2$ — By assumption
3. $\Delta'' \vdash \tau_2$ wf — By IH with (2)
4. $\Delta'' \vdash \tau_1 \xrightarrow{\Delta''} \tau_2$ wf — By wfarrow with (1) and (3)

Therefore $\Delta', \Delta'' \vdash \tau_1 \xrightarrow{\Delta''} \tau_2$ wf — By weakening

case $\boxed{\dfrac{\Delta', \Delta''; \Gamma \vdash e_1 : \tau_1 \xrightarrow{\Delta''} \tau_2 \quad \Delta', \Delta''; \Gamma \vdash e_2 : \tau_1}{\Delta', \Delta''; \Gamma \vdash (e_1\ e_2) : \tau_2} \textsf{htapp}}$

Because $\Delta = \Delta', \Delta''$ and $\Gamma = \Gamma', \Gamma''$

1. $\Delta', \Delta''; \Gamma', \Gamma'' \vdash e_1 : \tau_1 \xrightarrow{\Delta'} \tau_2$ — By assumption
2. $\Delta', \Delta'' \vdash \tau_1 \xrightarrow{\Delta''} \tau_2$ wf — By IH with (1)

Therefore $\Delta', \Delta'' \vdash \tau_2$ wf — By (2) and inversion of wfarrow

case $\boxed{\dfrac{\Delta, \{\rho\}; \Gamma \vdash e' : \tau'}{\Delta; \Gamma \vdash (\Lambda \rho . e') : \forall \rho . \tau'} \textsf{httabs}}$

1. $\Delta, \{\rho\}; \Gamma \vdash e' : \tau'$ — By assumption
2. $\Delta, \{\rho\} \vdash \tau'$ wf — By IH with (1)

Therefore $\Delta \vdash \forall \rho . \tau'$ wf — By wfall with (2)

case $\boxed{\dfrac{\Delta', \{\rho\}; \Gamma \vdash \rho' : \forall e' . \tau'}{\Delta', \{\rho\}; \Gamma \vdash (\rho'[\rho]) : \tau'[\rho / e']} \textsf{httapp}}$

Because $\Delta = \Delta', \{\rho'\}$

1. $\Delta', \{\rho'\}; \Gamma \vdash e' : \forall \rho . \tau'$ — By assumption
2. $\Delta', \{\rho'\} \vdash \forall \rho . \tau'$ wf — By IH with (1)
3. $\Delta', \{\rho'\}, \{\rho\} \vdash \tau'$ wf — By (2) and inversion of wfall

Therefore $\Delta', \{\rho'\} \vdash \tau'[\rho' / \rho]$ wf — By induction on the derivations of $\Delta', \{\rho'\}, \{\rho\} \vdash \tau'$ wf and (3) case $\boxed{\dfrac{\Delta \vdash \tau \text{ wf} \quad \Delta, \{\rho\}; \Gamma \vdash e' : \tau}{\Delta; \Gamma \vdash (\mathsf{letr}\ \rho\ \mathsf{in}\ e') : \tau}\ \mathsf{htletr}}$

$\Delta \vdash \tau$ wf by assumption

case $\boxed{\dfrac{\Delta', \{\rho\}; \Gamma \vdash e' : \tau'}{\Delta', \{\rho\}; \Gamma \vdash \mathsf{put}[\rho](e') : (\tau'\ \mathsf{at}\ \rho)}\ \mathsf{htput}}$

Because $\Delta = \Delta', \{\rho\}$

1. $\Delta', \{\rho\} \vdash e' : \tau'$ By assumption
2. $\Delta', \{\rho\} \vdash \tau'$ wf By IH with (1)

   Therefore $\Delta', \{\rho\} \vdash (\tau'\ \mathsf{at}\ \rho)$ wf By wfat with (2)

case $\boxed{\dfrac{\Delta', \{\rho\}; \Gamma \vdash e' : (\tau'\ \mathsf{at}\ \rho)}{\Delta', \{\rho\}; \Gamma \vdash \mathsf{get}[\rho](e') : \tau'}\ \mathsf{htget}}$

Because $\Delta = \Delta', \{\rho\}$

1. $\Delta', \{\rho\} \vdash e' : (\tau'\ \mathsf{at}\ \rho)$ By assumption
2. $\Delta', \{\rho\} \vdash (\tau'\ \mathsf{at}\ \rho)$ wf By IH with (1)

   Therefore $\Delta', \{\rho\} \vdash \tau'$ wf By (2) and inversion of wfat

case $\boxed{\dfrac{\Delta' \vdash \Gamma' \text{ wfenv} \qquad \Delta''; \Gamma'' \vdash e' : \mathsf{Ans}}{\Delta', \Delta''; \Gamma', \Gamma'' \vdash (\mathsf{only}\ \Delta''\ \mathsf{in}\ e') : \mathsf{Ans}}\ \mathsf{htonly}}$

Because $\Delta = \Delta', \Delta''$, $\Delta', \Delta'' \vdash \mathsf{Ans}$ wf by wfAns

case $\boxed{\dfrac{\Delta \vdash \tau \text{ wf} \quad \Delta; \Gamma, \{f : \tau\} \vdash v : \tau}{\Delta; \Gamma \vdash (\mathsf{fix} f : \tau . v) : \tau}\ \mathsf{htfix}}$

$\Delta \vdash \tau$ wf by assumption

case $\boxed{\dfrac{\Delta \vdash \tau \text{ wf}}{\Delta; \Gamma \vdash \mathsf{halt}^{\tau} : \tau}\ \mathsf{hthalt}}$

$\Delta \vdash \tau$ wf by assumption

Lemma 1.9 (Typing Under Region Variable Subsitution) *If* $\Delta, \{\rho'\}, \{\rho\}; \Gamma \vdash e : \tau$ *then* $\Delta, \{\rho'\}; \Gamma[\rho'/\rho] \vdash e[\rho'/\rho] : \tau[\rho'/\rho]$.

Proof. By structural induction on the derivations of $\Delta, \{\rho'\}, \{\rho\}; \Gamma \vdash e : \tau$.

Induction Hypothesis: If $\Delta, \{\rho'\}, \{\rho\}; \Gamma \vdash e : \tau$ *then* $\Delta, \{\rho'\}; \Gamma[\rho'/\rho] \vdash e[\rho'/\rho] : \tau[\rho'/\rho]$.

Lemma 1.10 (Control Context Independence) *If* $\Delta; \Gamma \vdash E[e] : \tau$ *then* $\Delta; \Gamma \vdash e : \tau'$.

Proof. By induction on the structure of E

Induction Hypothesis: If $\Delta; \Gamma \vdash E[e] : \tau$ *then* $\Delta; \Gamma \vdash e : \tau'$.

case $\boxed{E = [\,]}$

Because $E[e] = e$ and $\tau = \tau'$ therefore $\Delta; \Gamma \vdash e : \tau$ by assumption

case $\boxed{E = (E'\ e')}$

Because $E[e] = (E'[e]\ e')$ and $\Delta = \Delta', \Delta''$

1. $\Delta', \Delta''; \Gamma \vdash (E'[e]\ e') : \tau$ — By assumption
2. $\Delta', \Delta''; \Gamma \vdash E'[e] : \tau_1 \xrightarrow{\Delta''} \tau$ — By (1) and inversion of htapp

   Therefore $\Delta', \Delta''; \Gamma \vdash e : \tau'$ — By IH and (2)

case $\boxed{E = (v\ E')}$

Because $E[e] = (v\ E'[e])$ and $\Delta = \Delta', \Delta''$

1. $\Delta', \Delta''; \Gamma \vdash (v\ E'[e]) : \tau$ — By assumption
2. $\Delta', \Delta''; \Gamma \vdash E'[e] : \tau_1$ — By (1) and inversion of htapp

   Therefore $\Delta', \Delta''; \Gamma \vdash e : \tau'$ — By IH and (2)

case $\boxed{E = (E'[\rho'])}$

Because $E[e] = (E'[e][\rho'])$

1. $\Delta; \Gamma \vdash (E'[e][\rho']) : \tau$ — By assumption
2. $\Delta; \Gamma \vdash E'[e] : \forall \rho.\tau''$ — By (1) and inversion of httapp

   Therefore $\Delta; \Gamma \vdash e : \tau'$ — By IH and (2)

case $\boxed{E = \mathsf{put}[\rho](E')}$

Because $E[e] = (\rho\ E'[e])$ and $\Delta = \Delta', \{\rho\}$

1. $\Delta', \{\rho\}; \Gamma \vdash \mathsf{put}[\rho](E'[e]) : \tau$ — By assumption
2. $\Delta', \{\rho\}; \Gamma \vdash E'[e] : \tau''$ — By (1) and inversion of htput

Therefore $\Delta', \{\rho\}; \Gamma \vdash e : \tau'$ By IH and (2)

case $\boxed{E = \mathsf{get}[\rho](E')}$

Because $E[e] = (\rho\ E'[e])$ and $\Delta = \Delta', \{\rho\}$

1. $\Delta', \{\rho\}; \Gamma \vdash \mathsf{get}[\rho](E'[e]) : \tau$ By assumption
2. $\Delta', \{\rho\}; \Gamma \vdash E'[e] : \tau''$ By (1) and inversion of htget

Therefore $\Delta', \{\rho\}; \Gamma \vdash e : \tau'$ By IH and (2)

Lemma 1.11 (Control Context Replacement) *If* $\Delta; \Gamma \vdash e_1 : \tau$, $\Delta; \Gamma \vdash e_2 : \tau$, *and* $\Delta; \Gamma \vdash E[e_1] : \tau'$ *then* $\Delta; \Gamma \vdash E[e_2] : \tau'$.

Proof. By induction on the structure of E

Induction Hypothesis: If $\Delta; \Gamma \vdash e_1 : \tau$, $\Delta; \Gamma \vdash e_2 : \tau$, *and* $\Delta; \Gamma \vdash E[e_1] : \tau'$ *then* $\Delta; \Gamma \vdash E[e_2] : \tau'$.

case $\boxed{E = [\,]}$

Because $E[e_2] = e_2$ and $\tau = \tau'$ therefore $\Delta; \Gamma \vdash e_2 : \tau$ by assumption

case $\boxed{E = (E'\ e')}$

Because $E[e_1] = (E'[e_1]\ e')$ and $\Delta = \Delta', \Delta''$

1. $\Delta', \Delta''; \Gamma \vdash e_1 : \tau$ By assumption
2. $\Delta', \Delta''; \Gamma \vdash e_2 : \tau$ By assumption
3. $\Delta', \Delta''; \Gamma \vdash (E'[e_1]\ e') : \tau'$ By assumption
4. $\Delta', \Delta''; \Gamma \vdash E'[e_1] : \tau_1 \xrightarrow{\Delta''} \tau'$ By (3) and inversion of htapp
5. $\Delta', \Delta''; \Gamma \vdash e' : \tau_1$ By (3) and inversion of htapp
6. $\Delta', \Delta''; \Gamma \vdash E'[e_2] : \tau_1 \xrightarrow{\Delta''} \tau'$ By IH with (1), (2), and (4)

Therefore $\Delta', \Delta''; \Gamma \vdash (E'[e_2]\ e') : \tau'$ By htapp with (6) and (5)

case $\boxed{E = (v\ E')}$

Because $E[e_1] = (v\ E'[e_1])$ and and $\Delta = \Delta', \Delta''$

1. $\Delta', \Delta''; \Gamma \vdash e_1 : \tau$ By assumption
2. $\Delta', \Delta''; \Gamma \vdash e_2 : \tau$ By assumption
3. $\Delta', \Delta''; \Gamma \vdash (v\ E'[e_1]) : \tau'$ By assumption
4. $\Delta', \Delta''; \Gamma \vdash v : \tau_1 \xrightarrow{\Delta''} \tau'$ By (3) and inversion of htapp

5. $\Delta', \Delta''; \Gamma \vdash E'[e_1] : \tau_1$ By (3) and inversion of htapp
6. $\Delta', \Delta''; \Gamma \vdash E'[e_2] : \tau'$ By IH with (1), (2), and (5)

   Therefore $\Delta', \Delta''; \Gamma \vdash (v\ E'[e_2]) : \tau'$ By htapp with (4) and (6)

case $\boxed{E = (E'[\rho'])}$

Because $E[e_1] = (E'[e_1][\rho'])$ and $\Delta = \Delta', \{\rho'\}$

1. $\Delta', \{\rho'\}; \Gamma \vdash e_1 : \tau$ By assumption
2. $\Delta', \{\rho'\}; \Gamma \vdash e_2 : \tau$ By assumption
3. $\Delta', \{\rho'\}; \Gamma \vdash (E'[e_1][\rho']) : \tau'$ By assumption
4. $\Delta', \{\rho'\}; \Gamma \vdash E'[e_1] : \forall \rho.\tau''$ By (3) and inversion of httapp
5. $\Delta', \{\rho'\}; \Gamma \vdash E'[e_2] : \forall \rho.\tau''$ By IH with (1), (2), and (4)

   Therefore $\Delta', \{\rho'\}; \Gamma \vdash (E'[e_2][\rho']) : \tau'$ By httapp with (5)

case $\boxed{E = \mathsf{put}[\rho](E')}$

Because $E[e_1] = \mathsf{put}[\rho](E'[e_1])$ and $\Delta = \Delta', \{\rho\}$

1. $\Delta', \{\rho\}; \Gamma \vdash e_1 : \tau$ By assumption
2. $\Delta', \{\rho\}; \Gamma \vdash e_2 : \tau$ By assumption
3. $\Delta', \{\rho\}; \Gamma \vdash \mathsf{put}[\rho](E'[e_1]) : \tau'$ By assumption
4. $\Delta', \{\rho\}; \Gamma \vdash E'[e_1] : \tau''$ By (3) and inversion of of htput
5. $\Delta', \{\rho\}; \Gamma \vdash E'[e_2] : \tau''$ By IH with (1), (2), and (4)

   Therefore $\Delta', \{\rho\}; \Gamma \vdash \mathsf{put}[\rho](E'[e_2]) : \tau'$ By htput with (5)

case $\boxed{E = \mathsf{get}[\rho](E')}$

$E[e_1] = \mathsf{get}[\rho](E'[e_1])$ and $\Delta = \Delta', \{\rho\}$

1. $\Delta', \{\rho\}; \Gamma \vdash e_1 : \tau$ By assumption
2. $\Delta', \{\rho\}; \Gamma \vdash e_2 : \tau$ By assumption
3. $\Delta', \{\rho\}; \Gamma \vdash \mathsf{get}[\rho](E'[e_1]) : \tau'$ By assumption
4. $\Delta', \{\rho\}; \Gamma \vdash E'[e_1] : \tau''$ By (3) and inversion of of htget
5. $\Delta', \{\rho\}; \Gamma \vdash E'[e_2] : \tau''$ By IH with (1), (2), and (4)

   Therefore $\Delta', \{\rho\}; \Gamma \vdash \mathsf{get}[\rho](E'[e_2]) : \tau'$ By htget with (5)

Lemma 1.12 (Region Stack Independence) *If $\Delta; \Gamma \vdash R[e] : \tau$ then there exists Δ' such that $\Delta, \Delta'; \Gamma \vdash e : \tau$.*

Proof. By induction on the structure of R

Induction Hypothesis: If $\Delta; \Gamma \vdash R[e] : \tau$ *then there exists* Δ' *such that* $\Delta, \Delta'; \Gamma \vdash e : \tau$.

case $\boxed{R = [\ ]}$

Because $R[e] = e$ and $\Delta = \Delta, \{\}$ therefore $\Delta; \Gamma \vdash e : \tau$ by assumption

case $\boxed{R = \mathsf{letr}\ \rho\ \mathsf{in}\ R'}$

Because $R[e] = \mathsf{letr}\ \rho\ \mathsf{in}\ R'[e]$

1. $\Delta; \Gamma \vdash \mathsf{letr}\ \rho\ \mathsf{in}\ R'[e] : \tau$ — By assumption
2. $\Delta, \{\rho\}; \Gamma \vdash R'[e] : \tau$ — By (1) and inversion of htletr
3. $\Delta, \{\rho\}, \Delta''; \Gamma \vdash e : \tau$ — By IH with (2)

Therefore $\Delta, \Delta'; \Gamma \vdash e : \tau$ — By (3) where $\Delta' = \{\rho\}, \Delta''$

Lemma 1.13 (Region Stack Replacement) *There exists* Δ' *such that if* $\Delta, \Delta'; \Gamma \vdash e_1 : \tau$, $\Delta, \Delta'; \Gamma \vdash e_2 : \tau$, *and* $\Delta; \Gamma \vdash R[e_1] : \tau$ *then* $\Delta; \Gamma \vdash R[e_2] : \tau$.

Proof. By induction on the structure of R

Induction Hypothesis: There exists Δ' *such that if* $\Delta, \Delta'; \Gamma \vdash e_1 : \tau$, $\Delta, \Delta'; \Gamma \vdash e_2 : \tau$, *and* $\Delta; \Gamma \vdash R[e_1] : \tau$ *then* $\Delta; \Gamma \vdash R[e_2] : \tau$.

case $\boxed{R = [\ ]}$

When $\Delta' = \{\}$ because $R[e_2] = e_2$ and $\Delta, \Delta' = \Delta$ therefore $\Delta; \Gamma \vdash e_2 : \tau$ by assumption

case $\boxed{R = \mathsf{letr}\ \rho\ \mathsf{in}\ R'}$

When $\Delta' = \{\rho\}$ because $R[e_1] = \mathsf{letr}\ \rho\ \mathsf{in}\ R'[e_1]$

1. $\Delta, \{\rho\}; \Gamma \vdash e_1 : \tau$ — By assumption
2. $\Delta, \{\rho\}; \Gamma \vdash e_2 : \tau$ — By assumption
3. $\Delta; \Gamma \vdash \mathsf{letr}\ \rho\ \mathsf{in}\ R'[e_1] : \tau$ — By assumption
4. $\Delta \vdash \tau$ wf — By (3) and inversion of htletr
5. $\Delta, \{\rho\}; \Gamma \vdash R'[e_1] : \tau$ — By (3) and inversion of htletr
6. $\Delta, \{\rho\}, \Delta''; \Gamma \vdash e_1 : \tau$ — By Lemma 1.12 with (1)
7. $\Delta, \{\rho\}, \Delta''; \Gamma \vdash e_2 : \tau$ — By Lemma 1.12 with (2)
8. $\Delta, \{\rho\}; \Gamma \vdash R'[e_2] : \tau$ — By IH with (6), (7), and (3)

Therefore $\Delta; \Gamma \vdash \mathsf{letr}\ \rho\ \mathsf{in}\ R'[e_2] : \tau$ — By htletr with (4) and (8)

Bibliography

[Abr93] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111(1–2):3–57, 1993.

[ADM98] Ole Agesen, David Detlefs, and J. Eliot B. Moss. Garbage collection and local variable type-precision and liveness in Java Virtual Machines. In PLDI98 [PLD98], pages 269–279.

[AF00] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Conference Record of the Twenty-seventh Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 243–253. ACM Press, January 2000.

[AFH94] Shail Aditya, Christine H. Flood, and James Hicks. Garbage collection for strongly-typed languages using run-time type reconstruction. In *Proc. 1994 ACM Conf. on Lisp and Functional Programming*, pages 12–23. ACM Press, 1994.

[AFL95] Alexander Aiken, Manuel Fähndrich, and Raph Levien. Better static memory management: Improving region-based analysis of higher-order languages. In PLDI95 [PLD95], pages 174–185.

[App87] Andrew W. Appel. Garbage collection can be faster than stack allocation. *IPL*, 25(4):275–279, 1987.

[App89] Andrew W. Appel. Runtime tags aren't necessary. *Lisp and Symbolic Computation*, 2:153–62, 1989.

[App92a] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, Cambridge, England, 1992.

[App92b] Andrew W. Appel. *Compiling with Continuations*, chapter 4, pages 37–44. Cambridge University Press, Cambridge, England, 1992.

[App96] Andrew W. Appel. Intensional equality ;=) for continuations. *ACM SIGPLAN Notices*, 31(2):55–57, 1996.

[App98] Andrew W. Appel. SSA is functional programming. *ACM SIGPLAN Notices*, 33(4):17–20, 1998.

[App01] Andrew W. Appel. Foundational proof-carrying code. In *Proceedings, Sixteenth Annual IEEE Symposium on Logic in Computer Science*, pages 247–258, Boston, 2001. IEEE Computer Society Press.

[AS96] Andrew W. Appel and Zhong Shao. Empirical and analytic study of stack versus heap cost for languages with closures. *Journal of Functional Programming*, 6(1):47–74, January 1996.

[AS00] Andrew W. Appel and Zhong Shao. Efficient and safe-for-space closure conversion. *ACM Transactions on Programming Languages and Systems*, 22(1):129–161, January 2000.

[BA97] Matthias Blume and Andrew W. Appel. Lambda-splitting: A higher-order approach to cross-module optimizations. In ICFP97 [ICF97], pages 112–124.

[Bak92a] Henry G. Baker. Lively linear Lisp — 'Look Ma, no garbage!'. *ACM SIGPLAN Notices*, 27(9):89–98, August 1992.

[Bak92b] Henry G. Baker. The Treadmill, real-time garbage collection without motion sickness. *ACM SIGPLAN Notices*, 27(3):66–70, March 1992.

[Bak93] Henry G. Baker. The boyer benchmark meets linear logic. *Lisp Pointers*, 6(4):3–10, October 1993.

[BBdH93] Nick Benton, Gavin Bierman, Valeria de Paiva, and Martin Hyland. A term calculus for intuitionistic linear logic. In M. Bezem and J. F. Groote, editors, *Proceedings Intl. Conf. on Typed Lambda Calculi and Applications, TLCA'93, Utrecht, The Netherlands, 16–18 March 1993*, volume 664, pages 75–90. Springer-Verlag, Berlin, 1993.

[BH98] Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae*, 33:309–338, 1998.

[BHR99] Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Region analysis and the polymorphic lambda calculus. In *Proceedings, Fourteenth Annual IEEE Symposium on Logic in Computer Science*, pages 88–97, Trento, Italy, 1999. IEEE Computer Society Press.

[Boe96] Hans-Juergen Boehm. Simple garbage-collector safety. In PLDI96 [PLD96], pages 89–98.

[BORT00] Josh Berdine, Peter O'Hearn, Uday S. Reddy, and Hayo Thielecke. Linearly used continuations. In *Second International Workshop on Types in Compilation*, pages 47–58, Bloomington, Indiana 47405-4101, December 2000.

[Bro75] F. P. Brooks, Jr. *The Mythical Man Month: Essays on Software Engineering*. Addison-Wesley Publishing Company, 1975.

[Bul] Bullet Train, a static optimizing compiler for Java. `http://www.naturalbridge.com/bullettrain.html`.

[BWD96] Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. Representing control in the presence of one-shot continuations. In PLDI96 [PLD96], pages 99–107.

[Cal01] Cristiano Calcagno. Stratified operational semantics for safety and correctness of the region calculus. In POPL01 [POP01], pages 155–165.

[Car97] Luca Cardelli. Program fragments, linking, and modularization. In POPL97 [POP97], pages 266–277.

[CG77] Douglas W. Clark and C. Cordell Green. An empirical study of list structure in Lisp. *Communications of the ACM*, 20(2):78–86, 1977.

[Che70] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–8, November 1970.

[CHT01] Cristiano Calcagno, Simon Helsen, and Peter Thiemann. Syntactic type soundness results for the region calculus. report00148, Institut f"ur Informatik, Universit"at Freiburg, February 20 2001.

[CJW00] Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. Flow-directed closure conversion for typed languages. In *European Symposium on Programming*, pages 56–71, 2000.

[CLN$^+$00] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Ken Cline, and Mark Plesko. A certifying compiler for Java. In PLDI00 [PLD00].

[Cop94] Max Copperman. Debugging optimized code without being misled. *ACM Transactions on Programming Languages and Systems*, 16(3):387–427, May 1994.

[Cra99] Karl Crary. A simple proof technique for certain parametricity results. In ICFP99 [ICF99], pages 82–89.

[Cra00] Karl Crary. Typed compilation of inclusive subtyping. In ICFP00 [ICF00], pages 68–81.

[CW99] Karl Crary and Stephanie Weirich. Flexible type analysis. In ICFP99 [ICF99], pages 233–248.

[CWM98] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. In ICFP98 [ICF98], pages 301–312.

[CWM99] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In POPL99 [POP99], pages 262–275.

[DDP00] Olivier Danvy, Belmina Dzafic, and Frank Pfenning. On proving syntactic properties of CPS programs. In Andrew Gordon and Andrew Pitts, editors, *Electronic Notes in Theoretical Computer Science*, volume 26. Elsevier Science Publishers, 2000.

[Dea96] Jeffery A. Dean. *Whole-Program Optimization of Object-Oriented Languages*. PhD thesis, University of Washington, 1996.

[DEB94] R. Kent Dybvig, David Eby, and Carl Bruggeman. Don't stop the BIBOP: Flexible and efficient storage management for dynamically-typed languages. Technical Report 400, Indiana University Computer Science Department, {dyb,deby,druggema}@cs.indiana.edu, March 1994.

[DMH92] Amer Diwan, J. Eliot B. Moss, and Richard L. Hudson. Compiler support for garbage collection in a statically typed language. In *Proceedings of SIGPLAN'92 Conference on Programming Languages Design and Implementation*, volume 27 of *ACM SIGPLAN Notices*, pages 273–282, San Francisco, CA, June 1992. ACM Press.

[Fer95] Mary F. Fernández. Simple and effective link-time optimization of Modula-3 programs. In PLDI95 [PLD95], pages 103–115.

[FGS96] William M. Farmer, Joshua D. Guttman, and Vipin Swarup. Security for mobile agents: Issies and requirements. In *Proceedings of the 19th National Information Systems Security Conference*, pages 591–597, Baltimore, MD, October 1996.

[Fil92] Andrzej Filinski. Linear continuations. In POPL92 [POP92], pages 27–38.

[Fis72] Michael J. Fischer. Lambda-calculus schemata. In *Proceedings ACM Conference on Proving Assertions about Programs*, pages 104–109, Los Cruces, 1972.

[GA98] David Gay and Alex Aiken. Memory management with explicit regions. In PLDI98 [PLD98], pages 313–323.

[GG92] Benjamin Goldberg and Michael Gloger. Polymorphic type reconstruction for garbage collection without tags. In *Conference Record of the 1992 ACM Symposium on Lisp and Functional Programming*, pages 53–65, San Francisco, CA, June 1992. ACM Press.

[GH93] John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Gle99] Neal Glew. Type dispatch for named hierarchical types. In ICFP99 [ICF99], pages 172–182.

[Gle00] Neal Glew. Object closure conversion. In Andrew Gordon and Andrew Pitts, editors, *Electronic Notes in Theoretical Computer Science*, volume 26. Elsevier Science Publishers, 2000.

[GMR+92] J. D. Guttman, L. G. Monk, J. D. Ramsdell, W. M. Farmer, and V. Swarup. A guide to VLISP, a verified programming language implementation. M 92B091, The MITRE Corporation, September 1992.

[Gon89] L. Gong. On security in capability-based systems. *ACM Operating Systems Review, SIGOPS*, 23(2):56–60, 1989.

[Gos95] J. Gosling. Java intermediate bytecodes. *ACM SIGPLAN Notices*, 30(3):111–118, March 1995.

[Hal99] Niels Hallenberg. Combining garbage collection and region inference in the ML Kit. Master's thesis, Department of Computer Science, University of Copenhagen, 1999.

[Han69] Wilfred J. Hansen. Compact list representation: Definition, garbage collection, and system implementation. *Communications of the ACM*, 12(9):499–507, 1969.

[HDB90] Robert Hieb, R. K. Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of SIGPLAN'90 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 66–77, White Plains, NY, June 1990. ACM Press.

[HM95] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Conference Record of the Twenty-second Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 130–141. ACM Press, January 1995.

[Hof00] Martin Hofmann. A type system for bounded space and functional in-place update. In *European Symposium on Programming*, pages 165–180, 2000.

[ICF97] *Proceedings of Second International Conference on Functional Programming*, Amsterdam, June 1997.

[ICF98] *Proceedings of Third International Conference on Functional Programming*, Paris, France, September 1998.

[ICF99] *Proceedings of Fourth International Conference on Functional Programming*, Baltimore, MA, September 1999.

[ICF00] *Proceedings of Fifth International Conference on Functional Programming*, Montreal, Canada, September 2000.

[Jon96] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[JRR99] Simon Peyton Jones, Norman Ramsey, and Germin Reig. `C--`: a portable assembly language that supports garbage colelction. In *International Conference on Principals and Practice of Declarative Programming*, September 1999.

[Kaf] Kaffe, a cleanroom implementation of a Java virtual machine. `http://www.kaffe.org`.

[Kat97] Kazuhiko Kato. Safe and secure execution mechanisms for mobile objects. In *Mobile Object Systems: Towards the Programmable Internet*, pages 157–176. Springer-Verlag: Heidelberg, Germany, 1997.

[Kel95] Richard A. Kelsey. A correspondence between continuation passing style and static single assignment form. *ACM SIGPLAN Notices*, 30(3):13–22, 1995.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Progamming Language*. Prentice Hall, Englewood Cliffs, 2 edition, 1988.

[Kra87] David Kranz. *ORBIT: An optimizing compiler for Scheme*. PhD thesis, Yale University, New Haven, CT, 1987.

[Laf88] Yves Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988.

[Ler92] Xavier Leroy. Unboxed objects and polymorphic typing. In POPL92 [POP92], pages 177–188.

[Lev84] Henry M. Levy. *Capability-Based Computer Systems*. Digital Press, Bedford, Massachusetts, 1984.

[Lia99] Sheng Liang. *Java Native Interface, The: Programmer's Guide and Specification*. Addison Wesley, 1999.

[LM92] Patrick Lincoln and John Mitchell. Operational aspects of linear lambda calculus. In *Proceedings 7th Annual IEEE Symp. on Logic in Computer Science, LICS'92, Santa Cruz, CA, USA, 22–25 June 1992*, pages 235–246. IEEE Computer Society Press, Los Alamitos, California, 1992.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184–195, 1960.

[MCGW98] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *Second International Workshop on Types in Compilation*, pages 95–117, Kyoto, Japan, March 1998.

[MDCB91] R. Morrison, A. Dearle, R. C. H. Connor, and A. L. Brown. An ad hoc approach to the implementation of polymorphism. *ACM Transactions on Programming Languages and Systems*, 13(3):342–371, July 1991.

[MFH95] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *Functional Programming and Computer Architecture*, San Diego, 1995.

[MLK] The ML Kit with regions version 3. `http://www.it.edu/research/mlkit/kit3/readme.html`.

[MLT] MLton, a whole program optimizing compiler for Standard ML. `http://www.sourcelight.com/MLton/`.

[MMH96] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Conference Record of the Twenty-third Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 271–283. ACM Press, January 1996.

[MR94] James S. Miller and Guillermo J. Rozas. Garbage collection is fast, but a stack is faster. Technical Report AIM-1462, MIT, Cambridge, Massachusetts, 1994.

[MSS01] Stefan Monnier, Bratin Saha, and Zhong Shao. Principled scavenging. In *Proceedings of SIGPLAN'01 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 8 – 91, Salt Lake City, Utah, June 2001. ACM Press.

[MWCG98] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *Conference Record of the Twenty-fifth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 85–97. ACM Press, January 1998.

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

[Nec97] George Necula. Proof-carrying code. In POPL97 [POP97], pages 106–119.

[Net92] Scott M. Nettles. A Larch specification of copying garbage collection. Research paper CMU-CS-92-219, School of Computer Science, Carnegie Mellon University, December 1992.

[OW97] Martin Odersky and Philip Wadler. Pizza into java: Translating theory into practice. In POPL97 [POP97], pages 146–159.

[PLD95] *Proceedings of SIGPLAN'95 Conference on Programming Languages Design and Implementation*, volume 30 of *ACM SIGPLAN Notices*, La Jolla, CA, June 1995. ACM Press.

[PLD96] *Proceedings of SIGPLAN'96 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices. ACM Press, 1996.

[PLD98] *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Montreal, Canada, June 1998. ACM Press.

[PLD00] *Proceedings of SIGPLAN'00 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Vancouver, Canada, June 2000. ACM Press.

[Plo75] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[POP89] *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices. ACM Press, January 1989.

[POP92] *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, Albuquerque, New Mexico, January 1992. ACM Press.

[POP97] *Conference Record of the Twenty-fourth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices. ACM Press, January 1997.

[POP99] *Conference Record of the Twenty-sixth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices. ACM Press, January 1999.

[POP01] *Conference Record of the Twenty-eight Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices. ACM Press, January 2001.

[PP00] Jeff Polakow and Frank Pfenning. Properties of terms in continuation passing style in an ordered logical framework. In *In Workshop on Logical Frameworks and Meta-Languages*, Santa Barbara, California, June 2000.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *The 16th International Conference on Automated Deduction*. Springer-Verlag, July 1999.

[Rey72] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, volume 2, pages 717–740, New York, 1972. ACM.

[RFG+92] J. D. Ramsdell, W. M. Farmer, J. D. Guttman, L. G. Monk, and V. Swarup. The VLISP PreScheme front end. M 92B098, The MITRE Corporation, September 1992.

[RJ00] Norman Ramsey and Simon Peyton Jones. A single intermediate language that supports multiple implementations of exceptions. In PLDI00 [PLD00], pages 285–298.

[Ros67] D. T. Ross. The AED free storage package. *Communications of the ACM*, 10(8):481–492, August 1967.

[RR96] John Reppy and Jon Riecke. Simple objects for Standard ML. In PLDI96 [PLD96], pages 171–180.

[Sab98] Amr A. Sabry. *The Formal Relationship between Direct and Continuation-Passing Style Optimizing Compilers: A Synthesis of Two Paradigms*. PhD thesis, Rice University, Houston, Texas, August 1998.

[SF98] Jonathan Sobel and Daniel P. Friedman. Recycling continuations. In ICFP98 [ICF98], pages 251–270.

[SH87] Peter Steenkiste and John Hennessy. Tags and type checking in LISP: Hardware and software approaches. In *Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pages 50–59, Palo Alto, CA, October 1987.

[Sha97] Zhong Shao. Flexible representation analysis. In ICFP97 [ICF97], pages 85–98.

[Shi91] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1991.

[SRA94] Zhong Shao, John H. Reppy, and Andrew W. Appel. Unrolling lists. In *Proceedings 1994 ACM Conference on Lisp and Functional Programming*, pages 185–195. ACM Press, 1994.

[Ste78] Guy L. Steele. Rabbit: a compiler for Scheme. Technical Report AI-TR-474, MIT, Cambridge, Massachusetts, 1978.

[STS00] Bratin Saha, Valery Trifonov, and Zhong Shao. Fully reflexive intensional type analysis in type erasure semantics. Technical Report YALEU/DCS/TR-1197, Dept. of Computer Science, Yale University, New Haven, CT, June 2000.

[SW67] H. Schorr and W. Waite. An efficient machine independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8):501–506, August 1967.

[SW92] Amitabh Srivastava and David W. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, 1(1):1–18, December 1992.

[SWM00] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In Gert Smolka, editor, *Ninth European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381, Berlin, April 2000. Springer-Verlag.

[TBE+98] Mads Tofte, Lars Birkedal, Martin Elsman, Neils Hallenberg, Tommy Højfeld Olesen, Peter Sestoft, and Peter Bertelsen. Programming with reigons in the ML Kit (for version 3). Technical Report DIKU-TR-98/25, University of Copenhagen, December 1998.

[TD94] David Tarditi and Amer Diwan. Measuring the cost of storage management. Technical Report CMU-CS-94-201, Carnegie Mellon University, 1994. Accepted for publication in Lisp and Symbolic Computation.

[TH01] Mades Tofte and Niels Hallenberg. Region-based memory management in perspective. Notes distributed at SPACE2001, January 2001.

[Tho84] Ken Thompson. Reflections on trusting trust. *CACM*, 27(8):761–763, August 1984. appeared also in "ACM Turing Award Lectures: The First Twenty Years 1965-1985" (ACM Press) and Peter J. Denning (ed.): "Computers under attack: intruders, worms and viruses" (ACM Press, New York 1990, 97-104).

[TJ92] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 162–173, Los Alamitos, California, 1992. IEEE Computer Society Press.

[TMC+96] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In PLDI96 [PLD96], pages 181–192.

[TO98] Andrew Tolmach and Dino P. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, July 1998.

[Tol94] Andrew Tolmach. Tag-free garbage collection using explicit type parameters. In *Proc. 1994 ACM Conf. on Lisp and Functional Programming*, pages 1–11, New York, 1994. ACM Press.

[TSS+97] David L. Tennenhouse, Jonathan M. Smith, W David Sincoskie, David J. Wehterall, and Gary J. Minden. A survey of actice network research. *IEEE Communication Magazine*, 35(1):80–86, January 1997.

[TSS00] Valery Trifonov, Bratin Saha, and Zhong Shao. Fully reflexive intensional type analysis. In ICFP00 [ICF00], pages 82–93.

[TT94] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 188–201. ACM Press, January 1994.

[Tuc97] Allen B. Tucker, editor. *The Computer Science and Engineering Handbook*, chapter 103, pages 2208–2236. CRC Press, 1997.

[TW99] David N. Turner and Philip Wadler. Operational interpretations of linear logic. *Theoretical Computer Science*, 227:231–248, 1999. Special issue on linear logic.

[Ull98] Jeffrey D. Ullman. *Elements of ML Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, second edition, 1998.

[Vei76] G. Veillon. Transformations de programmes recursifs. *R.A.I.R.O. Informatique*, 10(9):7–20, September 1976.

[WA01] Daniel C. Wang and Andrew W. Appel. Type-preserving garbage collectors. In POPL01 [POP01], pages 166–178.

[Wad90] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods*, Sea of Galilee, Israel, April 1990. North Holland. IFIP TC 2 Working Conference.

[Wan89] Thomas Wang. The MM garbage collector for C++. Master's thesis, California State Polytechnic University, October 1989.

[WB89a] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In POPL89 [POP89], pages 60–76.

[WB89b] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In POPL89 [POP89], pages 60–76.

[Wei01] Stephanie Weirich. Encoding intensional type analysis. In *European Symposium on Programming*, Genova, Italy, April 2001.

[Wen90] E. P. Wentworth. Pitfalls of conservative garbage collection. *Software Practice and Experience*, 20(7):719–727, 1990.

[WF94] Wright and Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jacques Cohen, editors, *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, University of Texas, 16–18 September 1992. Springer-Verlag.

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In Henry Baker, editor, *Proceedings of International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, Kinross, Scotland, September 1995. Springer-Verlag.

[WM00] David Walker and Greg Morrisett. Alias types for recursive data structures (extended version). Technical Report TR2000-1787, Cornell University, March 2000.

[XP99] Howgwei Xi and Frank Pfenning. Dependent types in practical programming. In POPL99 [POP99], pages 214–227.

[Yat99] Bennett Norton Yates. A type-and-effect system for encapsulating memory in Java. Master's thesis, Department of Computer Science and Information Science, University of Oregon, 1999.

[ZG00] Silvano Dal Zilio and Andrew D. Gordon. Region analysis and a pi-calculus with groups. In *Proceedings of the Twenty-fifth International Symposium on Mathematical Foundations of Computer Science*, volume 1893. Springer-Verlag, August 2000.

[ZGM99] Steve Zdancewic, Dan Grossman, and Greg Morrisett. Principals in programming languages: A syntactic proof technique. In ICFP99 [ICF99], pages 197–207.