Better Static Memory Management: Improving Region-Based Analysis of Higher-Order Languages

(Extended Abstract)

Alexander Aiken[*] Manuel Fähndrich Raph Levien[†]

Computer Science Division
University of California, Berkeley[‡]

Abstract

Static memory management replaces runtime garbage collection with compile-time annotations that make all memory allocation and deallocation explicit in a program. We improve upon the Tofte/Talpin region-based scheme for compile-time memory management [TT94]. In the Tofte/Talpin approach, all values, including closures, are stored in regions. Region lifetimes coincide with lexical scope, thus forming a runtime stack of regions and eliminating the need for garbage collection. We relax the requirement that region lifetimes be lexical. Rather, regions are allocated late and deallocated as early as possible by explicit memory operations. The placement of allocation and deallocation annotations is determined by solving a system of constraints that expresses all possible annotations. Experiments show that our approach reduces memory requirements significantly, in some cases asymptotically.

1 Introduction

In a recent paper, Tofte and Talpin propose a novel method for memory management in typed, higher-order languages [TT94]. In their scheme, runtime memory is partitioned into *regions*. Every computed value is stored in some region. Regions themselves are allocated and deallocated according to a stack discipline akin to the standard implementation of activation records in procedural languages and similar to that of [RM88]. The assignment of values to regions is decided statically by the compiler and the program is annotated to include operations for managing regions. Thus, there is no need for a garbage collector—all memory allocation and deallocation is statically specified in the program.

The system in [TT94] makes surprisingly economical use of memory. However, it is usually possible to do significantly better and in some cases dramatically better than the Tofte/Talpin algorithm. In this paper, we present an extension to the Tofte/Talpin system that removes the restriction that regions be stack allocated, so that regions may have arbitrarily overlapping extent. Preliminary experimental results support our approach. Programs transformed using our analysis typically use significantly less (by a constant factor) memory than the same program annotated with the Tofte/Talpin system alone. We have also found that for some common programming idioms the improvement in memory usage is asymptotic. The memory behavior is never worse than the memory behavior of the same program annotated using the Tofte/Talpin algorithm.

It is an open question to what degree static decisions about memory management are an effective substitute for runtime garbage collection. Our results do not resolve this question, but we do show that static memory management can be significantly better than previously demonstrated. Much previous work has focussed on reducing, rather than eliminating, garbage collection [HJ90, Deu90]. The primary motivation for static memory management put forth in [TT94] is to reduce the amount of memory required to run general functional programs efficiently. Two other applications interest us. First, the pauses in execution caused by garbage collection pose a difficulty for programs with real-time constraints. While there has been substantial work on real-time garbage collection [DLM$^+$78, NO93], we find the simpler model of having no garbage collector at all appealing and worth investigation. Second, most programs written today are not written in garbage-collected applicative languages, but rather in procedural languages with programmer-specified memory management. A serious barrier to using applicative languages is that they do not always interoperate easily with procedural languages. The interoperability problem is due in part to the gap between the two memory management models. We expect that implementations of applicative languages with static memory management would make writing components

[*]Supported in part by an NSF NYI award.

[†]Supported by an NSF graduate research fellowship.

[‡]Authors' address: Computer Science Division, Soda Hall, University of California, Berkeley, CA 94720-1776.
Email: `{aiken,manuel,raph}@cs.berkeley.edu`
URL: `http://kiwi.cs.berkeley.edu/~nogc`

of large systems in applicative languages more attractive.

Our approach to static memory management is best illustrated with an example. We present the example informally; the formal presentation begins in Section 2. Consider the following simple program, taken from [TT94]:

$$\texttt{(let}\ x = \texttt{(2,3) in}\ \lambda y.\texttt{(fst}\ x, y\texttt{) end) 5}$$

The source language is a conventional typed, call-by-value lambda calculus; it is essentially the applicative subset of ML [MTH90]. The annotated program produced by the Tofte/Talpin system is:

Example 1.1
```
letregion $\rho_4, \rho_5$ in
    letregion $\rho_6$ in
        let $x$ = (2@$\rho_2$,3@$\rho_6$)@$\rho_4$ in
            ($\lambda y$.(fst $x, y$)@$\rho_1$)@$\rho_5$
        end
    end 5@$\rho_3$
end
```

There are two kinds of annotations: `letregion` ρ `in` e binds a new region to the region variable ρ. The scope of ρ is the expression e. Upon completion of the evaluation of e, the region bound to ρ and any values it contains are deallocated. The expression $e@\rho$ evaluates e and writes the result in ρ. All values—including integers, pairs, and closures—are stored in some region.[1] Note that certain region variables appear free in the expression; they refer to regions needed to hold the result of evaluation. The regions introduced by a `letregion` are local to the computation and are deallocated when evaluation of the `letregion` completes.

The solid lines in Figure 1c depict the lifetimes of regions with respect to the sequence of memory accesses performed by the annotated program above. Operationally, evaluating the function application first allocates the regions bound to ρ_4, ρ_5, and ρ_6. Next the integer `2` is stored (in the region bound to ρ_2), then the integer `3` (in ρ_6), the pair x (in ρ_4), and the closure $\lambda y. \ldots$ (in ρ_5). At this point, the inner `letregion` is complete and ρ_6 is deallocated. Evaluating the argument of the function application stores the integer `5` (in ρ_3). Finally, evaluating the application itself requires retrieving the closure (from ρ_5), retrieving the first component of x (from ρ_4), and constructing another pair (in ρ_1).

In the Tofte/Talpin system, the `letregion` construct combines the introduction of a region, region allocation, and region deallocation. In our system, we separate these three operations. For us, `letregion` just introduces a new, lexically scoped, region variable bound to an unallocated region. The operation `alloc_before` ρ e allocates space for the region bound to ρ before evaluating e, and the operation `free_after` ρ e deallocates space assigned to the region bound to ρ after evaluating e. The operations `free_before` and `alloc_after` are defined analogously.

The problem we address is: given a program annotated by the Tofte/Talpin system, produce a *completion* that adds allocation/deallocation operations on region variables. Figure 1a shows the most conservative legal completion of the example program. Each region is allocated immediately upon entering and deallocated just before exiting the region's scope; this program has the same region lifetimes as the Tofte/Talpin annotated program above. The `alloc_before` ρ and `free_after` ρ annotations may be attached to any program point in the scope of ρ, so long as the region bound to ρ actually is allocated where it is used. In addition, for correctness it is important that a region be allocated only once and deallocated only once during its lifetime. Within these parameters there are many legal completions. Figure 1b shows the completion computed by our algorithm. There is one new operation `free_app`. In an application $e_1\ e_2$, the region containing the closure can be freed after both e_1 and e_2 are evaluated but before the function body itself is evaluated. This point is not immediately before or after the evaluation of any expression, so we introduce `free_app` to denote freeing a region at this point.

The dotted lines in Figure 1c depict the lifetimes of regions under our completion. This particular completion is optimal—space for a value is allocated at the last possible moment (immediately prior to the first use of the region) and deallocated at the earliest possible moment (immediately after the last use of the region). For example, the value `3`$@\rho_6$ is deallocated immediately after it is created, which is correct because there are no uses of the value. While an optimal completion does not always exist, this example does illustrate some characteristic features of our algorithm. For example, space for a pair ideally is allocated only after both components of the pair have been evaluated—the last point before the pair itself is constructed. Similarly, at the last use of a function its closure is deallocated after the closure has been fetched from memory but before the function body is evaluated. These properties are not special cases—they follow from the general approach we adopt.

For any given program, our method produces a system of constraints characterizing all completions. Each solution of the constraints corresponds to a valid completion. The constraints rely on knowledge of the sequence of reads and writes to regions. Thus, the constraints are defined over the program's control flow. However, because of higher order functions, inferring control flow from the syntactic form of the program is difficult. A well-known solution to this problem is closure analysis [Ses92], which gives a useful approximation to the set of possible closures at every application.

Our algorithm consists of two phases. We begin with the Tofte/Talpin annotation of a program. In the first phase, an extended closure analysis computes the set of closures that may result from evaluating each expression in every possible *region environment* (Section 3). In the second phase, local constraints are generated from the *(expression, region environment)* pairs (Section 4). These constraints express facts about regions that must hold at a given program point in a given context. For example, if an expression e accesses a region z, there are constraints such as "z must be allocated sometime before the evaluation of e" and "z must be deallocated sometime after the evaluation of e."

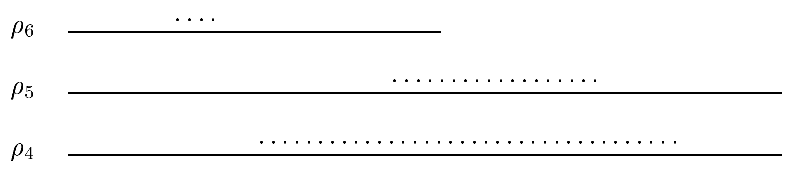
[1] We assume small integers are boxed to make the presentation simple and uniform. In practice, small integers can be unboxed.

```
letregion $\rho_4, \rho_5$ in
    alloc_before $\rho_4$ free_after $\rho_4$ alloc_before $\rho_5$ free_after $\rho_5$
    letregion $\rho_6$ in
        alloc_before $\rho_6$ free_after $\rho_6$
        let $x$ = (2@$\rho_2$,3@$\rho_6$)@$\rho_4$ in
            ($\lambda y$.(fst $x, y$)@$\rho_1$)@$\rho_5$
        end
    end 5@$\rho_3$
end
```

(a) The example with explicit region allocation/deallocation operations.

```
letregion $\rho_4, \rho_5$ in
    free_app $\rho_5$
        letregion $\rho_6$ in
            let $x$ = (2@$\rho_2$,alloc_after $\rho_4$ alloc_before $\rho_6$ free_after $\rho_6$ 3@$\rho_6$)@$\rho_4$ in
                alloc_before $\rho_5$ ($\lambda y$.(free_after $\rho_4$ fst $x, y$)@$\rho_1$)@$\rho_5$
            end
        end 5@$\rho_3$
end
```

(b) The example with the optimal explicit region allocation/deallocation operations.



operation	write	write	write	write	write	read	read	write
value	2	3	x (a pair)	λy	5	λy	x	pair
region	ρ_2	ρ_6	ρ_4	ρ_5	ρ_3	ρ_5	ρ_4	ρ_1

———————— region lifetimes in program (a)
. . . . region lifetimes in program (b)

(c) Graph of region lifetimes with respect to the sequence of memory operations.

Figure 1: An example comparing stack vs. non-stack region allocation.

A novel aspect of our algorithm arises in the resolution of the constraints. As one might expect, solving the constraints yields an annotation of the program, but finding a solution is not straightforward. Some program points will be, in fact, under constrained. For example, in the program in Figure 1, the initial constraints specify that the region bound to ρ_5 must be allocated when $\lambda y \ldots$ is evaluated, but there is no constraint on the status of the region bound to ρ_5 prior to the evaluation of λy. That is, we must choose whether ρ_5 is allocated prior to the evaluation of λy or not—there are legal completions in both scenarios. Given the choice, we prefer that ρ_5 not be allocated earlier to minimize memory usage; this choice forces the completion `alloc_before` ρ_5 $\lambda y \ldots$. Adding the constraint that ρ_5 is unallocated prior to evaluation of λy affects the legal completion in other parts of the program. Thus, our algorithm alternates between finding "choice points" and constraint resolution until a completion has been constructed. This structure is unusual among program analysis algorithms and may be of independent interest.

An outline of the soundness proof is presented in Section 5. Detailed discussion and measurements of the behavior of our algorithm are presented in Section 6. Section 7 concludes with a discussion of practical issues.

2 Definitions

The input to our analysis is a program annotated by the Tofte/Talpin algorithm. The syntax of such programs is

$$\begin{array}{rcl} e & ::= & x \mid \lambda x.e@\rho \mid e_1\ e_2 \mid f[\vec{\rho}]@\rho \\ & \mid & \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end} \\ & \mid & \texttt{letrec}\ f[\vec{\rho}](x)@\rho = e_1\ \texttt{in}\ e_2\ \texttt{end} \\ & \mid & \texttt{letregion}\ \rho\ \texttt{in}\ e\ \texttt{end} \end{array}$$

Other operations, such as pairing and selection, are omitted for brevity. The language includes *region polymorphic* functions—functions that take regions as arguments. Region polymorphism allows each invocation of a recursive function to operate on different regions, which is important for achieving good separation of region lifetimes [TT94].

The Tofte/Talpin annotations are derived using a nonstandard type system. A type is a pair (τ, ρ), where τ indicates the kind of value (integer, function, etc.) and ρ refers to the region where values of the type are stored. The determination of region scope is made by tracking the *effect* of an expression, which is the set of regions the expression may read or write when evaluated. Types are defined by the following grammar:

$$\begin{array}{rcl} \tau & ::= & \text{int} \mid \mu \overset{\epsilon.\varphi}{\to} \mu \\ \mu & ::= & (\tau, \rho) \end{array}$$

An object of the form $\epsilon.\varphi$ is called an *arrow effect:* it is the effect of applying a function of type $\mu \overset{\epsilon.\varphi}{\to} \mu'$. The "$\epsilon$." is an *effect variable* which names the effect and is useful for type inference purposes.

To the base language we add operations to allocate and free regions:

$$\begin{array}{rclcl} e & ::= & \cdots & & \\ & \mid & \texttt{alloc\_before}\ \rho\ e & \mid & \texttt{alloc\_after}\ \rho\ e \\ & \mid & \texttt{free\_before}\ \rho\ e & \mid & \texttt{free\_after}\ \rho\ e \\ & \mid & \texttt{free\_app}\ \rho\ e_1\ e_2 & & \end{array}$$

The operational semantics of this language derives facts of the form

$$s, n, r \vdash e \to a, s'$$

which is read "in store s, environment n, and region environment r the expression e evaluates to store address a and new store s'." The structures of the operational semantics are:

$$\begin{array}{rcl} \text{RegionState} & = & \text{unallocated} + \text{deallocated} + \\ & & (\text{Offset} \overset{\text{fin}}{\to} \text{Clos} + \text{RegClos}) \\ \text{Store} & = & \text{Region} \overset{\text{fin}}{\to} \text{RegionState} \\ \text{Clos} & = & \text{Lam} \times \text{Env} \times \text{RegEnv} \\ \text{RegClos} & = & \text{RegionVar}^* \times \text{Lam} \times \text{Env} \times \text{RegEnv} \\ \text{Env} & = & \text{Var} \overset{\text{fin}}{\to} \text{Region} \times \text{Offset} \\ \text{RegEnv} & = & \text{RegionVar} \overset{\text{fin}}{\to} \text{Region} \end{array}$$

A store contains a set of regions $z_1, z_2, \ldots$. A region has one of three states: it is *unallocated*, *deallocated*, or it is *allocated*, in which case it is a function from integer offsets $o_1, o_2, \ldots$ within the region to storable values. A region can hold values only if it is allocated. Note that regions are not of fixed size—a region potentially holds any number of values. A *region environment* maps region variables $\rho_1, \rho_2, \ldots$ to regions. A vector of region variables is written $\vec{\rho}$.

In this small language, the only storable values are ordinary closures and region polymorphic closures. Ordinary closures have the form $\langle \lambda x.e@\rho, n, r\rangle$, where $\lambda x.e@\rho$ is the function, n is the closure's environment, and r is the closure's region environment. A region polymorphic closure has additional region parameters. The set of $\lambda x.e@\rho$ terms is Lam; the $@\rho$ annotation is elided when it is clear from context or unneeded.

Figure 2 gives the operational semantics. An *address* is a *(region, offset)* pair. Given an address $a = (z, o)$, we generally abbreviate $s(z)(o)$ by $s(a)$. All maps (e.g., environment, store, etc.) in the semantics are finite. The set $Dom(f)$ is the domain of map f. The map $f[x \leftarrow v]$ is map f modified at argument x to give v. Finally, $f|_X$ is map f with the domain restricted to X.

The semantics in Figure 2 enforces two important restrictions on regions. First, the semantics forbids operations on a region that is not allocated; reads or writes to unallocated/deallocated regions are errors. Second, every region introduced by a `letregion` progresses through three stages: it is initially unallocated, then allocated, and finally deallocated. For example, the [*ALLOCBEFORE*] rule allocates a previously unallocated region before the evaluation of an expression. Only one representative of each of the allocation and deallocation operations is presented in the semantics; the others are defined analogously.

An example illustrates the [*LETREC*] and [*REGAPP*] rules. Consider the following program:

Example 2.1

```
letregion $\rho_1, \rho_2, \rho_3$ in
    let $i = 1@\rho_1, j = 2@\rho_2$ in
        letrec $f[\rho_5, \rho_6]$($k : (\text{int}, \rho_5)$) $@\rho_3 =$
            letregion $\rho_7$ in
                ($k +$ (1$@\rho_7$)) $@\rho_6$
            end
        in
            ($f[\rho_1, \rho_4]@\rho_0\ i + f[\rho_2, \rho_4]@\rho_0\ j$)$@\rho_4$
        end
    end
end
```

In this program, nested `let` and `letregion` constructs are abbreviated. To make the example interesting, we use constructs outside the minimal language presented above. The expression $i@\rho$ stores integer i in the region bound to ρ; the expression $(e_1 + e_2)\ @\rho$ stores the sum of e_1 and e_2 in the region bound to ρ. Region allocation/deallocation operations are omitted for clarity.

In Example 2.1, `letrec` $f[\rho_5, \rho_6]$(`k`) $@\rho_3 = \ldots$ stores a new region polymorphic closure at a fresh address a in the region bound to ρ_3. Next, the expression $(f[\rho_1, \rho_4]\ @\rho_0\ i + f[\rho_2, \rho_4]\ @\rho_0\ j)\ @\rho_4$ is evaluated in an environment n where $n(f) = a$. A region application $f[\rho_1, \rho_4]\ @\rho_0$ creates an ordinary closure (stored at the region bound to ρ_0) with formal region parameters ρ_5 and ρ_6 bound to the region values of ρ_1 and ρ_4 respectively. When applied to the argument i (in ρ_1), the result is stored in ρ_4. The closure resulting from $f[\rho_2, \rho_4]$ expects its argument in ρ_2 instead. Region polymorphism allows the function f to take arguments and return results in different regions in different contexts.

$$\frac{n(x) = a}{s, n, r \vdash x \rightarrow a, s} \qquad [\mathit{VAR}]$$

$$\frac{\begin{array}{l} n(f) = a \quad s(a) = \langle \vec{\rho}, \lambda x.e, n_0, r_0 \rangle \\ o \notin \mathit{Dom}(s(r(\rho'))) \\ a' = (r(\rho'), o) \\ c = \langle \lambda x.e, n_0, r_0[\vec{\rho} \leftarrow r(\vec{\rho}\,')] \rangle \end{array}}{s, n, r \vdash f[\vec{\rho}\,'] \,@\, \rho' \rightarrow a',\ s[a' \leftarrow c]} \qquad [\mathit{REGAPP}]$$

$$\frac{o \notin \mathit{Dom}(s(r(\rho))) \quad a = (r(\rho), o)}{s, n, r \vdash \lambda x.e \,@\, \rho \rightarrow a, s[a \leftarrow \langle \lambda x.e, n, r \rangle]} \qquad [\mathit{ABS}]$$

$$\frac{\begin{array}{l} s, n, r \vdash e_1 \rightarrow a_1, s_1 \\ s_1, n, r \vdash e_2 \rightarrow a_2, s_2 \\ s_2(a_1) = \langle \lambda x.e, n_0, r_0 \rangle \\ s_2, n_0[x \leftarrow a_2], r_0 \vdash e \rightarrow a_3, s_3 \end{array}}{s, n, r \vdash e_1\ e_2 \rightarrow a_3, s_3} \qquad [\mathit{APP}]$$

$$\frac{\begin{array}{l} s, n, r \vdash e_1 \rightarrow a_1, s_1 \\ s_1, n[x \leftarrow a_1], r \vdash e_2 \rightarrow a_2, s_2 \end{array}}{s, n, r \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end} \rightarrow a_2, s_2} \qquad [\mathit{LET}]$$

$$\frac{\begin{array}{l} o \notin \mathit{Dom}(s(r(\rho))) \\ n' = n[f \leftarrow (r(\rho), o)] \\ s[(r(\rho), o) \leftarrow \langle \vec{\rho}, \lambda x.e_1, n', r \rangle], n', r \vdash e_2 \rightarrow a, s' \end{array}}{s, n, r \vdash \texttt{letrec}\ f[\vec{\rho}](x) \,@\, \rho = e_1\ \texttt{in}\ e_2 \rightarrow a, s'} \qquad [\mathit{LETREC}]$$

$$\frac{\begin{array}{l} z \notin \mathit{Dom}(s) \\ s_0 = s[z \leftarrow \mathit{unallocated}] \\ s_0, n, r[\rho \leftarrow z] \vdash e \rightarrow a_1, s_1 \\ s_1(z) = \mathit{deallocated} \end{array}}{s, n, r \vdash \texttt{letregion}\ \rho\ \texttt{in}\ e \rightarrow a_1, s_1|_{\mathit{Dom}(s)}} \qquad [\mathit{LETREGION}]$$

$$\frac{\begin{array}{l} r(\rho) = z \\ s(z) = \mathit{unallocated} \\ s_0 = s[z \leftarrow \{\}] \\ s_0, n, r \vdash e \rightarrow a_1, s_1 \end{array}}{s, n, r \vdash \texttt{alloc\_before}\ \rho\ e \rightarrow a_1, s_1} \qquad [\mathit{ALLOCBEFORE}]$$

$$\frac{\begin{array}{l} s, n, r \vdash e \rightarrow a_1, s_1 \\ r(\rho) = z \\ s_1(z) \text{ is allocated} \\ s_2 = s_1[z \leftarrow \mathit{deallocated}] \end{array}}{s, n, r \vdash \texttt{free\_after}\ \rho\ e \rightarrow a_1, s_2} \qquad [\mathit{FREEAFTER}]$$

Figure 2: Operational semantics.

$$
\begin{array}{rcl}
[\![x]\!]\ R & = & [\![x]\!]\ R|_{\mathit{Vis}(x)} \\
[\![\lambda x.e \mathbin{@} \rho]\!]\ R & = & \{\langle \lambda x.e \mathbin{@} \rho, R\rangle\} \\
[\![e_1\ e_2]\!]\ R & & \text{for each } \langle \lambda x.e \mathbin{@} \rho, R'\rangle \in [\![e_1]\!]\ R \\
 & & \quad [\![e]\!]\ R' \subseteq [\![e_1\ e_2]\!]\ R \\
 & & \quad [\![e_2]\!]\ R \subseteq [\![x]\!]\ R' \\
[\![\texttt{let}\ x = e_1\ \texttt{in}\ e_2]\!]\ R & = & [\![e_2]\!]\ R \\
 & & [\![e_1]\!]\ R \subseteq [\![x]\!]\ R \\
[\![\texttt{letrec}\ f[\rho_1,\ldots,\rho_n](x) \mathbin{@} \rho = e_1\ \texttt{in}\ e_2]\!]\ R & = & [\![e_2]\!]\ R \\
[\![f[\rho_1' \ldots \rho_n'] \mathbin{@} \rho']\!]\ R & = & \{\langle \lambda x.e \mathbin{@} \rho', (R|_{\mathit{Vis}(f)})[\rho_i \leftarrow R(\rho_i')]\rangle\} \\
 & & \text{where } \texttt{letrec}\ f[\rho_1,\ldots,\rho_n](x) \mathbin{@} \rho = e\ \ldots \\
[\![\texttt{letregion}\ \rho\ \texttt{in}\ e]\!]\ R & = & [\![e]\!]\ R[\rho \leftarrow c] \text{ where } c \text{ is a color not in } R
\end{array}
$$

Figure 3: Region-based closure analysis.

3 Extended Closure Analysis

In reasoning about the memory behavior of a program, it is necessary to know the order of program reads and writes of memory. *Closure analysis* approximates execution order in higher-order programs [Shi88, Ses92]. However, closure analysis alone is not sufficient for our purposes, because of problems with *state polymorphism* and *region aliasing* (see below). Imprecision in state polymorphism gives poor completions, but failure to detect aliasing may result in unsound completions.

Consider again the program in Example 2.1. Note that, within the body of the function f, the $+$ operation is always the last use of the value k in ρ_5. Thus, it is safe to deallocate the region bound to ρ_5 inside the body of f after the sum:

$$
\begin{array}{l}
\texttt{letrec}\ f[\rho_5,\rho_6](k) \mathbin{@} \rho_3 = \texttt{letregion}\ \rho_7\ \texttt{in} \\
\quad \texttt{free\_after}\ \rho_5\ ((k + (1 \mathbin{@} \rho_7)) \mathbin{@} \rho_6)\ \texttt{end}\ \ldots
\end{array}
$$

Now consider the two uses of f in the body of the `letrec` in Example 2.1. With this completion, the region bound to ρ_1 is allocated (not shown) when $f[\rho_1,\rho_4]\ a$ is evaluated, and deallocated when $f[\rho_2,\rho_4]\ b$ is evaluated. Thus, to permit this completion the analysis of f must be polymorphic in the state (unallocated, allocated, or deallocated) of the region bound to ρ_1. If the analysis requires that the region bound to ρ_1 be in the same state at all uses of f, then in the body of f, the same region (now bound to ρ_5) cannot be deallocated.

Region aliasing occurs when two region variables in the same scope are bound to the same region value. There is no aliasing in Example 2.1 as written. However, if the expression $f[\rho_2,\rho_4]$ is replaced by $f[\rho_2,\rho_2]$, then region parameters ρ_5 and ρ_6 of f are bound to the same region. In this scenario, it is incorrect to deallocate the region bound to ρ_5 as shown above, since the result of the call to f (stored in the same region, but bound to ρ_6) is deallocated even though it is used later. This example illustrates three points. First, region aliasing must be considered in determining legal completions. Second, the completion of a function body depends strongly on the context in which the function is used; i.e., determining legal completions requires a global program analysis. Third, to obtain accurate completions, we require *precise* aliasing information. Approximate or *may-alias* information does not permit the allocation or deallocation of a region.

Our solution to these problems is to distinguish for each expression e the region environments in which e can be evaluated. We define $[\![e]\!]\ R$ to be the set of values to which e may evaluate in region environment R. Including region environments makes region aliasing explicit in the analysis. Since the only values are closures, $[\![e]\!]\ R$ is represented by sets of abstract closures $\{\langle \lambda x.e \mathbin{@} \rho, R'\rangle\}$, which intuitively denotes closures with function $\lambda x.e$ and region environment R'.

Since each `letregion` introduces a region, the set of region environments is infinite. We use a finite abstraction of region environments, mapping region variables to *colors*. A color stands for a set of runtime regions. An abstract region environment R has a very special property: R maps two region variables to the same color iff they are bound to the same region at runtime. Thus, an abstract region environment preserves the region aliasing structure of the underlying region environment.

The extended closure analysis is given in Figure 3. Following [PS92], the analysis is presented as a system of constraints; any solution of the constraints is sound. We assume that program variables are renamed as necessary so that each variable is identified with a unique binding. We write *Vis*(x) for the set of region variables in scope at `letrec` $x[\vec{\rho}](y) =$, `let` $x =$, or λx.

The rule for `letregion` introduces a new color c not already occurring in R. A distinct color is chosen because `letregion` allocates a fresh region, distinct from all existing regions. To make the analysis deterministic, colors are ordered and the minimal color is selected. There can be no more colors than the maximum number of region variables in scope at any point in the program. Thus, the set of abstract region environments is finite, which ensures that the closure constraints have a finite solution.

From the extended closure analysis, it is possible to derive an ordering on program points. For example, in an application $e_1\ e_2$ within region environment R, first e_1 is evaluated, then e_2, and finally one of the closures in $[\![e]\!]\ R$. This ordering plays a central role in computing completions.

4 Completions

Legal completions with explicit allocation/deallocation operations are expressed as a system of constraints. This section describes the constraint language, constraint generation, and constraint resolution. Constraint generation is a function of the input expression, the Tofte/Talpin types, and the result of the extended closure analysis.

4.1 Definitions

At each program point, every region in scope is in one of three *states*: unallocated (U), allocated (A), or deallocated (D). With each program point, abstract region environment, and color is associated a *state variable* ranging over $\{U, A, D\}$. State variables record the state of each region in the range of an abstract region environment at a program point. State variables are associated with regions (colors) rather than region variables because region variables may be aliased. Since the evaluation of an expression e may allocate/deallocate regions, a region state may be different before and after the evaluation of e. Thus, there are program points *in* and *out* for each expression e. We group state variables together into *state vectors* $S^{\text{in}}_{e,R}$ and $S^{\text{out}}_{e,R}$ associated with every expression e and region environment R. We refer to state variables by indexing state vectors with a color c, as in $S^{\text{in}}_{e,R}[c]$.

Constraints are placed on individual state variables in a state vector. There are three kinds of constraints: (1) *allocation constraints*, (2) *choice constraints*, and (3) *equality constraints*:

$$\begin{array}{ll} s = A & (1) \\ \langle s_1, c_p, s_2\rangle_a & \\ \langle s_1, c_p, s_2\rangle_d & (2) \\ s_1 = s_2 & (3) \end{array}$$

Allocation constraints are placed at program points where values are read from or written to a region; they express that a region must be allocated at this point.

Choice constraints are either *allocation triples* or *deallocation triples*. An allocation triple expresses a relationship between two state variables s_1, s_2 and a boolean variable c_p:

$$(c_p \Leftrightarrow (s_1 = U \wedge s_2 = A)) \wedge (\neg c_p \Leftrightarrow s_1 = s_2)$$

The boolean c_p encodes whether or not the associated region is to be allocated at program point p. If $c_p =$ *true* the region state prior to the allocation point is U and afterwards A, i.e. allocation. If $c_p =$ *false*, then the state prior is equal to the state after, i.e. no allocation. This approach is similar in spirit to the coercions of [Hen92]. The definition of deallocation triples is analogous:

$$(c_p \Leftrightarrow (s_1 = A \wedge s_2 = D)) \wedge (\neg c_p \Leftrightarrow s_1 = s_2)$$

Finally, equality constraints express that the state of a region is the same at two program points.

4.2 Constraint Generation

Constraint generation produces all constraints necessary to guarantee that regions are allocated when they are accessed. This task involves placing allocation constraints wherever regions are read or written, as well as linking the *in* and *out* states of each subexpression with the corresponding program points in the enclosing expression. Choice constraints are introduced at possible allocation or deallocation points and link the region states before and after the choice point.

What are the possible allocation and deallocation points? Every program point is a potential allocation or deallocation point for region variables that appear in the *overall effect* at that program point. Recall that the effect of e is the set of region variables possibly read or written during evaluation of e. The overall effect of an expression e is defined to be the arrow-effect (see Section 2) of the enclosing abstraction plus any `letregion`-bound variables inside the abstraction and in scope at e. We restrict the set of regions allowed to change state (be allocated or deallocated) on entry or exit of e to be regions in the overall effect of e. This restriction is crucial to the correctness of our system. A potential allocation (resp. deallocation) point is indicated by the syntax `alloc_before` $c_p\ e$ (resp. `free_before` $c_p\ e$), where c_p is the boolean variable associated with the allocation (resp. deallocation) point. Prior to constraint generation, all potential `alloc_before`, `free_after` expressions are added to the input program.

We briefly explain the constraint generation rules in Figure 4. Constraints are generated as a function of the *in* and *out* state vectors of each expression e, the current abstract region environment R, and the overall effect φ_o at e. The notation $R(\varphi)$ is the pointwise union of $R(\rho)$ for $\rho \in \varphi$, giving the set of colors in an effect. The rule for variables says that the state of regions in the overall effect is unchanged by a variable reference. No allocation constraint is needed, because no regions are read or written.

In the abstraction rule, we place an allocation constraint on the region where the closure is written. Furthermore, as in the variable rule, the states of all regions in the overall effect are the same on input and output of the abstraction expression.

The color $c_{e,R}$ in the `letregion` rule is the color chosen for ρ by the extended closure analysis in the same context.

Regions may change state only at potential allocation and deallocation points. The `alloc_before` rule connects the states of regions bound to ρ between the input states of e and e_1 with an allocation triple. The state of all other regions cannot change. A key point is that allocation triples generated from the same potential allocation point, but in different region environment contexts, share the same boolean

$$
\begin{array}{rcl}
e = x & \to & \forall c \in R(\varphi_o).\ S^{\text{in}}_{e,R}[c] = S^{\text{out}}_{e,R}[c] \\[1em]
e = \lambda x.e_1 @ \rho & \to & S^{\text{in}}_{e,R}[R(\rho)] = A \text{ and } \forall c \in R(\varphi_o).\ S^{\text{in}}_{e,R}[c] = S^{\text{out}}_{e,R}[c] \\[1em]
e = f[\vec{\rho}\,']@\rho' & \to & \begin{array}{l} S^{\text{in}}_{e,R}[R(\rho')] = A \\ S^{\text{in}}_{e,R}[R(\rho)] = A, \text{ where } (\tau, \rho) \text{ is the type of } f \\ \forall c \in R(\varphi_o).\ S^{\text{in}}_{e,R}[c] = S^{\text{out}}_{e,R}[c] \end{array} \\[2.5em]
e = e_1\, e_2 & \to & \begin{array}{l} \forall c \in R(\varphi_o).\ S^{\text{in}}_{e,R}[c] = S^{\text{in}}_{e_1,R}[c] \\ \forall c \in R(\varphi_o).\ S^{\text{out}}_{e_1,R}[c] = S^{\text{in}}_{e_2,R}[c] \\ \text{let } (\mu_1 \overset{\epsilon.\varphi}{\to} \mu_2, \rho) \text{ be the type of } e_1 \\ S^{\text{out}}_{e_2,R}[R(\rho)] = A \\ \text{for all } \langle \lambda x.e_0, R' \rangle \in [\![e_1]\!]\, R, \text{ with type } (\mu_1' \overset{\epsilon'.\varphi'}{\to} \mu_2', \rho) \\ \quad B = R(\varphi) = R'(\varphi') \\ \quad C = R(\varphi_o) - B \\ \quad \forall c \in B.\ S^{\text{out}}_{e_2,R}[c] = S^{\text{in}}_{e_0,R'}[c] \wedge S^{\text{out}}_{e_0,R'}[c] = S^{\text{out}}_{e,R}[c] \\ \quad \forall c \in C.\ S^{\text{out}}_{e_2,R}[c] = S^{\text{out}}_{e,R}[c] \end{array} \\[6em]
e = \texttt{let}\ x = e_1\ \texttt{in}\ e_2 & \to & \begin{array}{l} \forall c \in R(\varphi_o).\ S^{\text{in}}_{e,R}[c] = S^{\text{in}}_{e_1,R}[c] \\ \forall c \in R(\varphi_o).\ S^{\text{out}}_{e_1,R}[c] = S^{\text{in}}_{e_2,R}[c] \\ \forall c \in R(\varphi_o).\ S^{\text{out}}_{e_2,R}[c] = S^{\text{out}}_{e,R}[c] \end{array} \\[2.5em]
e = \texttt{letrec}\ f[\vec{\rho}](x)@\rho = e_1\ \texttt{in}\ e_2 & \to & \begin{array}{l} S^{\text{in}}_{e,R}[R(\rho)] = A \\ \forall c \in R(\varphi_o).\ S^{\text{in}}_{e,R}[c] = S^{\text{in}}_{e_2,R}[c] \\ \forall c \in R(\varphi_o).\ S^{\text{out}}_{e_2,R}[c] = S^{\text{out}}_{e,R}[c] \end{array} \\[2.5em]
e = \texttt{letregion}\ \rho\ \texttt{in}\ e_1 & \to & \begin{array}{l} R' = R[\rho \leftarrow c_{e,R}] \\ S^{\text{in}}_{e_1,R'}[c_{e,R}] = U \\ S^{\text{out}}_{e_1,R'}[c_{e,R}] = D \\ \forall c \in R(\varphi_o).\ S^{\text{in}}_{e,R}[c] = S^{\text{in}}_{e_1,R'}[c] \\ \forall c \in R(\varphi_o).\ S^{\text{out}}_{e_1,R'}[c] = S^{\text{out}}_{e,R}[c] \end{array} \\[4em]
e = \texttt{alloc\_before}\ \rho\ c_e\ e_1 & \to & \begin{array}{l} \forall c \in (R(\varphi_o) - \{R(\rho)\}).\ S^{\text{in}}_{e,R}[c] = S^{\text{in}}_{e_1,R}[c] \\ \forall c \in R(\varphi_o).\ S^{\text{out}}_{e_1,R}[c] = S^{\text{out}}_{e,R}[c] \\ \langle S^{\text{in}}_{e,R}[R(\rho)], c_e, S^{\text{in}}_{e_1,R}[R(\rho)] \rangle_a \end{array} \\[2.5em]
e = \texttt{free\_after}\ \rho\ c_e\ e_1 & \to & \begin{array}{l} \forall c \in R(\varphi_o).\ S^{\text{in}}_{e,R}[c] = S^{\text{in}}_{e_1,R}[c] \\ \forall c \in (R(\varphi_o) - R(\rho)).\ S^{\text{out}}_{e_1,R}[c] = S^{\text{out}}_{e,R}[c] \\ \langle S^{\text{out}}_{e_1,R}[R(\rho)], c_e, S^{\text{out}}_{e,R}[R(\rho)] \rangle_d \end{array}
\end{array}
$$

Figure 4: Constraint generation rules.

ρ_6	$s_{6,1}$	$s_{6,2} = A$	$s_{6,3}$	$s_{6,4}$				
ρ_5	$s_{5,1}$	$s_{5,2}$	$s_{5,3}$	$s_{5,4} = A$	$s_{5,5}$	$s_{5,6} = A$	$s_{5,7}$	$s_{5,8}$
ρ_4	$s_{4,1}$	$s_{4,2}$	$s_{4,3} = A$	$s_{4,4}$	$s_{4,5}$	$s_{4,6}$	$s_{4,7} = A$	$s_{4,8}$
operation	write	write	write	write	write	read	read	write
value	2	3	x (a pair)	λy	5	λy	x	pair
region	ρ_2	ρ_6	ρ_4	ρ_5	ρ_3	ρ_5	ρ_4	ρ_1

Table 1: Example constraint resolution.

variable, which guarantees that the completion is valid in all contexts. Allocation/deallocation choice points for different region variables are sequentialized to ensure that if two region variables are aliased (i.e. they map to the same color in the abstract region environment), at most one allocation/deallocation point is chosen.

The application rule is the most difficult. The key idea is that at runtime, the regions in the arrow-effect of the function expression e_1 (call this set E), are the same as the regions in the effect of the closure. Therefore, the states of regions in E in the caller's context prior to evaluation of the function body match the states of regions in E on entry to the function (and similarly on return). In the abstract region environments of the caller and callee, the colors of the effect of the call (set B) are equal, justifying equality constraints between state variables at the call site and in the input vector of the function body (similarly on output). These equality constraints model the flow of regions from the caller into the function body and back. All regions a function touches appear in the function's effect. It is thus sufficient to place the equality constraints only on state variables corresponding to colors from B. Other regions in the caller's context (set C) are not touched in the the function body; the function is state-polymorphic in these regions. The set of possible closures in an application of a given region environment is computed by the extended closure analysis. For brevity, we do not describe the handling of quantified effect variables (for details see [AFL95]).

4.3 Constraint Resolution

In general, the constraint system has multiple solutions. For example, the state of a region after the last use is unspecified. We may place the point of deallocation of such a region anywhere after its last use, but obviously we prefer the first possible program point. The choice of where to allocate (or deallocate) a region affects the states of regions in other parts of the program. Therefore, it is necessary to iterate solving constraints and choosing allocation/deallocation points based on the partial solutions.

Recall Example 1.1. Consider ρ_5 and the control flow path from the point p_1, where the lambda abstraction is stored in the region bound to ρ_5, to the point p_2, where it is retrieved to perform the application. Clearly the region bound to ρ_5 must be allocated both at p_1 and p_2. Because the language semantics forbid the region to change from the deallocated state to the allocated state, we can conclude that on all control paths from p_1 to p_2, it must be allocated.

The constraints are simple first-order formulas for which resolution algorithms are well-known. There is, however, the issue of deciding which solution to choose; clearly some completions are better than others. We illustrate our resolution algorithm with an example.

Refer again to the example in Figure 1. Table 1 shows the state variables associated with ρ_4, ρ_5, ρ_6. Assume that we have added allocation triples between all consecutive program points for colors bound by ρ_4—ρ_6, with associated boolean variables $c_{i,j}$, meaning a possible allocation of ρ_i just after state $s_{i,j}$.

Table 1 contains explicit allocation constraints on states where regions are accessed. We must have $s_{5,5} = A$ because it lies on an execution path between two states where the region bound to ρ_5 is allocated. The same holds for $s_{4,4\text{–}6}$. We also set all allocation choice points $c_{6,2\text{–}4}$, $c_{5,4\text{–}8}$, and $c_{4,3\text{–}8}$ to *false*, because the regions must be allocated before these program points are reached. At this point we have proven all facts derivable from the initial constraints—nothing forces other states to be unallocated, allocated, or deallocated. We can now choose to set any boolean variable c_p of an allocation triple $\langle s_1, c_p, s_2\rangle_a$ to *true*, if the variable c_p is not constrained. Among the possible choices, we are particularly interested in allocation points lying on the border of an unconstrained state and an allocated state, i.e., allocation triples $\langle s_1, c_p, s_2\rangle_a$ where:

$$s_1 \text{ is } \textit{unconstrained} \wedge s_2 = A$$

By the definition of an allocation triple, choosing $c_p = \textit{true}$ forces $s_1 = U$. The state U is propagated to earlier program points, since the region can be in no other state there. In the example, we choose $c_{5,3} = \textit{true}$, set $s_{5,3} = U$, and propagate U backwards through $s_{5,2\text{–}1}$ to the `letregion` for ρ_5. Similarly, we choose $c_{6,1} = \textit{true}$ and $c_{4,2} = \textit{true}$.

In general, given a constraint system $\mathcal{C}$, we first prove all facts $\mathcal{C} \vdash s = X$ and $\mathcal{C} \vdash s \neq X$ implied by $\mathcal{C}$. If $\mathcal{C} \not\vdash s = X$ and $\mathcal{C} \not\vdash s \neq X$, then we are free to choose either $s = X$ or $s \neq X$. This procedure repeats, proving facts and making choices, until a complete solution is constructed.

Any solution of the constraints specifies a completion of the program, where allocate/deallocate operations are added for the boolean variables c_p that are *true* in the solution. The constraints have a trivial solution, obtained by choosing for each region the first allocation choice point and the last deallocation choice point inside the corresponding `letregion`. This most conservative completion has exactly the same memory behavior as the original Tofte/Talpin program (e.g. Figure 1a).

5 Soundness

This section states a soundness theorem for our system and sketches the proof. The soundness theorem is formulated as follows. Assume that $s, r, n \vdash e \rightarrow a, s'$, and assume that $[\![e]\!]\, R = V$ is the result of the extended closure analysis for e, where R is an abstraction of the region environment r. Assume further that the regions of the overall effect φ_o mapped by r in store s are initially in the states given by $S^{\text{in}}_{e,R}$. The theorem shows that the evaluation of e leaves these regions in the states specified by $S^{\text{out}}_{e,R}$. To prove this theorem we first state the relationship between the concrete semantics and our abstraction. For the proof concrete regions in the operational semantics are colored the same way as in the extended closure analysis. We use capital letters for abstract entities and lowercase letters for concrete entities, z_c denotes a concrete region with color c, s is a concrete store, and $\mathcal{S}$: StateVar $\rightarrow \{U, A, D\}$ is the solution of the constraints.

We say a concrete region environment r satisfies an abstract region environment R if they have the same domain and aliasing structure.

$$\begin{aligned}
&r \;\mathsf{sat}\; R \overset{\text{def}}{\equiv} \\
&\quad \mathit{Dom}(R) = \mathit{Dom}(r) \wedge \\
&\quad R(\rho) = R(\rho') \iff r(\rho) = r(\rho') \wedge \\
&\quad R(\rho) = c \iff r(\rho) = z_c
\end{aligned}$$

A store s and address a satisfy a set of abstract values V, if V contains an abstraction of the concrete value stored at address a in s, and the environment of the concrete closure satisfies the extended closure analysis $[\![\cdot]\!]$.

$$\begin{aligned}
&s, a \;\mathsf{sat}\; V \overset{\text{def}}{\equiv} \\
&\quad \text{address } a \text{ is allocated in } s \Longrightarrow \\
&\qquad s(a) = \langle \lambda x.e, r', n' \rangle \wedge \\
&\qquad \exists \langle \lambda x.e, R' \rangle \in V \text{ s.t.} \\
&\qquad\quad s, r', n' \;\mathsf{sat}\; R', [\![\cdot]\!]
\end{aligned}$$

A store s, concrete region environment r, and concrete value environment n satisfy an abstract region environment R and the extended closure analysis $[\![\cdot]\!]$, if the region environments match and for every variable x in the concrete environment, $[\![x]\!]\; R$ contains an abstraction satisfying the concrete value.

$$\begin{aligned}
&s, r, n \;\mathsf{sat}\; R, [\![\cdot]\!] \overset{\text{def}}{\equiv} \\
&\quad r \;\mathsf{sat}\; R \wedge \\
&\quad \forall (x \in \mathit{Dom}(n)) \exists R' \text{ s.t.} \\
&\qquad (R|_{\mathit{Dom}(R')} = R' \;\wedge\; s, n(x) \;\mathsf{sat}\; [\![x]\!]\; R')
\end{aligned}$$

A store s and a concrete region z_c with color c satisfy a state variable $S_{e,R}[c]$ if the state of the region z_c in the store s corresponds to the solution for $S_{e,R}[c]$.

$$\begin{aligned}
&s, z_c \;\mathsf{sat}\; S_{e,R}, c \overset{\text{def}}{\equiv} \\
&\quad z_c \in \mathit{Dom}(s) \Longrightarrow \mathrm{state}(s, z_c) = \mathcal{S}(S_{e,R}[c])
\end{aligned}$$

Finally, a state s and concrete region environment r satisfy an abstract region environment R, state vector $S_{e,R}$, and effect set φ if r and R match and the states of all regions in φ match the solution of the constraints for $S_{e,R}$.

$$\begin{aligned}
&s, r \;\mathsf{sat}\; R, S_{e,R}, \varphi \overset{\text{def}}{\equiv} \\
&\quad \varphi \subseteq \mathit{Dom}(R) \wedge \\
&\quad r \;\mathsf{sat}\; R \wedge \\
&\quad \forall (\rho \in \varphi)\; s, r(\rho) \;\mathsf{sat}\; S_{e,R}, R(\rho)
\end{aligned}$$

The soundness of our analysis is summarized by Theorem 5.1.

Theorem 5.1 Given that

$$\begin{aligned}
&s, r, n \vdash e \rightarrow a, s' \\
&[\![e]\!]\; R = V \\
&s, r, n \;\mathsf{sat}\; R, [\![\cdot]\!] \\
&s, r \;\mathsf{sat}\; R, S^{\text{in}}_{e,R}, \varphi_o
\end{aligned}$$

we conclude

$$\begin{aligned}
&s', a \;\mathsf{sat}\; V \\
&s', r \;\mathsf{sat}\; R, S^{\text{out}}_{e,R}, \varphi_o
\end{aligned}$$

The proof is by induction on the structure of e and is included in the full version of the paper [AFL95].

6 Implementation and Experiments

We have implemented our algorithm in Standard ML [MTH90]. Our system is built on top of an implementation of the system described in [TT93, TT94], generously provided to us by Mads Tofte. The implementation is extended with numbers, pairs, lists, and conditionals, so that non-trivial programs can be tested. For each source program, we first use the Tofte/Talpin system to region annotate the program. We then compute the extended closure analysis (Section 3). The next step adds allocation and deallocation choice points and generates the allocation constraints (Section 4). The constraints are solved and the solution is used to complete the source program, transforming selected choice points into allocation/deallocation operations, and removing the rest.

Our annotations are orthogonal to the storage mode analysis mentioned in [TT94] and described in more detail in [Tof94]. Thus, the target programs contain both storage mode annotations and the allocation annotations described in this paper. On the other hand, our analysis subsumes the optimization described in Appendix B of [TT94], so that optimization is disabled in our system. Summary performance measures are in Table 2. We have not measured carefully the time required to compute our analysis, but our method appears to scale as well as the Tofte/Talpin system. All of the examples we have tried are analyzed in a matter of seconds by our system on a standard workstation.

The target programs were run on an instrumented interpreter, also written in Standard ML/NJ. In addition to the data above, we also gather complete memory traces, which we present as graphs depicting memory usage over time.

While we have tested our system on many programs, neither the size of our benchmarks nor the size of our benchmark suite is large enough to draw meaningful statistical conclusions. Instead, we present representative examples of three typical patterns of behavior we have identified.

A number of programs show asymptotic improvement over the Tofte/Talpin system. One example given in their paper (due to Appel [App92]), has $O(n^2)$ space complexity. Our completion of this program exhibits $O(n)$ space complexity (Figure 5). In this program, our analysis is able to deallocate a recursive function's parameter before function evaluation completes. Because the Tofte/Talpin system enforces a stack discipline, it cannot reclaim function parameters that become "dead" part way through the activation of a function.

Another typical pattern is that our system has the same asymptotic space complexity as Tofte/Talpin, but with a constant factor improvement. Representative examples include Quicksort, Fibonacci, and Randlist. The memory usage graphs are shown in Figures 6, 7, and 8, respectively. The measurements for the graphs were made using smaller inputs than the experiments in Table 2; smaller problem sizes yield more readable graphs.

The Quicksort graph (Figure 6) has a curious feature: at times the memory usage drops below the amount needed to store the list! Our measurements count only heap memory

	Appel(100)		Quicksort(500)		Fibonacci(6)		Randlist(25)		Fac(10)	
	A-F-L	T-T	A-F-L	T-T	A-F-L	T-T	A-F-L	T-T	A-F-L	T-T
(1)	208	1111	112	1520	15	20	12	90	25	25
(2)	81915	81915	45694	45694	190	190	289	289	66	66
(3)	101814	101814	65266	65266	190	190	363	363	66	66
(4)	306	20709	2509	8078	10	14	85	161	14	14
(5)	1	1	1502	1502	1	1	77	77	1	1

(1) Maximum number of regions allocated (unit: 1 region)
(2) Total number of region allocations
(3) Total number of value allocations
(4) Maximum number of storable values held (unit: 1 *sv*)
(5) Number of values stored in the final memory (unit: 1 *sv*)

Table 2: Summary of results.

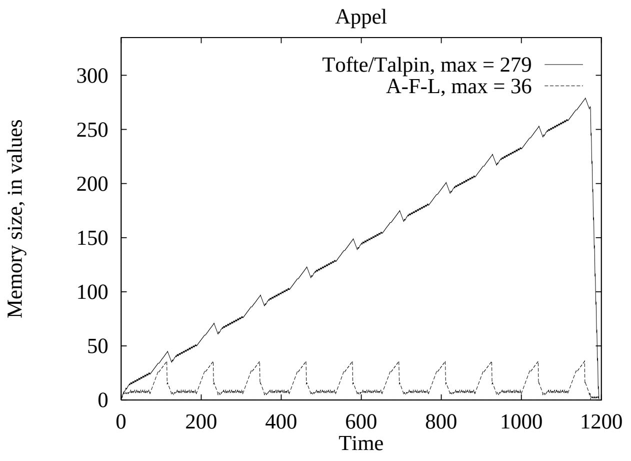


Figure 5: Memory usage in Appel example [App92] *($n = 10$)*.

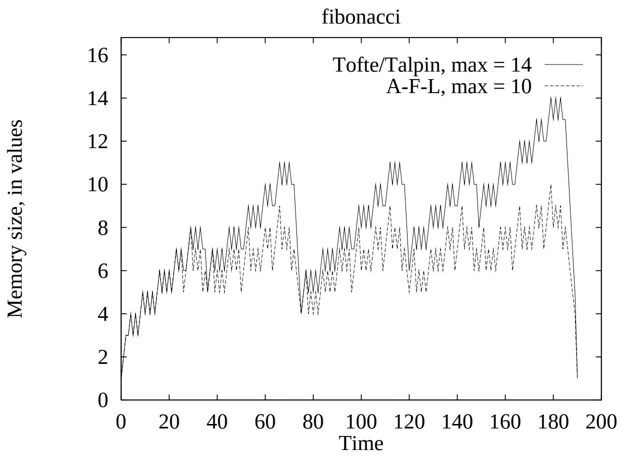


Figure 7: Memory usage in Fibonacci example *(recursive fibonacci of 6)*.

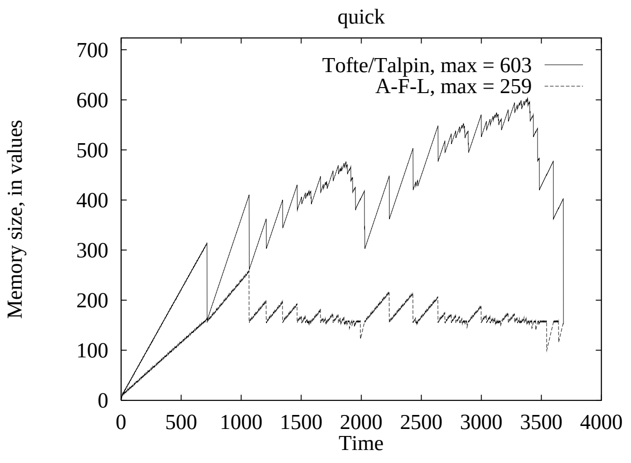


Figure 6: Memory usage in Quicksort example *(sort 50 element list of random integers)*.

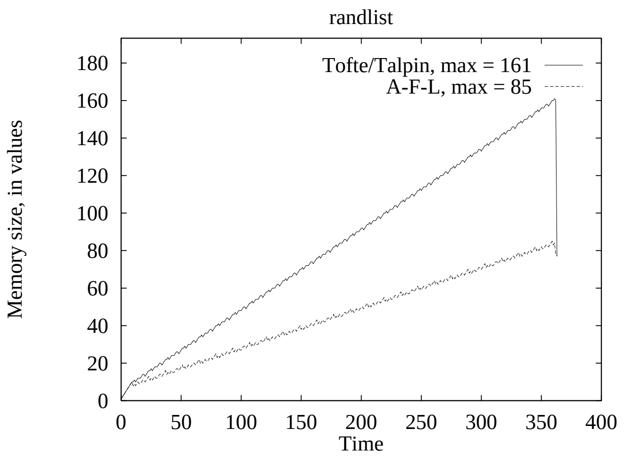


Figure 8: Memory usage in Randlist example *(generate 25 element list of random integers)*.

usage. The evaluation stack is not counted, a measurement methodology consistent with [TT94]. Quicksort is not unusual in this behavior. The program recursively traverses its input list, stores the contents on the evaluation stack, frees the list cells when it reaches the end, and builds up the output list upon return.

In the third class of programs our system has nearly the same memory behavior as Tofte/Talpin (e.g., the factorial function). This case arises most often when the Tofte/Talpin annotation is either already the best possible or very conservative. Conservative annotations distinguish few regions. Because values in regions must be deallocated together, having fewer regions results in coarser annotations. Of course, the memory behavior of a program annotated using our algorithm is never worse than that of the same program annotated using the Tofte/Talpin algorithm.

Our system is accessible for remote experimentation through the World Wide Web at:

```
http://kiwi.cs.berkeley.edu/˜nogc
```

7 Discussion and Conclusions

It remains an open question whether our system is a practical approach to memory management. The complexity of the extended closure analysis is worst-case exponential time. In practice, we have found it to be of comparable complexity to the Tofte/Talpin system, but we do not as yet have enough experience to judge whether this holds in general. The constraint generation and constraint solving portions of our analysis both run in low-order polynomial time. A separate issue is that the global nature of our analysis presents serious problems for separate compilation, which we leave as future work. Finally, we have found that static memory allocation is very sensitive to the form of the program. Often, a small change to the program, such as copying one value, makes a dramatic difference in the quality of the completion. Thus, for this approach to memory management to be practical, feedback to programmers about the nature of the completion will be important.

Our system does do a good job of finding very fine-grain, and often surprising, memory management strategies. Removing the stack allocation restriction in the Tofte/Talpin system allows regions to be freed early and allocated late. The result is that programs often require significantly less memory (in some cases asymptotically less) than when annotated using the Tofte/Talpin system alone.

References

[AFL95] Alexander Aiken, Manuel Fähndrich, and Raph Levien. Better static memory management: Improving region-based analysis of higher-order languages. Technical Report CSD-95-866, UC Berkeley, April 1995.

[App92] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[Deu90] Alain Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Proc. of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 157–168, January 1990.

[DLM$^+$78] Edsger W. Dijkstra, Leslie Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[Hen92] Fritz Henglein. Global tagging optimization by type inference. In *Proc. of the 1992 ACM Conference on Lisp and Functional Programming*, pages 205–215, July 1992.

[HJ90] Geoff W. Hamilton and Simon B. Jones. Compile-time garbage collection by necessity analysis. In *Proc. of the 1990 Glasgow Workshop on Functional Programming*, pages 66–70, August 1990.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[NO93] Scott Nettles and James O'Toole. Real-time replication garbage collection. In *Proc. SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 217–226, June 1993.

[PS92] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference. *Information Processing Letters*, 43(4):175–180, September 1992.

[RM88] Cristina Ruggieri and Thomas P. Murtagh. Lifetime analysis of dynamically allocated objects. In *Proc. of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 285–293, January 1988.

[Ses92] Peter Sestoft. *Analysis and Efficient Implementation of Functional Programs*. PhD dissertation, University of Copenhagen, Department of Computer Science, 1992.

[Shi88] Olin Shivers. Control flow analysis in Scheme. In *Proc. SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 164–174, June 1988.

[Tof94] Mads Tofte. Storage mode analysis. Personal communication, October 1994.

[TT93] Mads Tofte and Jean-Pierre Talpin. A theory of stack allocation in polymorphically typed languages. Technical Report 93/15, Department of Computer Science, University of Copenhagen, July 1993.

[TT94] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proc. of the 21st Annual ACM Symposium on Principles of Programming Languages*, pages 188–201, January 1994.