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

(Tech Report CSD-95-866)

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 a factor of $\Omega(n)$ or more. 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

*This material is based in part upon work supported by NSF Young Investigator Award No. CCR-9457812 and NSF Infrastructure Grant No. CDA-9401156. The content of the information does not necessarily reflect the position or the policy of the Government.

†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

the two memory management models. We expect that implementations of applicative languages with static memory management would make writing components 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

$$
\begin{array}{l}
\texttt{letregion}\ \rho_4,\rho_5\ \texttt{in} \\
\quad \texttt{letregion}\ \rho_6\ \texttt{in} \\
\quad\quad \texttt{let}\ x = \texttt{(2@}\rho_2\texttt{, 3@}\rho_6\texttt{)@}\rho_4\ \texttt{in} \\
\quad\quad\quad (\lambda y.\texttt{(fst}\ x,y\texttt{)@}\rho_1\texttt{)@}\rho_5 \\
\quad\quad \texttt{end} \\
\quad \texttt{end 5@}\rho_3 \\
\texttt{end}
\end{array}
$$

There are two kinds of annotations: $\texttt{letregion}\ \rho\ \texttt{in}\ e$ 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 $\lambda y.\dots$ (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 $\texttt{alloc\_before}\ \rho\ e$ allocates space for the region bound to ρ before evaluating e, and the operation $\texttt{free\_after}\ \rho\ e$ 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 $\texttt{alloc\_before}\ \rho$ and $\texttt{free\_after}\ \rho$ 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

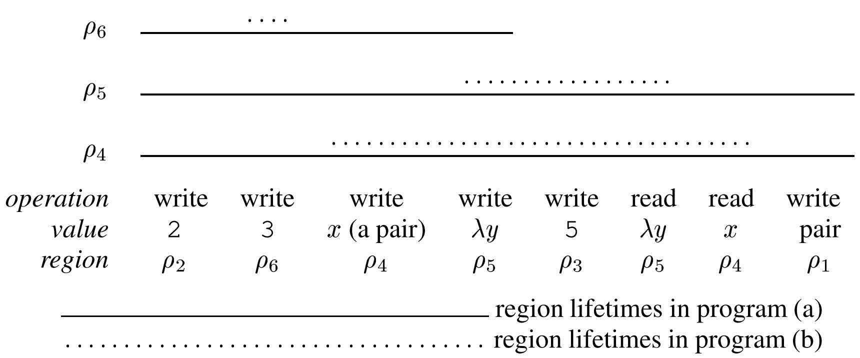
[1] We assume small integers are boxed to make the presentation simple and uniform. In practice, small integers can be unboxed.

```
letregion ρ4, ρ5 in
    alloc_before ρ4 free_after ρ4 alloc_before ρ5 free_after ρ5
    letregion ρ6 in
        alloc_before ρ6 free_after ρ6
        let x = (2@ρ2, 3@ρ6)@ρ4 in
            (λy.(fst x, y)@ρ1)@ρ5
        end
    end 5@ρ3
end
```

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

```
letregion ρ4, ρ5 in
    free_app ρ5
        letregion ρ6 in
            let x = (2@ρ2, alloc_after ρ4 alloc_before ρ6 free_after ρ6 3@ρ6)@ρ4 in
                alloc_before ρ5 (λy.(free_after ρ4 fst x, y)@ρ1)@ρ5
            end
        end 5@ρ3
end
```

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



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

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

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 (Section 2), ultimately producing a completed program with explicit allocation and deallocation operations (Section 3). 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 4). In the second phase, local constraints are generated from the *(expression, region environment)* pairs (Section 5). 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."

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.

The soundness proof is presented in Section 6. Detailed discussion and measurements of the behavior of our algorithm are presented in Section 7. A discussion of some relevant related work is in Section 8. Section 9 concludes with a discussion of practical issues. Our system is accessible for remote experimentation through the World Wide Web. The server analyzes arbitrary programs, and displays the translated program as well graphs showing memory usage over time. The URL is:

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

2 Background on the Tofte/Talpin System

Our approach makes use of Tofte and Talpin's *region and effect inference* algorithm. This section describes the Tofte/Talpin region inference system in more detail. For a full description, please refer to [TT94]. This section is intended as an informal overview to aid in understanding our extensions.

We use the term *full system* to refer to the composition of the Tofte/Talpin system as described in [TT94] and our extensions. In the full system, each source program is first translated by the Tofte/Talpin system, and then the translated program is further analyzed and annotated by our extensions.

2.1 Source language

The source language of the full system, and thus of the Tofte/Talpin region inference system, is an ML-like functional language. We present only the core language, omitting pairing, selection, lists, and arithmetic operations for clarity. The grammar is:

$$\begin{array}{rcl} e & ::= & x \mid \lambda x.e \mid e_1\ e_2 \\ & \mid & \texttt{let}\ x = e_1\,\texttt{in}\ e_2\ \texttt{end} \\ & \mid & \texttt{letrec}\ f(x) = e_1\ \texttt{in}\ e_2\ \texttt{end} \end{array}$$

The operational semantics for the source language is quite standard, and given in Figure 2. The rules derive sentences of the form $E \vdash e \rightarrow v$, meaning that in environment E, expression e evaluates to value v. There is no explicit store. The notation $E + E'$ extends finite maps: $(E + E')(x)$ is $E'(x)$ when $x \in \mathit{Dom}(E')$, or $E(x)$ otherwise. The notation $\{x \mapsto v\}$ stands for the singleton map which maps x to v.

2.2 Target language of of the Tofte/Talpin system

The Tofte/Talpin region inference system translates source language programs into *target* language programs. In the context of the full system, this target language is the input to the extended closure analysis and constraint generation phases presented later in this report.

The target language is given by the following grammar:

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

The target language differs from the source language in that allocation and use of memory is made explicit. The dynamic semantics of the target language introduces a new, nonstandard model of the store. In this model, the store is a stack of regions, each capable of holding an arbitrary number of values. Any region in the stack can support both *get* (retrieving a value previously stored in the region) and *put* (increasing the size of the region by one value, and storing a value in the space thus allocated) operations.

The stack of regions model differs from the traditional stack model in *put* operations are not restricted to the top of the stack. Thus, we cannot implement the stack of regions using a traditional stack. Rather, the implementation technique is similar to the "stack of subheaps" scheme of Ruggieri and Murtagh [RM88].

The target language introduces two annotations:

$$\begin{array}{l} \texttt{letregion}\ \rho\ \texttt{in}\ e\ \texttt{end} \\ e@\rho \end{array}$$

$$\frac{E(x) = v}{E \vdash x \rightarrow v}$$

$$\frac{}{E \vdash \lambda x.e \rightarrow \langle x, e, E \rangle}$$

$$\frac{E \vdash e_1 \rightarrow \langle x_0, e_0, E_0 \rangle \qquad E \vdash e_2 \rightarrow v_2 \qquad E_0 + \{x_0 \mapsto v_2\} \vdash e_0 \rightarrow v}{E \vdash e_1\, e_2 \rightarrow v}$$

$$\frac{\begin{array}{c} E \vdash e_1 \rightarrow \langle x_0, e_0, E_0, f \rangle \qquad E \vdash e_2 \rightarrow v_2 \\ E_0 + \{f \mapsto \langle x_0, e_0, E_0, f \rangle\} + \{x_0 \mapsto v_2\} \vdash e_0 \rightarrow v \end{array}}{E \vdash e_1\, e_2 \rightarrow v}$$

$$\frac{E \vdash e_1 \rightarrow v_1 \qquad E + \{x \mapsto v_1\} \vdash e_2 \rightarrow v}{E \vdash \texttt{let}\; x = e_1 \;\texttt{in}\; e_2 \rightarrow v}$$

$$\frac{E + \{f \mapsto \langle x, e_1, E, f \rangle\} \vdash e_2 \rightarrow v}{E \vdash \texttt{letrec}\; f\,\texttt{(}x\texttt{)} = e_1 \;\texttt{in}\; e_2 \rightarrow v}$$

Figure 2: Operational semantics of source language.

The first annotation (letregion ρ in e end) specifies a runtime sequence of operations: first a new, empty region is created and bound to *region variable* ρ, then e is evaluated, and finally the region (including all its contents) is destroyed. The second annotation ($e@\rho$) simply specifies that the result of evaluating expression e will be stored at the region bound to the region variable ρ. Region variable bindings are lexical.

Any operation (*get* or *put*) dynamically occurring outside the extent of the corresponding letregion is an error. That the translation must not derive programs that can exhibit this error is the major correctness constraint.

Just adding letregion and "$@\rho$" annotations would result in very conservative translations for most interesting programs. Thus, Tofte and Talpin extended their system to include *region polymorphism,* which allows expressions to perform operations on different regions depending on the context. In the Tofte/Talpin system, introduction of region polymorphism is combined with the letrec construct. A region polymorphic function takes extra region parameters, which are bound to actual regions by a region instantiation construct. The syntax for region polymorphic letrec is:

$$\texttt{letrec}\ f[\vec{\rho}](x) = e_1\ \texttt{in}\ e_2\ \texttt{end}$$

Similarly, the syntax for region instantiation is:

$$f[\vec{\rho}]@\rho$$

Region instantiation retrieves the region polymorphic closure, instantiates the formal regions parameters of the closure with actual region parameters $\vec{\rho}$, and stores a new region monomorphic closure in ρ (see rule [*REGAPP*] in Figure 4).

Region polymorphism is implemented by passing the regions at runtime as extra parameters to the function.

2.3 Types

Source language types are the standard ones of the Damas-Milner type system [DM82], as specified by the following grammar:

$$\tau ::= \alpha \mid \text{int} \mid \tau \rightarrow \tau$$

The target language has a somewhat richer type language, which describes an expression's use of regions, as well as the type of its value. In the following grammar, τ represents the type of an expression, and μ represents the type and region.

$$\begin{array}{lcl} \tau & ::= & \alpha \mid \text{int} \mid \mu \xrightarrow{\epsilon.\varphi} \mu \\ \mu & ::= & (\tau, \rho) \end{array}$$

The $\epsilon.\varphi$ denotes an *arrow effect.* The *effect* of evaluating an expression is the set of regions accessed (i.e. read or written) as a result of that evaluation. The arrow efffect of a function is the effect of applying that function. In [TT94], there is a further distinction between *get* and *put* effects, but that distinction is not needed here.

The ϵ names the effect, and is useful for two purposes. First, when quantified, it reflects that the function can be *effect polymorphic*, which is to say that its effect can depend on the context. A typical example is a higher-order function that calls its argument. The arrow effect of the argument is incorporated into the overall effect of applying the function; different arrow effects for the argument will result in different effects for the application.

By design, a value's type captures all regions accessible by the value, either by traversal in the case of pairs and lists, or by application in the case of functions.

Type schemes are given by the following grammar:

$$\begin{array}{lcll} \sigma & ::= & \forall \alpha_1, \ldots, \alpha_n, \epsilon_1, \ldots, \epsilon_m.\tau & \text{simple type schemes} \\ & \mid & \forall \rho_1, \ldots, \rho_k, \alpha_1, \ldots, \alpha_n, \epsilon_1, \ldots, \epsilon_m.\tau & \text{compound type schemes} \end{array}$$

Simple type schemes quantify over both ordinary type variables and effect variables, and are introduced by let constructs. Compound type schemes quantify over region variables as well, and are introduced by letrec constructs.

Given a compound type scheme $\sigma = \forall \rho_1, \ldots, \rho_k, \alpha_1, \ldots, \alpha_n, \epsilon_1, \ldots, \epsilon_m.\tau$ and type τ', we use the notation of [TT94] to denote instantiation: $\sigma \geq \tau'$ *(via S)* means that τ' is *an instance of* σ, if there exists a substitution S such that $S(\tau) = \tau'$. Instantiation in the case where σ is a simple type scheme is defined similarly.

As an example of the types, consider the following target language code fragment:

$$\texttt{letrec}\ f\,\texttt{[}\rho_1\texttt{,}\,\rho_2\texttt{]}\ \texttt{(}x : (\text{int}, \rho_1)\texttt{)} = \texttt{(}x\texttt{+}\,\texttt{(1}@\rho_0\texttt{))}\,@\rho_2\ \texttt{in}\ \ldots$$

The region polymorphic function f has the following compound type scheme:

$$\forall \rho_1, \rho_2, \epsilon_1.(\text{int}, \rho_1) \xrightarrow{\epsilon_1.\{\rho_0, \rho_1, \rho_2\}} (\text{int}, \rho_2)$$

$$\frac{TE(x) = (\sigma, \rho) \qquad \sigma \text{ simple} \qquad \sigma \geq \tau}{TE \vdash x \Rightarrow x : (\tau, \rho), \emptyset} \qquad [\mathit{VAR}]$$

$$\frac{\begin{array}{l} TE(f) = (\sigma, \rho') \qquad \sigma \text{ compound, i.e. } \sigma = \forall \rho_1, \ldots, \rho_k.\sigma_1, \text{ where } \sigma_1 \text{ is simple} \\ \sigma \geq \tau \text{ via } S \qquad \varphi = \{\rho, \rho'\} \end{array}}{TE \vdash f \Rightarrow f[S(\rho_1), \ldots, S(\rho_k)]@\rho : (\tau, \rho), \varphi} \qquad [\mathit{REGAPP}]$$

$$\frac{TE + \{x \mapsto \mu_1\} \vdash e \Rightarrow e' : \mu_2, \varphi \qquad \varphi \subseteq \varphi'}{TE \vdash \lambda x.e \Rightarrow \lambda x.e'@\rho : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \rho), \{\rho\}} \qquad [\mathit{ABS}]$$

$$\frac{TE \vdash e_1 \Rightarrow e_1' : (\mu' \xrightarrow{\epsilon.\varphi} \mu, \rho), \varphi_1 \quad TE \vdash e_2 \Rightarrow e_2' : \mu', \varphi_2}{TE \vdash e_1\, e_2 \Rightarrow e_1'\, e_2' : \mu, \varphi \cup \varphi_1 \cup \varphi_2 \cup \{\epsilon, \rho\}} \qquad [\mathit{APP}]$$

$$\frac{\begin{array}{l} TE \vdash e_1 \Rightarrow e_1' : (\tau_1, \rho_1), \varphi_1 \\ TE + \{x \mapsto (\sigma, \rho_1)\} \vdash e_2 \Rightarrow e_2' : \mu, \varphi_2 \qquad \sigma = \text{TyEffGen}(TE, \varphi_1)(\tau_1) \end{array}}{TE \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Rightarrow \texttt{let}\ x = e_1'\ \texttt{in}\ e_2' : \mu, \varphi_1 \cup \varphi_2} \qquad [\mathit{LET}]$$

$$\frac{\begin{array}{l} TE + \{f \mapsto (\forall \vec{\rho}\,\vec{\epsilon}.\tau, \rho)\} \vdash \lambda x.e_1 \Rightarrow \lambda x.e_1'@\rho : (\tau, \rho), \varphi_1 \\ \forall \vec{\rho}\,\vec{\epsilon}.\tau = \text{RegEffGen}(TE, \varphi_1)(\tau) \\ \sigma' = \text{RegTyEffGen}(TE, \varphi_1)(\tau) \\ TE + \{f \mapsto (\sigma', \rho)\} \vdash e_2 \Rightarrow e_2' : \mu, \varphi_2 \end{array}}{\begin{array}{l} TE \vdash \texttt{letrec}\ f(x) = e_1\ \texttt{in}\ e_2 \Rightarrow \\ \qquad \texttt{letrec}\ f[\vec{\rho}](x)@\rho = e_1'\ \texttt{in}\ e_2' : \mu, \varphi_1 \cup \varphi_2 \end{array}} \qquad [\mathit{LETREC}]$$

$$\frac{TE \vdash e \Rightarrow e' : \mu, \varphi \qquad \varphi' = \text{Observe}(TE, \mu)(\varphi) \qquad \{\rho_1, \ldots, \rho_k\} = \text{frv}(\varphi \setminus \varphi')}{TE \vdash e \Rightarrow \texttt{letregion}\ \rho_1, \ldots, \rho_k\ \texttt{in}\ e' : \mu, \varphi'} \qquad [\mathit{LETREGION}]$$

Figure 3: Region inference rules.

2.4 The region inference algorithm

A full description of the region inference algorithm is beyond the scope of this report (for more details, see [TT94]). However, we give the inference rules (Figure 3) and a high-level overview.

The inference rules are a refinement of the ML type inference rules [DM82]. They derive translations of the following form:

$$TE \vdash e \Rightarrow e' : \mu, \varphi$$

This sentence states that, in type environment TE, source language expression e translates into target language expression e', which has type and place μ, and effect φ. A type environment is a finite map from program variables to (σ, ρ) pairs.

For any semantic object A, frv(A) is the set of region variables that occur free in A, ftv(A) is the set of free type variables in A, and fev(A) is the set of free effect variables in A. The notation $A \setminus B$ represents set difference. The three operations for forming type schemes are defined thus:

$$\begin{array}{rcl} \text{RegTyEffGen}(A)(\tau) & = & \forall\, \text{frv}(\tau) \setminus \text{frv}(A), \text{ftv}(\tau) \setminus \text{ftv}(A), \text{fev}(\tau) \setminus \text{fev}(A).\tau \\ \text{RegEffGen}(A)(\tau) & = & \forall\, \text{frv}(\tau) \setminus \text{frv}(A), \text{fev}(\tau) \setminus \text{fev}(A).\tau \\ \text{TyEffGen}(A)(\tau) & = & \forall\, \text{ftv}(\tau) \setminus \text{ftv}(A), \text{fev}(\tau) \setminus \text{fev}(A).\tau \end{array}$$

The *observable part of* φ *with respect to* A is written Observe$(A)(\varphi)$ and is defined to be the following subset of φ:

$$\text{Observe}(A)(\varphi) = \varphi \cap (\text{frv}(A) \cup \text{fev}(A))$$

The [*LETREGION*] rule, in its use of the Observe function, contains the key idea of region inference. Observe(TE, φ) describes the regions that appear free in the type environment and result type. The effect φ of the expression includes these *observable* regions, as well as additional, non-observable regions. The idea behind the [*LETREGION*] rule is that non-observable regions in the effect set of an expression are purely local to the evaluation of the expression, in that no other part of the program will ever access them. Thus, these regions can be created immediately before the expression, and destroyed immediately after.

Another important aspect of the type inference rules is the use of effect variables ϵ. Effectively, these are used to implement a simple form of constraints on set variables using unification, in this case sets of atomic effects. To do so requires an extended notion of substitution of effect variables: $S(\epsilon.\varphi) = \epsilon'.(\varphi' \cup S(\varphi))$, where $\epsilon'.\varphi' = S(\epsilon)$. As a result of this modification, when multiple arrow effects are unified, the union is taken of all the atomic effects (φ).

A related property of effect variables is their use in describing dependence relationships among effect sets. For example, the effect of $(\lambda f.\lambda x.fx)\,h\,5$ depends on the arrow effect of h. This is reflected in the type for the expression $\lambda f.\lambda x.fx$:

$$(\alpha \xrightarrow{\epsilon.\emptyset} \beta, \rho) \xrightarrow{\epsilon''.\{\rho'\}} (\alpha \xrightarrow{\epsilon'.\{\epsilon,\rho\}} \beta, \rho')$$

One way of understanding ϵ variables is as a way of expressing constraints between effect sets. In fact, some previous work on effect systems used explicit systems of constraints [TJ92]. However, explicit constraints were not used in the Tofte/Talpin system because of a problem in conjunction with recursion. The `letrec` region inference rule expresses the recursion by matching the type assumptions for f (ocurring in the body) with the derived type for f. In the implementation of the region inference rules [TT94], this matching is accomplished by iterating until a fixpoint is reached. Such an implementation requires that testing equality of types can be done efficiently. Allowing arbitrary constraints to appear in types would defeat this. As it is, the type language is expressive enough that the necessary constraints can be specified, and simple enough that type equality can be tested efficiently.

If the language included recursive generic polymorphism, typing would be undecidable [Hen93]. Thus, in an expression of the form `letrec` $f(x) = e_1$ `in` e_2, the type variables in the type of f are quantified only within e_2, not in e_1. However, typing *is* still decidable in the presence of recursive region polymorphism, so region variables in the type of f are quantified within both e_1 and e_2. The introduction of recursive region polymorphism is one of the main technical advances of [TT94] and is essential for the quality of the results.

Here is an example of the translation. The source program, given below, simply counts down to zero in tail-recursive fashion. 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 difference of e_1 and e_2 in the region bound to ρ. Other arithmetic and comparison operations are defined analogously.

```
letrec f(x) =
    if x = 0 then x
        else f(x-1)
in
    f 100
end
```

The translation produces the target language program below:

```
letregion ρ1
in
    letrec
        f [ρ2] (x : (int, ρ2)) @ρ1 =
            if letregion ρ3
                in letregion ρ4 in (x=(0@ρ4))@ρ3 end
                end
            then x
            else
                letregion ρ5
                in
                    f [ρ2]@ρ5
                    (letregion ρ6 in (x-(1@ρ6))@ρ2 end)
                end
    in letregion ρ7 in f [ρ0]@ρ7 (100@ρ0) end
    end
end
```

2.5 Concrete semantics of target programs

The concrete semantics of target programs is very similar to the concrete semantics of fully annotated programs as given in Figure 4 (for an explanation of the notation, see Section 3. The only differences are in the [*LETREGION*] rule (the target language `letregion` actually corresponds to the [*LETREGION_TT*] rule in Figure 4), and the absence of [*ALLOCBEFORE*] and [*FREEAFTER*] rules, since these constructs are missing from the target language.

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

Example 2.1

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

In this program, nested `let` and `letregion` constructs are abbreviated.

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.

2.6 Storage mode analysis

The Tofte/Talpin system contains one more optimization after the region inference phase: storage mode analysis. The analysis is motivated by the fact that some programs have very poor memory utilization. In fact, the standard notion of "tail call optimization" is not expressible by the region inference translation alone. Thus, is is impossible that a program with arbitrary recursion depth can execute in constant space.

Storage mode annotation addresses this problem. It extends the existing store operation implicit in $e@\rho$ to two different kinds of store operations, known as `attop` and `atbot`. An expression of the form e `attop` ρ is operationally the same as before; the size of the region bound to ρ is increased by one, and the value is stored in the space thus allocated. However, e `atbot` ρ behaves differently; first, the region is reset, meaning that all values in the region are discarded. Then, the new value is stored in the region. After the operation, the region has a size of one storable value.

The purpose of storage mode analysis is to determine when the store can be replaced with the `atbot` annotation, and when it must remain `attop`. To do so, it uses a rather standard backwards flow algorithm, similar to globalization and single-threadedness analyses [Ses92, Fra91, Sch85].

A further goal of storage mode analysis is to allow region polymorphic functions to also be polymorphic in storage mode. This is accomplished by adding storage modes to region instantiation constructs, and allowing a third storage mode (`somewhereat`) for `letrec` bound region variables, indicating that the storage mode is to be one specified with that region variable when it is instantiated. In fact, the only storage modes permitted for `letrec` bound variables are `attop` and `somewhereat`; `atbot` is not a valid choice because it is always to instantiate the function in a region in which values written before the application would be used afterwards. At runtime, the storage modes are passed in as extra arguments to region polymorphic functions, as are the regions themselves.

There is no published account of storage mode analysis. Our knowledge of it is based on the prototype implementation provided to us by Mads Tofte, as well as a personal communication [Tof94].

As an example of the storage mode analysis, the result of the analysis applied to the countdown example of Section 2.4 is below:

$$
\begin{array}{l}
\texttt{letregion}\ \rho_1 \\
\texttt{in} \\
\quad \texttt{letrec} \\
\quad\quad f\,\texttt{[}\rho_2\texttt{]}\ (x : (\text{int}, \rho_2))\ \texttt{atbot}\ \rho_1 =
\end{array}
$$

```
            if letregion ρ3
                in letregion ρ4 in (x=(0 atbot ρ4)) atbot ρ3 end
                end
            then x
            else
                letregion ρ5
                in
                    f[sat ρ2] atbot ρ5
                    (letregion ρ6 in (x-(1 atbot ρ6)) sat ρ2 end)
                end
        in letregion ρ7 in f[atbot ρ0] atbot ρ7 (100 atbot ρ0) end
        end
    end
```

It is worth noting that all of the writes into region ρ_0 resolve to `atbot` annotations, either directly (as with the initial value of `100`), or indirectly through `somewhereat` (above abbreviated to `sat`) when the region is `letrec`-bound to variable ρ_2. Thus, in this case, storage mode annotation correctly optimizes the tail recursion of this region. Unfortunately, it does not optimize the regions storing the region instantiated closures for f. The Tofte/Talpin system contains an optimization by which the application and `letregion` constructs are combined so that the lifetimes of the new regions are limited to the evaluation of the function and the argument, but not of the function body itself. This optimization in conjunction with storage mode analyis does improve the results to constant space. We will have more to say about the quality of these results in Section 7.2.

3 Definitions

In this section, we develop our extensions to the Tofte/Talpin system. To the Tofte/Talpin target language described in Section 2.2 we add operations to allocate and free regions:

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

The operational semantics of this language derives facts of the form

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

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:

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

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 the last 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 4 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 operational semantics in Figure 4 differs from the semantics given for the source language in Figure 2 in several important respects. Most importantly, the notion of *store* is explicit. The representation of recursion is changed accordingly; instead of a special closure form for recursive functions, a standard closure is used, but with a cycle in the store. The operational semantics of Figure 4 are quite similar to those given in [TT94], for which it was proved that the translation specified by the region inference rules is sound. The semantics given in Figure 4 do differ somewhat from the Tofte/Talpin system, in that *unallocated* and *deallocated* states for regions, and extra allocation and deallocation expressions have been added. The correctness of our translation scheme with respect to these additions is the subject of Section 6.

The semantics in Figure 4 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.

4 Region-based 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*, an application of abstract interpretation [CC77], 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. 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:

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

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 $f[\rho_1, \rho_4]\ i$ is evaluated, and deallocated when $f[\rho_2, \rho_4]\ j$ 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 is not good enough. Knowing only that two region variables may be aliased would not permit allocation and deallocation operations on the 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 $\{\langle \lambda x.e' \ @\ \rho, R' \rangle\}$, 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 region-based closure analysis is given in Figure 5. 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 $\texttt{letrec}\ x[\vec{\rho}](y) =$, $\texttt{let}\ x =$, 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 min-

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

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

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

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

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

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

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

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

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

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

Figure 4: Operational semantics.

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

Figure 5: Region-based closure analysis.

imal 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 region-based 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.

4.1 Implementation

Our implementation of the extended closure analysis is fairly standard. Each of the constraints in Figure 5 can be expressed in the form of either simple set inclusion constraints of the form $S \subseteq T$, or conditional set inclusion constraints of the form "if value v is in set S, then $T \subseteq U$," where S, T, and U are all set variables. The form of the constraints suggests a worklist-based algorithm that iteratively refines a partial solution. Each step finds a constraint $S \subseteq T$ which is inconsistent, and replaces the value of T with $S \cup T$, thus satisfying that constraint. Whenever the guard of a conditional constraint becomes true, the set inclusion constraint is added to the worklist of constraints.

The actual implementation represents the constraints using a graph. Each node in the graph corresponds to an *(expression, region environment)* pair (e, R). Each node is associated with a set variable containing the (partial) result of evaluating $[\![e]\!]\ R$. An edge from (e, R) to (e', R') means that the abstract value $[\![e']\!]\ R'$ *depends on* $[\![e]\!]\ R$. This means that if the value of $[\![e]\!]\ R$ changes (i.e. becomes larger), then constraints of the form $S \subseteq [\![e']\!]\ R'$ may become inconsistent. Dependence edges encode direct dependencies (for each constraint of the form $[\![e]\!]\ R \subseteq [\![e']\!]\ R$ there is a dependence edge from (e, R) to (e', R'), and also indirect dependencies, for example those associated with conditional constraints. All such dependencies are represented; the absence of a dependency edge in the graph implies the absence of a dependency between the corresponding set variables.

We wish to avoid the overhead of storing the constraints explicitly. Thus, for all language constructs other than variables, the constraints are determined by syntactic examination of e' and its subexpressions. Variables, however, are subject to nonlocal dependencies.

One efficiency concern is the quadratic number of dependency edges from each binding of a variable's value (in applications) to each use of the variable. Our approach to eliminating the quadratic blowup is to introduce *variable nodes* into the graph. The graph contains one such node $(\mathbf{var}\ x, R)$ for each program variable x and each region environment R in which x is bound. An application of a closure $\{\langle \lambda x.e\ @\ \rho, R\rangle\}$ to an abstract value $[\![e_2]\!]\ R_2$ establishes the constraint $[\![\mathbf{var}\ x]\!]\ R \subseteq [\![e_2]\!]\ R_2$. Similarly, an occurrence of x in a region environment R' which is an extension of R (i.e. $R = R'|_{\mathit{Dom}(R)}$) establishes the constraint $[\![x]\!]\ R' \subseteq [\![\mathbf{var}\ x]\!]\ R$. The latter constraints are not syntactically apparent from the expression x. They can, however, be determined from the dependence edges leading into (x, R'). Thus the rule: constraints on variable occurrences are derived from the in-edges in the dependence graph; for all other expressions, the constraints are derived from the syntax of the expression.

A worklist keeps track of which constraints are inconsistent. Specifically, the worklist holds nodes n such that constraints between n and n's successor nodes may be inconsistent. For any node n not in the worklist, all constraints between n and n's successors are satisfied (this is the invariant of the algorithm). The nodes are also marked with one of `WORK` or `IDLE` to denote their membership in the worklist, so that adding a node to the worklist without duplication is a constant time operation.

The worklist step chooses a node on the worklist, removes it from the worklist, determines the constraints associated with that node using the rule above, and if necessary updates the abstract value to satisfy the constraints. If the abstract value changes, all successor nodes are added to the worklist.

When the worklist becomes empty, the algorithm terminates. From the invariant, it holds that if there are no nodes on the worklist, then all constraints are satisfied. Further, termination is guaranteed, because each step increases the size of the partial solution, and the space of solutions is finite.

The implementation of the closure analysis is about 1000 lines of SML/NJ code, not counting support code to translate data structures used in the Tofte/Talpin system.

5 Completions

This section first outlines the constraint language and constraint generation. The following subsections deal with finite abstractions for regions, constraint generation, and constraint resolution in detail.

We want to characterize the set of legal completions of a program using a constraint system. The variables in this system are the states (unallocated U, allocated A, or deallocated D) of regions at program points in the source program. These variables are referred to as *state variables* and written $t, t_1, t_2, \ldots$. The constraint system for a particular program must encode

1. at which program points particular regions must be in the allocated state,
2. the flow of regions along the program's execution paths.

Both points are necessary to guarantee that a region is actually allocated wherever it is accessed. Encoding the control flow is necessary to ensure that the state of a region is consistent over time and changes only at explicit allocation and deallocation constructs. Closure analysis provides an approximation to the control flow in higher order functions. Region states can thus be tracked along control flow paths by suitably constraining the state variables between successive program points. However, the main problems in constraint generation are to find a suitable abstraction for dynamic regions and to define which regions' state is modeled by a particular state variable. An abstraction of dynamic regions is required because programs may be recursive and their execution may produce an infinite number of distinct regions. Section 5.1 develops an abstraction for dynamic regions and region environments.

Given a suitable definition of the set of regions described by a single state variable, the following three kinds of constraints are sufficient to encode points 1 and 2 above: (1) *allocation constraints*, (2) *choice constraints*, and (3) *equality constraints*:

$$
\begin{array}{ll}
t = A & (1) \\
\langle t_1, c_p, t_2 \rangle_a & \multirow{2}{*}{(2)} \\
\langle t_1, c_p, t_2 \rangle_d & \\
t_1 = t_2 & (3)
\end{array}
$$

At program points where values are read from or written to regions, allocation constraints are placed on state variables abstracting those regions; they express that a region must be allocated at this point. Choice constraints are used to connect state variables at consecutive program points where a region may change state. Choice constraints are either *allocation triples* or *deallocation triples*.

Definition 5.1 An allocation triple expresses a relationship between two state variables t_1, t_2 and a boolean variable c_p associated with program point p:

$$\langle t_1, c_p, t_2 \rangle_a \stackrel{\text{def}}{\equiv} (c_p \iff (t_1 = U \wedge t_2 = A)) \wedge (\neg c_p \iff t_1 = t_2)$$

The boolean c_p encodes whether or not the associated region is to be allocated at program point p. If $c_p = \mathit{true}$ the region state prior to the allocation point is U and afterwards A, i.e. allocation. If $c_p = \mathit{false}$, 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:

Definition 5.2 A deallocation triple expresses a relationship between two state variables t_1, t_2 and a boolean variable c_p associated with program point p:

$$\langle t_1, c_p, t_2 \rangle_d \stackrel{\text{def}}{\equiv} (c_p \iff (t_1 = A \wedge t_2 = D)) \wedge (\neg c_p \iff t_1 = t_2)$$

Equality constraints are used to constrain state variables to the same state, e.g. at consecutive program points, where no allocation/deallocation is possible.

Constraints are generated as a function of the target program structure, the Tofte/Talpin types and effects of the program, and a refinement of the region-based closure analysis (Section 4). Before presenting the details of constraint generation, we define an abstraction for dynamic regions in Section 5.1. Section 5.2 contains further notation and definitions, Section 5.3 contains the constraint generation rules, and Section 5.4 discusses constraint resolution.

5.1 Abstracting dynamic Regions

This section develops a finite abstraction for dynamic regions by successively refining a naive approach.

5.1.1 Naive approach

Summary All regions created by a particular `letregion` construct are abstracted by a single abstract region. The problem with this approach is that it does not distinguish enough regions to make insertion of any allocation/deallocation constructs safe.

In the naive approach, dynamic regions are abstracted by *colors*, one color per syntactic `letregion`. Because of recursion, a color represents the (possibly infinite) set of dynamic regions produced by the particular `letregion` statement it is associated with. Encoding the flow of regions in the constraint system is done by associating with each program point and each color a single state variable. State variables of consecutive program points are constrained with equalities or choice constraints, according to color.

Although intuitive, this approach does not allow us to infer where regions can be allocated or deallocated. The difficulty stems from the fact that each color represents a *set* of dynamic regions. For example, if in a particular execution context, two region variables ρ_1 and ρ_2 bind two distinct regions z_1 and z_2, but the abstraction uses a single color c to denote z_1 and z_2, then the abstraction cannot model the states of regions z_1, z_2 before and after a construct like `alloc_before` ρ_1 with a single state per program point. The region z_1 must be in the unallocated state before the construct, and in the allocated state after. If the abstraction captures the states of z_1, then it is wrong for z_2 (z_2's state is not affected by `alloc_before` ρ_1), and vice versa.

In general, at a program point p, allocation or deallocation constructs can only be inserted on region variables whose color unambiguously denotes a *single* unknown dynamic region in the context of p. This is of course impossible to achieve for all program points, due to the unbounded number of dynamic regions. However, below we explore an abstraction that is able to assert the singleton requirement at some program points. In the remainder of this report, region abstractions are always termed colors.

5.1.2 Abstract region environments

Summary Instead of abstracting individual regions, region environments are abstracted as a whole, resulting in a closer correspondence between abstract and concrete regions. The remaining problem is the handling of quantified effect variables.

In the naive approach above, the state of every dynamic region is apparent at every program point. This over-specification is the source of the problem that a single color abstracts many distinct dynamic regions. The problem is overcome by expressing facts only about dynamic regions that actually matter at a program point in a particular context. Regions are accessed through region variables appearing in the target program. Region variables are mapped to regions through a region environment as shown in the operational semantics in Figure 4. It seems that the evaluation of an expression can therefore only access regions that are bound in the current region environment. This observation is not entirely true, but it can be refined later. For the moment, we assume that evaluation of an expression in a region environment r needs only be concerned with the state of regions appearing in r.

Defining the region or set of regions abstracted by a color becomes easier in the context of an entire abstract region environment. An obvious choice is to say that if an abstract region environment R abstracts a concrete region environment r, a color c bound to a region variable ρ in R abstracts exactly the region bound to ρ in r. However, more than one region variable may bind the same color. In this case it is no longer clear which region the color abstracts, and how many distinct regions there are. This problem is solved by adding a consistency constraint between abstract and concrete region environments, expressing that each color in an abstract region environment denotes a unique region in the concrete region environment, and vice versa:

$$R(\rho) = R(\rho') \iff r(\rho) = r(\rho')$$

We call this relation $r \;\mathsf{sat}\; R$, and define it more precisely as:

$$\begin{array}{rcl} r \;\mathsf{sat}\; R & \stackrel{\text{def}}{\equiv} & \mathit{Dom}(r) = \mathit{Dom}(R) \;\wedge \\ & & \forall (\rho, \rho' \in \mathit{Dom}(r))\; R(\rho) = R(\rho') \iff r(\rho) = r(\rho') \end{array}$$

The implication $R(\rho) = R(\rho') \implies r(\rho) = r(\rho')$ says that each color denotes a single dynamic region in the region environment r. This fact alone is not sufficient to reason about the states of all regions in the region environment after e.g. an `alloc_before` ρ construct. If two region variables ρ_1 and ρ_2 are aliased (bind the same region), an allocation on ρ_1 implies an allocation of the region bound to ρ_2 and vice versa. The symmetry of the $r \;\mathsf{sat}\; R$ relation, however, is strong enough to capture aliasing between region variables, since the aliasing is the same in the abstraction. Abstract region environments are computed by the region-based closure analysis given in Section 4.

We now describe how state variables are associated with program points. Consider a program point p, and the smallest syntactic expression e that contains p. Let φ be the effects inferred by the Tofte-Talpin effect inference for expression e in the entire program. Given a region environment, the effect set φ characterizes the regions that are used by the evaluation of e. From the region-based closure analysis, we can compute a set of abstract region environments $\{R_1, \dots, R_n\}$ that are possible at p. For each such abstract region environment R_i, we associate a state variable with each color in the range of R_i and mapped by some region ρ in φ. Thus, not only can we distinguish between the states of a region occurring in two distinct region environments, but we also restrict our state variables to the set needed to express the states of regions accessed during the evaluation of e.

Compared to the initial naive approach, there is an important difference: the state variable associated with color c mapped by region variable ρ in region environment R, stands only for the state of the dynamic region mapped by ρ in a corresponding concrete region environment, but not for the states of regions abstracted by the same color c in unrelated region environments.

However, as hinted at above, the set of regions accessed by an expression may include regions not mapped by region variables! Effect sets may contain effect variables. Effect variables are introduced by effect quantification in `let` and `letrec` expressions, and they express that an expression may access different sets of regions in different contexts (Section 2.2). As a result, if region environments do not contain mappings for effect variables, the set of regions that an effect refers to is only partially known; i.e. the regions (or abstractions thereof) denoted by effect variables are not known. The following partially annotated example illustrates the problem:

Example 5.3
```
let app1 = λf.f 1    (*)
in
    letregion ρ1
    in
        let a = 2@ρ1
        in
            app1 (λy.y+a)    (**)
        end
    end
end
```

Assume the quantified type of the *app1* function is $\forall \epsilon_1 \epsilon_2.(int \xrightarrow{\epsilon_1.\emptyset} int) \xrightarrow{\epsilon_2.\{\epsilon_1\}} int$ (region variables have been omitted for clarity). Suppose the color associated with region variable ρ_1 is c_1 in a particular context. The flow of this region from the beginning of the `letregion` expression to the application of *app1* at (**) is easy. The *latent effect set* (effect of evaluating the function body) of λy includes ρ_1 because it is accessed when the variable a is used in the addition. The instantiated type of *app1* at the application is therefore $(int \xrightarrow{\{\rho_1,\dots\}} int) \xrightarrow{\{\rho_1,\dots\}} int$ (again most region variables have been omitted for clarity). If we now look at the latent effect set at the application of *app1* at (**) and the latent effect set of the callee λf, we see that they differ. In particular, the set of colors obtained by pointwise application of the region environment to the instantiated effect set at (**) differs from the set obtained from the quantified effect set and the region environment captured at (*). Color c_1 does not appear in the latent effect set of the quantified type of *app1* at (*). There is no state variable inside the *app1* function that we could connect to from the caller. The inverse problem appears at the application of f `1` at (*). The latent effect at the application is ϵ_1, but the latent effect of the function λy, bound to f, is $\{\rho_1, \dots\}$. In essence, the handle on c_1 (through a region variable) is lost at program points in the body of *app1*, because *app1* is polymorphic in that effect.

In order to produce a constraint system that characterizes only legal completions, we need to track at all program points the states of regions bound only to effect variables, as well as the states of regions mapped by region variables. Section 5.1.3 extends the notion of a region environment to include mappings from effect variables to sets of regions. Such mappings are obtained through effect variable instantiation.

5.1.3 Effect variable instantiation

Summary Making the mappings of effect variables to sets of regions explicit through effect instantiation guarantees that all regions accessed during evaluation of an expression are bound to region or effect variables on all control flow paths. However, the cardinality of the set of regions bound to an effect variable may be unbounded.

What happens to the region bound to ρ_1 in Example 5.3 where the function *app1* is called? As described in the previous subsection, in the context of the application at (**) in Example 5.3, the effect variable ϵ_1 is instantiated to $\{\rho_1, \dots\}$. The region bound to ρ_1 does therefore not really disappear at program points inside the *app1* function. It is part of the set of regions bound to the effect variable ϵ_1. Effect variable instantiations are computed by the Tofte/Talpin system during effect inference, but target programs are not annotated with these instantiations because effect variable instantiations do not affect the evaluation of programs. For our analysis however, explicit effect variable instantiations are necessary. We therefore extend the Tofte/Talpin target language slightly:

$$\begin{array}{rcl} e & ::= & \cdots \\ & | & \texttt{letrec}\ f[\vec{\rho}, \vec{\epsilon.\varphi}](x)@\rho = e_1\ \texttt{in}\ e_2\ \texttt{end} \\ & | & f[\vec{\rho}\,', \vec{\varphi}\,']@\rho \end{array}$$

For purposes of presentation we only consider effect variable quantification at `letrec` constructs. Effect variable quantification at `let` constructs is handled similarly. Analogous to quantified region variables, quantified effect variables at `letrec` constructs become formal parameters ($\vec{\epsilon.\varphi}$). The parameter names are represented as arrow-effects $\epsilon.\varphi$, because the known part of the effect (φ) is required for the definition of how to compute instantiations (discussed below). At region application, effect variables are instantiated to sets of regions through effect parameters ($\vec{\varphi}\,'$). The *scope* of an effect variable is the function body e_1 bound by the `letrec`-definition where the effect variable is quantified.

The following paragraphs show how the set of regions to be bound to an effect variable at region application is computed. The goal is to extend the region environment of a closure obtained at a region application with mappings from effect variables to sets of regions. Consider a letrec-binding `letrec` $f[\vec{\rho}, \vec{\epsilon.\varphi}]\ldots$ in region environment r_0 and a region application $f[\vec{\rho}\,', \vec{\varphi}\,']$ in region environment r'. In the [REGAPP] rule of the operational semantics in Figure 4, the closure resulting from the region application contains a region environment r_1 obtained by extending r_0 with mappings for the region parameters ρ_i (*i*th component of vector $\vec{\rho}$):

$$r_1 = r_0[\rho_i \leftarrow r'(\rho_i')]$$

We further extend the region environment r_1 with mappings for the formal effect parameters (ϵ_i) to sets of regions (d_i) computed as a function of r_1, φ_i and r', φ_i':

$$r_2 = r_1[\epsilon_i \leftarrow d_i]$$

The sets d_i are computed such that the set of regions denoted by the arrow-effect $\epsilon_i.\varphi_i$ in region environment r_2 is equal to the set of regions denoted by the effect parameter φ_i' in region environment r':

$$d_i \cup r_1(\varphi_i) = r'(\varphi_i') \qquad \text{where } r_2(\epsilon_i.\varphi_i) = r_2(\epsilon_i) \cup r_2(\varphi_i) = d_i \cup r_1(\varphi_i)$$

To satisfy the equality, the set d_i is constrained by the following upper and lower bound (the safety of Tofte/Talpin typings guarantees that $r_1(\varphi_i) \subseteq r'(\varphi_i')$):

$$r'(\varphi_i') - r_1(\varphi_i) \subseteq d_i \subseteq r'(\varphi_i')$$

At this point it doesn't really matter how we choose d_i within the specified bounds. The exact way to compute the sets d_i depends on the abstraction chosen for dynamic regions.

We set out to compute effect variable mappings for region environments in order to maintain the following invariant: all regions accessed by the evaluation of an expression e in region environment r are bound in r by region or effect variables. Example 5.3 showed why this invariant is required to model the state of regions along all control flow paths in a program.

Unfortunately, abstracting region environments with effect variable mappings is non-trivial. Section 5.1.1 showed that in order to add an allocation/deallocation construct for a region variable ρ at a particular program point *p*, a state variable modelling exclusively the state of the region bound to ρ at *p* is required. This requirement motivated the r `sat` R relation of Section 5.1.2. Extending abstract region environments and the r `sat` R relation with effect variables, such that each color still denotes a single dynamic region, may require an unbounded number of colors for some programs because the number of regions bound to a single effect variable may be infinite. To appreciate this fact, consider the following example (only relevant annotations are shown):

Example 5.4
```
letrec fac(n) = λg.
    if n = 0 then g 1
    else
        letregion ρ1 in
            fac (n-1) (λx.g (x*n))@ρ1
        end
in
    fac 10 (λx.x*1)
end
```

The program defines the factorial function in "continuation passing style". Consider the continuation parameter g and its (simplified) type: *int* $\xrightarrow{\epsilon_1}$ *int*. The effect variable ϵ_1 is quantified by the `letrec` construct and stands for the set of regions that the continuation accesses when executed. At the recursive call to *fac*, a new instance of ϵ_1 is instantiated (made possible by recursive polymorphism). The instantiation is the latent effect of the lambda expression λx, which includes ϵ_1 due to the call to g. It also contains other regions, namely ρ_1 holding the closure of λx. Since, the region bound to ρ_1 is local to the body of *fac*, the number of regions in the latent effect of the continuation grows at each iteration.

5.1.4 Abstract region environments with effect variables

Summary This subsection describes an abstraction of region environments that requires only a finite number of colors for any program. This abstraction is the one used for constraint generation.

To keep the number of required colors finite, we choose an abstraction of region environments that models may-aliasing of regions bound to effect variables w.r.t. other regions. For regions bound solely to region variables, the aliasing information is perfect. In terms of colors this means that a color in the set bound to an effect variable denotes a set of regions (defined more precisely below), whereas a color mapped solely by region variables denotes the single region bound by the same region variables in the concrete region environment.

Colors for region variables in `letregion`-constructs are chosen according to the exact same rules described in Section 5.1.2: the color for the new region variable is different from any colors already bound to region variables in the current region environment. Consequently, the same finiteness argument applies, and aliasing between region variables is perfectly modelled by the aliasing of colors. In order to define the refined $r \;\mathsf{sat}\; R$ relation, we assume that each dynamic region is annotated with a color (written as subscript) chosen according to the same rule as in the abstraction. The consistency relation between abstract and concrete region environments is defined by:

$$\begin{array}{rcl} r \;\mathsf{sat}\; R & \stackrel{\text{def}}{\equiv} & \mathit{Dom}(r) = \mathit{Dom}(R) \;\wedge \\ & & \forall(\rho, \rho' \in \mathit{Dom}(r))\; R(\rho) = R(\rho') \iff r(\rho) = r(\rho') \;\wedge \\ & & \forall(\rho \in \mathit{Dom}(r))\; R(\rho) = \mathit{color}(r(\rho)) \;\wedge \\ & & \forall(\epsilon \in \mathit{Dom}(r))\; R(\epsilon) = \mathit{colors}(r(\epsilon)) \end{array}$$

where *color* is a function from dynamic regions the their color annotation, and *colors* is the set extension of *color*. The first two clauses are unchanged. The two new clauses require that the color annotation of dynamic regions correspond to the colors used to abstract those dynamic regions.

The consistency relation between abstract and concrete region environments leads to the following definition of what a color abstracts: (a formal statement can be found in Definition 6.6)

> A state variable associated with a color c, program point p, and abstract region environment R represents the state of all regions z_c colored by c (bound to region or effect variables) that may appear at program point p in a concrete region environment abstracted by R.

A color may now abstract multiple regions as in the naive approach of Section 5.1.1. In contrast to the naive approach however, the structure of an abstract region environment states which colors abstract a single dynamic region, and which colors potentially abstract more than one dynamic region. A color mapped solely by region variables (not contained in the mapping of any effect variable) abstracts a single dynamic region. If a region variable maps to such a color, potential allocation or deallocation constructs for that region variable can be modelled by the constraints. On the other hand, if a region variable maps to a color that appears in any effect variable mappings, allocation/deallocation on that region variable cannot be expressed in the constraints.

In order to find good program completions, we prefer to have as many potential allocation/deallocation choice points for a program as possible. As spelled out in the previous paragraph, the sets of colors mapped by effect variables has an influence on the potential choice points. The reminder of this subsection describes how to compute minimal color sets for effect variable instantiations in abstract region environments.

Instantiations of effect variables to sets of colors are computed analogous to the sets of concrete regions described in Section 5.1.3. Given a letrec-binding `letrec` $f[\vec{\rho}, \epsilon.\vec{\varphi}]\ldots$ in abstract region environment R_0 and a region application $f[\vec{\rho}\,', \vec{\varphi}\,']$ in abstract region environment R', the abstract closure resulting from the region application contains a region environment that is obtained by extending R_0 with mappings for the region parameters (giving R_1) and further extending R_1 with mappings for effect variables:

$$\begin{array}{rcl} R_1 & = & R_0[\rho_i \leftarrow R'(\rho_i')] \\ R_2 & = & R_1[\epsilon_i \leftarrow D_i] \end{array}$$

The sets of colors D_i are computed as a function of R_1, φ_i and R', φ_i' such that the set of colors for the arrow-effect $\epsilon_i.\varphi_i$ in abstract region environment R_2 is equal to the set of colors of φ_i' in abstract region environment R':

$$D_i \cup R_1(\varphi_i) = R'(\varphi_i') \qquad \text{where } R_2(\epsilon_i.\varphi_i) = R_2(\epsilon_i) \cup R_2(\varphi_i) = D_i \cup R_1(\varphi_i)$$

As is the case for concrete regions, the equality is satisfied if D_i lies within the following bounds:

$$R'(\varphi_i') - R_1(\varphi_i) \;\subseteq\; D_i \;\subseteq\; R'(\varphi_i')$$

The lower bound is obviously the preferred choice for our instantiation, since it adds the minimal number of colors to effect variable mappings. However, the abstract and concrete region environments R_2, r_2 after instantiation have to be consistent with respect to the r_2 sat R_2 relation. Choosing the lower bound for D_i and d_i does not result in consistent region environments as shown by the following example (colors are letters a, b):

effect formal parameter	$\epsilon_1.\{\rho_1, \rho_2\}$
concrete region environment	$r_1 = [\rho_1 \leftarrow z_a, \rho_2 \leftarrow z'_b]$
abstract region environment	$R_1 = [\rho_1 \leftarrow a, \rho_2 \leftarrow b]$
effect argument	$\varphi'_1 = \{\rho_1, \rho_2, \epsilon_2\}$
concrete region environment	$r' = [\rho_1 \leftarrow z_a, \rho_2 \leftarrow z'_b, \epsilon_2 \leftarrow \{z''_a\}]$
abstract region environment	$R' = [\rho_1 \leftarrow a, \rho_2 \leftarrow b, \epsilon_2 \leftarrow \{a\}]$

As before, region environments R_1, r_1 are obtained by extending R_0, r_0 captured at the `letrec`-definition, and R', r' are the region environments at the region application. Using the lower bound for D_1, d_1 and assuming that z_a and z''_a are distinct, we obtain

$$\begin{array}{rcl} R_2 & = & R_1[\epsilon_1 \leftarrow \{\}] \\ r_2 & = & r_1[\epsilon_1 \leftarrow \{z''_a\}] \end{array}$$

which clearly doesn't satisfy r_2 sat R_2, even though r_1 sat R_1 and r' sat R'. The color set of ϵ_1 in r_2 is $\{a\}$, whereas in R_2 the set is empty.

Failing to include color a in the mapping because it is already bound to ρ_1 is wrong because there are two distinct regions with color a in the effect argument, whereas ρ_1 binds only a single region. Whenever a region environment contains multiple distinct regions with the same color, all but one of those regions must appear in mappings of effect variables. This observation follows from the $R(\rho) = R(\rho') \iff r(\rho) = r(\rho')$ clause in the definition of r sat R. Since we know which colors potentially denote multiple regions, the sets D_i, d_i can be computed as follows:

$$\begin{array}{rcl} d_i & = & r'(\varphi'_i) - \{r_1(\rho) \mid \rho \in \varphi_i\} \cup \bigcup_{\epsilon \in \varphi'_i} r'(\epsilon) \\ D_i & = & R'(\varphi'_i) - \{R_1(\rho) \mid \rho \in \varphi_i\} \cup \bigcup_{\epsilon \in \varphi'_i} R'(\epsilon) \end{array} \tag{1}$$

By adding in all colors (resp. regions) bound to effect variables in φ'_i, we guarantee that if there are multiple distinct regions with the same color annotation in the effect argument, the instantiation retains this fact. The soundness of equation (1) is proven in Section 6 in the [REGAPP] case.

5.1.5 Discussion

Computing effect variable instantiations during the extended closure analysis is expensive, since it may lead to a combinatorial explosion of the number of distinct region environments.

A cheaper alternative is to infer a single mapping for every quantified effect variable appearing in the type derivation, independent of its context. Such mappings can be found quite easily through a global system of subset constraints using equation (1) at every instantiation. This cheaper approach computes for each effect variable, the union of instantiation maps over all contexts. Such an approximation is still safe, but leads to fewer potential allocation/deallocation points.

Yet a third approach avoids computing effect variable instantiations altogether, by being more conservative. At applications with potentially non-empty effect variable instantiations, all regions in the caller and callee can be constrained to be allocated without matching up individual regions. Thus, whenever regions disappear from the region environment due to effect variables, these regions are in the allocated state.

It turns out that it is easiest to prove the system correct with explicit effect variable instantiation. Section 6 shows that this approach is sound. Section 7 discusses the approaches used in the actual implementation.

The reminder of Section 5 is organized as follows. The region-based closure analysis based on explicit effect variable instantiations is given in Section 5.2. Section 5.3 describes the constraint generation rules. Finally, Section 5.4 contains the resolution algorithm.

5.2 Definitions

This section introduces notation used in the region-based closure analysis and the constraint generation rules. We overload notation for applying abstract region environments to region variables $R(\rho)$, effect variables $R(\epsilon)$, sets of both $R(\varphi)$, and arrow-

$$
\begin{array}{rcl}
 & \vdots & \\
[\![\lambda x.e : \mu]\!]\; R & = & \{\langle \lambda x.e{:}\mu, R\rangle\} \\
[\![\mathtt{letrec}\; f[\vec{\rho}, \epsilon.\vec{\varphi}\,](x) \mathbin{@} \rho = e_1 \;\mathtt{in}\; e_2]\!]\; R & = & [\![e_2]\!]\; R \\
 & & \{\langle \lambda x.e_1{:}\mu, \vec{\rho}, \epsilon.\vec{\varphi}, R\rangle\} = [\![f]\!]\; R \\
[\![f[\vec{\rho}, \vec{\varphi}\,] \mathbin{@} \rho]\!]\; R & = & \{\langle \lambda x.e{:}\mu, R'[\rho'_i \leftarrow R(\rho_i), \epsilon'_i \leftarrow R|_\rho(\varphi_i) - R'|_\rho(\varphi'_i) \cup R|_\epsilon(\varphi_i)]\rangle\} \\
 & & \text{where } [\![f]\!]\; R = \{\langle \lambda x.e{:}\mu, \vec{\rho'}, \epsilon'.\vec{\varphi'}, R'\rangle\}
\end{array}
$$

Figure 6: Changes to the region-based closure analysis (explicit effect variable instantiation).

effects $R(\epsilon.\varphi)$. The exact form should always be clear from context:

$$
\begin{array}{rcl}
R(\varphi) & = & \{c \mid \exists \rho \in \varphi \text{ s.t. } R(\rho) = c\} \cup \bigcup_{\epsilon \in \varphi} R(\epsilon) \\
R(\epsilon.\varphi) & = & R(\varphi \cup \{\epsilon\})
\end{array}
$$

Restricted domain and range functions separate region and effect variables. $\mathit{Dom}|_\rho(R)$ is the set of region variables in the domain of the region environment R, and $\mathit{Dom}|_\epsilon(R)$ is the set of effect variables in the domain of R. The range functions are defined analogously.

$$
\begin{array}{rcl}
\mathit{Dom}|_\rho(R) & = & \{\rho \mid \rho \in \mathit{Dom}(R) \wedge \rho \in \text{RegVar}\} \\
\mathit{Dom}|_\epsilon(R) & = & \{\epsilon \mid \epsilon \in \mathit{Dom}(R) \wedge \epsilon \in \text{EffVar}\} \\
\mathit{Range}|_\rho(R) & = & \{R(\rho) \mid \rho \in \mathit{Dom}|_\rho(R)\} \\
\mathit{Range}|_\epsilon(R) & = & \bigcup\{R(\epsilon) \mid \epsilon \in \mathit{Dom}|_\epsilon(R)\}
\end{array}
$$

We also make use of restricted mappings $R|_\rho(\varphi)$ and $R|_\epsilon(\varphi)$ to map only region or effect variables in φ:

$$
\begin{array}{rcl}
R|_\rho(\varphi) & = & \{R(\rho) \mid \rho \in (\varphi \cap \mathit{Dom}|_\rho(R))\} \\
R|_\epsilon(\varphi) & = & \bigcup\{R(\epsilon) \mid \epsilon \in (\varphi \cap \mathit{Dom}|_\epsilon(R))\}
\end{array}
$$

As motivated in Section 5.1, the region-based closure analysis needs to be extended to keep track of effect variables. Figure 6 shows the changes to the rules in Figure 5. Effect instantiation is explicit in the rules for `letrec` and region instantiation. Effect variable instantiations are computed as described in Section 5.1.4. The types of lambda abstractions are included in the abstract closures. This information is needed to produce constraints for applications. The `letregion` rule is unchanged, i.e. the color bound to the newly introduced region variable is chosen to be different from all colors already bound to region variables, but irrespective of the sets of colors bound to effect variables.

5.3 Constraint Generation

As motivated in Section 5.1, with each program point having effect set φ, abstract region environment R, and color c in $R(\varphi)$ is associated a *state variable* ranging over $\{U, A, D\}$ (unallocated, allocated, deallocated). State variables are grouped together into *state vectors* $S^{\text{in}}_{e,R}$, $S^{\text{out}}_{e,R}$, and $S^{i}_{e,R}$ associated with an expression e and region environment R. Vectors $S^{\text{in}}_{e,R}$ and $S^{\text{out}}_{e,R}$ represent the state variables of the first (resp. last) program point of expression e, whereas vectors $S^{i}_{e,R}$ represent state variables of internal program points of e (e.g. between two subexpressions e_1 and e_2 of e). 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. The three kinds of constraints introduced at the beginning of Section 5 are

$$
\begin{array}{ll}
t = A & \text{(Allocation constraint)} \\
\langle t_1, c_p, t_2\rangle_a & \\
\langle t_1, c_p, t_2\rangle_d & \text{(Choice constraints)} \\
t_1 = t_2 & \text{(Equality constraint)}
\end{array}
$$

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, and linking states between application contexts and function bodies. Choice constraints are introduced at possible allocation or deallocation points and link the region states before and after the choice point.

A program point p is a possible allocation/deallocation point for a region variable ρ, if ρ appears in the effect set at that program point and if in every abstract region environment R at p, the color bound to ρ does not appear in the effect variable mappings of R. The former condition implies that regions do not change state if they don't appear in the Tofte/Talpin effects. The latter condition is made explicit in the constraint generation to ease the proof.

Potential allocation and deallocation points are indicated by the syntax `alloc_before` $\rho\ c_p\ e$ and `free_before` $\rho\ c_p\ e$, where c_p is the boolean variable associated with the allocation (resp. deallocation) point. Prior to constraint generation, all potential `alloc_after`, `alloc_before`, `free_after`, `free_before`, `free_app` expressions are added to the Tofte/Talpin target program. We treat only `alloc_before` and `free_after` formally. The other constructs can be handled similarly. The `free_app` construct is a combination of an application and a deallocation.

The following paragraphs explain the constraint generation rules in Figure 7. 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 effect set φ of e (inferred by Tofte/Talpin effect inference). The rule for variables says that no regions are accessed and thus no constraints are needed.

In the abstraction rule, we place an allocation constraint on the region where the closure is written. No other regions are accessed by this expression.

For region instantiation, we place an allocation constraint on the region holding the polymorphic closure and on the region where the instantiated closure is written. No other regions are accessed.

The handling of `let` is straightforward. The relevant regions for e_1 are connected between the entry points of e and the intermediate program point between e_1 and e_2. Regions not relevant to e_1 or which cannot change state are connected directly from the entry of e to the intermediate program point. The reasoning for e_2 is analogous.

In the rule for `letrec`, an allocation constraint is placed on the region where the polymorphic closure is written. The relevant regions for e_1 are connected between e's input and output points.

The color d in the `letregion` rule is chosen according to the same rules as in the region-based closure analysis. The boolean variable c_e encodes whether the newly created region starts and ends in the allocated state or whether it is allocated and deallocated within e_1. This irregularity arises due to our finite abstraction of regions. Equally colored regions in a region environment must all be in the same state. The color for a new region at a `letregion`-construct is chosen to be different from all colors bound to region variables, but irrespective of colors appearing in effect variable mappings. As a result, the new region may be given a color that already appears in an effect variable mapping. In such a case, the new region cannot be start in the **U** state and end in the **D** state, because the region in the effect variable mapping shares these states. The only consistent states in that case are for the regions to be allocated.

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 variable. This setup 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 ($R(\varphi_b)$) of the function expression e_1, are the same as the regions in the effect of the closure ($R_0(\varphi'_b)$). Therefore, the states of regions in $R(\varphi_b)$ prior to evaluation of the function body match the states of regions in $R_0(\varphi'_b)$ 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 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 $R(\varphi_b)$. Other regions in the caller's context 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 region-based closure analysis.

5.4 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 solution.

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

$$
\begin{array}{rll}
e = x & R,\emptyset & \rightarrow \ \textit{no constraints}
\end{array}
$$

$$
\begin{array}{rll}
e = \lambda x.e_1 @ \rho & R,\{\rho\} & \rightarrow \ S^{\text{in}}_{e,R}[R(\rho)] = \mathbf{A} \\
 & & \ S^{\text{in}}_{e,R}[R(\rho)] = S^{\text{out}}_{e,R}[R(\rho)]
\end{array}
$$

$$
\begin{array}{rll}
e = f[\vec{\rho}\,', \vec{\varphi}\,']@\rho' & R,\varphi & \rightarrow \ S^{\text{in}}_{e,R}[R(\rho')] = A \\
 & & \ S^{\text{in}}_{e,R}[R(\rho)] = A, \text{ where } (\tau,\rho) \text{ is the type of } f \\
 & & \ \forall c \in R(\varphi).\ S^{\text{in}}_{e,R}[c] = S^{\text{out}}_{e,R}[c]
\end{array}
$$

$$
\begin{array}{rll}
e = e_1\, e_2 & R,\varphi & \rightarrow \ \text{Let } \varphi_1, \varphi_2 \text{ be the effect sets of } e_1, e_2 \\
 & & \ \forall (c \in R(\varphi_1))\ S^{\text{in}}_{e,R}[c] = S^{\text{in}}_{e_1,R}[c] \ \wedge \ S^{\text{out}}_{e_1,R}[c] = S^{1}_{e}R[c] \\
 & & \ \forall (c \in R(\varphi) - R(\varphi_1) \cup R|_{\epsilon}(\varphi))\ S^{\text{in}}_{e,R}[c] = S^{1}_{e,R}[c] \\
 & & \ \forall (c \in R(\varphi_2))\ S^{1}_{e,R}[c] = S^{\text{in}}_{e_2,R}[c] \ \wedge \ S^{\text{out}}_{e_2,R}[c] = S^{2}_{e,R} \\
 & & \ \forall (c \in R(\varphi) - R(\varphi_2) \cup R|_{\epsilon}(\varphi))\ S^{1}_{e,R}[c] = S^{2}_{e,R}[c] \\
 & & \ \text{Let } (\mu_1 \xrightarrow{\epsilon.\varphi_b} \mu_2, \rho) \text{ be the type of } e_1 \\
 & & \ S^{2}_{e,R}[R(\rho)] = \mathbf{A} \\
 & & \ \text{For all } \langle \lambda x.e_b, R_0 \rangle \in [\![e_1]\!]\ R, \text{ with type } \mu_1' \xrightarrow{\epsilon'.\varphi_b'} \mu_2' \\
 & & \quad \forall (c \in R(\varphi_b))\ S^{2}_{e,R}[c] = S^{\text{in}}_{e_b,R_0}[c] \ \wedge \ S^{\text{out}}_{e_b,R_0}[c] = S^{\text{out}}_{e,R}[c] \\
 & & \quad \forall (c \in R_0|_{\epsilon}(\varphi_b'))\ S^{\text{in}}_{e_b,R_0}[c] = \mathbf{A} \\
 & & \quad \forall (c \in R(\varphi) - R(\varphi_b) \cup R|_{\epsilon}(\varphi))\ S^{2}_{e,R}[c] = S^{\text{out}}_{e,R}[c]
\end{array}
$$

$$
\begin{array}{rll}
e = \texttt{let}\ x = e_1\ \texttt{in}\ e_2 & R,\varphi & \rightarrow \ \text{Let } \varphi_1, \varphi_2 \text{ be the effect sets of } e_1, e_2 \\
 & & \ \forall (c \in R(\varphi_1))\ S^{\text{in}}_{e,R}[c] = S^{\text{in}}_{e_1,R}[c] \ \wedge \ S^{\text{out}}_{e_1,R}[c] = S^{1}_{e}R[c] \\
 & & \ \forall (c \in R(\varphi) - R(\varphi_1) \cup R|_{\epsilon}(\varphi))\ S^{\text{in}}_{e,R}[c] = S^{1}_{e,R}[c] \\
 & & \ \forall (c \in R(\varphi_2))\ S^{1}_{e,R}[c] = S^{\text{in}}_{e_2,R}[c] \ \wedge \ S^{\text{out}}_{e_2,R}[c] = S^{\text{out}}_{e,R} \\
 & & \ \forall (c \in R(\varphi) - R(\varphi_2) \cup R|_{\epsilon}(\varphi))\ S^{1}_{e,R}[c] = S^{\text{out}}_{e,R}[c]
\end{array}
$$

$$
\begin{array}{rll}
e = \texttt{letrec}\ f[\vec{\rho}, \epsilon.\vec{\varphi}](x)@\rho = e_1\ \texttt{in}\ e_2 & R,\varphi & \rightarrow \ \text{Let } \varphi_2 \text{ be the effect set of } e_2 \\
 & & \ S^{\text{in}}_{e,R}[R(\rho)] = \mathbf{A} \\
 & & \ \forall (c \in R(\varphi_2))\ S^{\text{in}}_{e,R}[c] = S^{\text{in}}_{e_2,R}[c] \\
 & & \ \forall (c \in R(\varphi_2))\ S^{\text{out}}_{e_2,R}[c] = S^{\text{out}}_{e,R}[c] \\
 & & \ \forall (c \in R(\varphi) - R(\varphi_2) \cup R|_{\epsilon}(\varphi))\ S^{\text{in}}_{e,R}[c] = S^{\text{out}}_{e,R}[c]
\end{array}
$$

$$
\begin{array}{rll}
e = \texttt{letregion}\ \rho\ \texttt{in}\ e_1 & R,\varphi & \rightarrow \ \text{Let } \varphi_2 \text{ be the effect set of } e_2 \\
 & & \ \text{Let } d = \text{minimum color} \notin \textit{Range}|_{\rho}(R) \\
 & & \ \text{Let } R_1 = R[\rho \leftarrow d] \\
 & & \ \forall c \in R(\varphi).S^{\text{in}}_{e,R}[c] = S^{\text{in}}_{e_1,R_1}[c] \\
 & & \ \forall c \in R(\varphi).S^{\text{out}}_{e_1,R_1}[c] = S^{\text{out}}_{e,R}[c] \\
 & & \ c_e \Longrightarrow S^{\text{in}}_{e_1,R_1}[d] = \mathbf{U} \ \wedge \ S^{\text{out}}_{e_1,R_1}[d] = \mathbf{D} \\
 & & \ \neg c_e \Longrightarrow S^{\text{in}}_{e_1,R_1}[d] = \mathbf{A} \ \wedge \ S^{\text{out}}_{e_1,R_1}[d] = \mathbf{A}
\end{array}
$$

$$
\begin{array}{rll}
e = \texttt{alloc\_before}\ \rho\ c_e\ e_1 & R,\varphi & \rightarrow \ \forall (c \in R(\varphi) \text{ s.t. } c \neq R(\rho))\ S^{\text{in}}_{e,R} = S^{1}_{e,R} \\
 & & \ \langle S^{\text{in}}_{e,R}[R(\rho)], c_e, S^{1}_{e,R}[R(\rho)] \rangle_a \\
 & & \ \forall (c \in R(\varphi_1))\ S^{1}_{e,R} = S^{\text{in}}_{e_1,R} \wedge \ S^{\text{out}}_{e_1,R} = S^{\text{out}}_{e,R} \\
 & & \ \forall (c \in R(\varphi) - R(\varphi_1) \cup R|_{\epsilon}(\varphi))\ S^{1}_{e,R} = S^{\text{out}}_{e,R} \\
 & & \ R(\rho) \in R|_{\epsilon}(\varphi) \Longrightarrow c_e = \textit{false}
\end{array}
$$

$$
\begin{array}{rll}
e = \texttt{free\_after}\ \rho\ c_e\ e_1 & R,\varphi & \rightarrow \ \forall (c \in R(\varphi_1))\ S^{\text{in}}_{e,R} = S^{\text{in}}_{e_1,R} \wedge \ S^{\text{out}}_{e_1,R} = S^{1}_{e,R} \\
 & & \ \forall (c \in R(\varphi) - R(\varphi_1) \cup R|_{\epsilon}(\varphi))\ S^{\text{in}}_{e,R} = S^{1}_{e,R} \\
 & & \ \forall (c \in R(\varphi) \text{ s.t. } c \neq R(\rho))\ S^{1}_{e,R} = S^{\text{out}}_{e,R} \\
 & & \ \langle S^{1}_{e,R}[R(\rho)], c_e, S^{\text{out}}_{e,R}[R(\rho)] \rangle_d \\
 & & \ R(\rho) \in R|_{\epsilon}(\varphi) \Longrightarrow c_e = \textit{false}
\end{array}
$$

Figure 7: Constraint generation rules.

ρ_6	$t_{6,1}$	$t_{6,2} = A$	$t_{6,3}$	$t_{6,4}$				
ρ_5	$t_{5,1}$	$t_{5,2}$	$t_{5,3}$	$t_{5,4} = A$	$t_{5,5}$	$t_{5,6} = A$	$t_{5,7}$	$t_{5,8}$
ρ_4	$t_{4,1}$	$t_{4,2}$	$t_{4,3} = A$	$t_{4,4}$	$t_{4,5}$	$t_{4,6}$	$t_{4,7} = A$	$t_{4,8}$
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.

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 $t_{i,j}$.

Table 1 contains explicit allocation constraints on states where regions are accessed. We must have $t_{5,5} = A$ because it lies on an execution path between two states where the region bound to ρ_5 is allocated. The same holds for $t_{4,4\text{–}6}$. 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 $\langle t_1, c_p, t_2 \rangle_a$ 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 $\langle t_1, c_p, t_2 \rangle_a$ where:

$$t_1 \text{ is } \textit{unconstrained} \wedge t_2 = A$$

By the definition of an allocation triple, choosing $c_p = \textit{true}$ forces $t_1 = U$. 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 $t_{5,3} = U$, and propagate U backwards through $t_{5,2\text{–}1}$ to the `letregion` for ρ_5. Similarly, we choose $c_{6,1} = \textit{true}$ and $c_{4,2} = \textit{true}$.

For efficiency, our solver is optimistic and assumes that setting *any* choice variable to *true* results in a constraint system with at least one solution. This assumption appears to be valid for all realistic programs, and allows us to avoid what appears to be a combinatoric problem.

Our constraint solver consists of four components. The first component performs graph calculations to determine that some choice variables must be *false* in any consistent solution, and sets them to *false*. The second component assigns values to all variables for which the values can be computed entirely from local context. After all such variables have been found, the third component of the algorithm chooses an unconstrained choice variable c_p as above. In our experience, this process eventually leads to a consistent solution of the constraints for almost all programs. The only exceptions we have found are carefully constructed counterexamples designed to force our solver into an inconsistent state (e.g. a region constrained to be both allocated and deallocated). In such cases, the fourth component identifies the region which became inconsistent, and assign it the trivial solution (i.e. choosing the earliest allocation inside the `letregion`, and the latest deallocation). In this way, the solver is guaranteed to find a solution; in the worst case, it is identical to the original Tofte/Talpin target program.

5.4.1 Reformulation as graph problem

To facilitate an efficient solution, it is helpful to reformulate the constraint system as a graph problem. Several non-local graph properties can be immediately computed. These properties correspond to sets of choice variables which cannot be true in any consistent solution, so they are assigned a value of *false*.

The graph is constructed as follows: the nodes in the graph are state variables t_i, and the directed edges in the graph are allocation and deallocation triples $\langle t_1, c_k, t_2 \rangle$, where the endpoints of the edge are t_1 and t_2. Each choice variable thus corresponds to a set of edges. The graph is described formally as follows:

$$\begin{array}{rcl} \mathrm{G} & = & \mathrm{StateVar} \times \mathrm{E} \\ \mathrm{E} & = & \mathrm{StateVar} \times \mathrm{StateVar} \\ \mathrm{TripMap} & = & \mathrm{E} \xrightarrow{\mathrm{fin}} \mathrm{Triple} \\ \mathrm{Triple} & = & \mathrm{StateVar} \times \mathrm{ChoiceVar} \times \mathrm{StateVar} \times \{a, d\} \end{array}$$

A map $M \in \mathrm{TripMap}$ always satisfies the property that $M(t_1, t_2) = \langle t_1, c_k, t_2 \rangle_a$ or $M(t_1, t_2) = \langle t_1, c_k, t_2 \rangle_d$ for some k. There is at most one choice variable for any pair of state variables, which ensures that M is single-valued.

The graph does not encode equality constraints between state variables. In the implementation, equality constraints force unification of the state variables before the constraints are translated into graph form.

We define the notion of a *consistent labelling* of the graph and show a mapping between consistent labellings and solutions of the original constraints.

$$\begin{aligned}
\text{ELabelling} &= \text{E} \xrightarrow{\text{fin}} \text{Bool} \\
\text{NLabelling} &= \text{StateVar} \xrightarrow{\text{fin}} \{U, A, D\} \\
\text{Labelling} &= \text{ELabelling} \times \text{NLabelling}
\end{aligned}$$

We introduce a shorthand: any predicate containing the notation $\langle t_1, c_k, t_2\rangle_x$ must hold for both $\langle t_1, c_k, t_2\rangle_a$ and $\langle t_1, c_k, t_2\rangle_d$.

Definition 5.5 A labelling (EL, NL) of graph (N, E) with triple map M is *consistent* iff:

for all e s.t. $M(e) = \langle t_1, c_k, t_2\rangle_x \wedge EL(e) = \textit{false}, NL(t_1) = NL(t_2)$
for all e s.t. $M(e) = \langle t_1, c_k, t_2\rangle_a \wedge EL(e) = \textit{true}, NL(t_1) = U \wedge NL(t_2) = A$
for all e s.t. $M(e) = \langle t_1, c_k, t_2\rangle_d \wedge EL(e) = \textit{true}, NL(t_1) = A \wedge NL(t_2) = D$
for all e_1, e_2 s.t. $M(e_1) = \langle t_1, c_k, t_2\rangle_x \wedge M(e_2) = \langle t_3, c_k, t_4\rangle_x, EL(e_1) = EL(e_2)$

The last of these predicates states that all edges sharing a choice variable must be labelled the same. It implies that labelling an edge and labelling a choice point are equivalent notions; we will use these terms interchangeably in the sequel. The remainder of this section discusses each of the four components of the constraint solver in turn.

5.4.2 Graph properties

It is now possible to state the graph properties mentioned above. First, along a cycle, all edges must be labelled false. Second, if any two edges along a path share a choice variable, then those edges must be labelled false. These properties encode non-local information about the graph. Without this information, a solver based on strictly local information is likely to get stuck, and possibly arrive at an inconsistent labelling.

To prove these properties, we order the labellings for each state variable as follows: $U < A < D$. From the definition of consistent labelling, for any graph edge (t_1, t_2) we have $NL(t_1) \leq NL(t_2)$.

Lemma 5.6 For any consistent labelling (EL, NL) of a graph, for all edges (t_1, t_2) in a cycle in the graph $EL(t_1, t_2) = \textit{false}$.

Proof: The cycle induces a cycle of $\leq$ relationships among the labellings state variables at the nodes. Therefore, these labellings are equal. From the definition of consistent labelling, equality of endpoints of an edge implies a labelling of *false* for the corresponding choice variable. □

Lemma 5.7 For any consistent labelling (EL, NL) of a graph, for any path that includes two edges sharing a choice variable, the label for the edges is *false*.

Proof: By contradiction. Let (t_1, t_2) be the first edge, and (t_3, t_4) be the second (i.e. there is a path from t_2 to t_3. Without loss of generality, let both edges map to allocation triples. Assume that the $EL(t_1, t_2) = EL(t_3, t_4) = \textit{true}$. Then, $NL(t_2) = A$ and $NL(t_3) = U$, so $NL(t_2) > NL(t_3)$. However, there is a path from t_2 to t_3, implying (due to the transitivity of $\leq$) that $t_2 < t_3$, a contradiction. □

Our implementation detects cycles with a standard algorithm for finding strongly connected components. Cycles are removed, leaving an acyclic digraph for the remainder of the algorithm. Our implementation performs a depth first search for each choice variable to find edges on a path sharing a choice variable.

5.4.3 Local transformations

The next phase performs local transformations of the graph. This phase maintains a partial labelling of the graph. Each local transformation refines the partial labelling to contain more information. This process repeats until no more such local transformations are possible, at which point one of the remaining unconstrained choice points is labelled *true*. The choice phase alternates with the local transformation phase until either the labelling is completed or becomes inconsistent.

The partial labelling assigns three possible values to each choice variable and six possible values to each state variable. Choice variables can be *true, false,* or *unknown.* State variables can be one of $\{U, A, D, \not{D}, \not{U}, \theta\}$. Of these, $\not{D}$ states that U and A are still considered possible alternatives, $\not{U}$ states that A and D are possible, and θ means that the final labelling is as yet

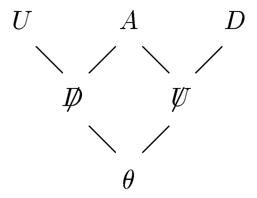


Figure 8: Strict partial order on the restrictedness of state variables

unconstrained. These six values form a strict partial order, as shown in Figure 8. We write $\theta < \not{U}$ to indicate that θ contains less information about the final labelling than does $\not{U}$. The least upper bound $x \sqcup y$ is defined as the least value which is greater than both x and y.

The partial labelling is initialized as follows. All edges known from the graph properties to be *false* are labelled as such. All other edges are labelled *unknown.* Any state variable constrained by a $t_j = A$ constraint is labelled A, otherwise θ.

The local transformations are specified by the following table. For each triple of the form $\langle t_1, c_k, t_2 \rangle_a$, where c_k is *unknown,* the table specifies a refinement to the labelling for the state variables t_1 and t_2 and the choice variable c_k. The row is $NL(t_1)$ and the column is $NL(t_2)$. A blank or numerical table entry indicates that the labelling cannot be made more precise just from local information. A "—" entry indicates that the partial labelling is inconsistent. Otherwise, the partial labelling is updated so that NL maps t_1 to the first symbol and t_2 to the third, and EL maps all instances of c_k to the third symbol (where t represents true, f represents false, and ? represents that the choice variable will remain unknown).

		$NL(t_2)$					
		U	A	D	$\not{D}$	$\not{U}$	θ
$NL(t_1)$	U	UfU	UtA	—	2	UtA	$U?\not{D}$
	A	—	AfA	—	AfA	AfA	AfA
	D	—	—	DfD	—	DfD	DfD
	$\not{D}$	UfU	1	—		$\not{D}?A$	$\not{D}?\not{D}$
	$\not{U}$	—	AfA	DfD	AfA	$\not{U}f\not{U}$	$\not{U}f\not{U}$
	θ	UfU	$\not{D}?A$	DfD	$\not{D}?\not{D}$		

Allocation triple: $\langle t_1, c_k, t_2 \rangle_a$

For a triple of the form $\langle t_1, c_k, t_2 \rangle_a$, where c_k is *true,* if $NL(t_1) \leq U$ and $NL(t_2) \leq A$, then the labelling is updated to so that NL maps t_1 to U and t_2 to A. Otherwise, the partial labelling is inconsistent. Finally, for a triple of the form $\langle t_1, c_k, t_2 \rangle_a$, where c_k is *false,* the labelling is updated so that NL maps both t_1 and t_2 to their least upper bound if comparable, otherwise the partial labelling is inconsistent.

The rules for deallocation triples are analogous. For each deallocation triple in which the choice variable is *unknown,* the following table specifies the rewrite, as above.

		$NL(t_2)$					
		U	A	D	$\not{D}$	$\not{U}$	θ
$NL(t_1)$	U	UfU	—	—	UfU	—	UfU
	A	—	AfA	AtD	AfA	1	$A?\not{U}$
	D	—	—	DfD	—	DfD	DfD
	$\not{D}$	UfU	AfA	AtD	$\not{D}f\not{D}$	$A?\not{U}$	
	$\not{U}$	—	AfA	2	AfA		$\not{U}?\not{U}$
	θ	UfU	AfA	$\not{U}?D$	$\not{U}?\not{U}$	$\not{U}f\not{U}$	

Deallocation triple: $\langle t_1, c_k, t_2 \rangle_d$

Deallocation triples in which the choice variable is *false* are handled as allocation triples. Deallocation triples in which the choice variable is *true* are handled similarly to allocation triples, except that the two state variables are assigned A and D rather than U and A, respectively.

5.4.4 Resolving choice points

Ultimately, the process of local transformation either reaches a consistent solution, an inconsistency, or is simply unable to refine the partial labelling based on local information. In the latter case, there are unresolved choice variables. Our algorithm chooses one and sets it to true, and then propagates local transformations again.

There remains the question of which choice variable to select. In principle, any *unknown* choice variable can be chosen. In practice, we want the choice to lead to a good annotation of the program, i.e. placing allocations as late as possible and deallocations as early as possible. Thus, the choice points are prioritized and one with the highest priority is chosen. The priority numbers are listed in the above tables, with 1 as the highest priority.

Priority 1 entries represent choice points "on the boundary", i.e. those that connect a state variable constrained to A and another one unconstrained. Labelling such a choice variable *true* corresponds to the latest possible allocation and the earliest possible deallocation. Priority 2 entries represent choice points in connecting two state variables neither of which is constrained to A. These choice points generally remain only when there is no A constraint on a path (e.g., in one branch of an `if` statement). In these cases, selecting such a choice point will result in a completion in which the allocation of the region is immediately followed by the corresponding deallocation.

5.4.5 Inconsistent labellings

It would be very nice if the system as presented above would always find a consistent labelling. However, we have been able to construct contrived examples which lead the solver to an inconsistent partial labelling. To cope with this case, we simply fall back to a more conservative solution for all choice variables associated with the region responsible for the inconsistency.

When an inconsistency is found, all choice and state variables that are affected by the inconsistency are assigned the most conservative labelling. The set of affected variables is the transitive closure of all graph edges, reversed graph edges, and shared choice variables. The most conservative labelling is defined as follows: set all affected choice variables to false, and all affected state variables to A. Then, the constraint solver continues.

The termination argument for the overall algorithm is quite simple: each step decreases the number of partially labelled variables. In the worst case, when an inconsistency is found for every region, eventually all variables would be assigned the most conservative labelling.

6 Soundness

This section states and proves a soundness theorem for our system. The soundness theorem expresses that when a program completion corresponds to a solution of the associated constraint system, the evaluation of the annotated program does not dereference dangling pointers. In particular, all accesses to regions are to allocated regions. The theorem is formulated as follows. Assume that $s, r, n \vdash e : \varphi \to a, s'$, and assume that $[\![e]\!]\ R = V$ is the result of the region-based closure analysis for e, where R abstracts the region environment r. Assume further that the regions in e's effect φ mapped by r in store s are initially in the states given by the solution of the state variables $S^{\text{in}}_{e,R}$. The theorem shows that the evaluation of e leaves these regions in the states specified by the the solution for $S^{\text{out}}_{e,R}$. To prove this theorem we introduce extra notation and definitions, an operational semantics for the completion language that makes effect instantiation explicit, and definitions of relations between abstract and concrete entities. We then present a few lemmas and the detailed proof.

6.1 Notation and Definitions

We generally use capital letters for abstract entities and lowercase letters for concrete entities, z_c denotes a concrete region with color c, s is a store, and $\mathcal{S}$: StateVar $\to \{\mathbf{U}, \mathbf{A}, \mathbf{D}\}$ is a solution of the constraints. The function *color* maps a concrete region z_c to its associated color c, and the function *colors* maps sets of regions to sets of colors by taking the union of the pointwise application of *color*. The function *state* maps the state of a region z_c in a store s to $\{\mathbf{U}, \mathbf{A}, \mathbf{D}\}$:

$$\mathit{state}(s, z_c) = \begin{cases} \mathbf{U} & \text{if } s(z_c) = \mathit{unallocated} \\ \mathbf{A} & \text{if } s(z_c) = \lambda o \ldots \\ \mathbf{D} & \text{if } s(z_c) = \mathit{deallocated} \end{cases}$$

As for abstract region environments, we overload notation for applying concrete region environments to region variables $r(\rho)$, effect variables $r(\epsilon)$, sets of both $r(\varphi)$, and arrow-effects $r(\epsilon.\varphi)$. The exact form should always be clear from context and the operand kind (definitions for abstract region environments were given in Section 5.2):

$$\begin{array}{lcl} r(\varphi) & = & \{z_c \mid \exists \rho \in \varphi \text{ s.t. } r(\rho) = z_c\} \cup \bigcup_{\epsilon \in \varphi} r(\epsilon) \\ r(\epsilon.\varphi) & = & r(\varphi \cup \{\epsilon\}) \end{array}$$

Special domain and range functions separate region and effect variables. $Dom|_\rho(r)$ is the set of region variables in the domain of the region environment r, and $Dom|_\epsilon(r)$ is the set of effect variables in the domain of r. The range functions are defined analogously.

$$\begin{array}{lcl} Dom|_\rho(r) & = & \{\rho \mid \rho \in Dom(r) \wedge \rho \in \text{RegVar}\} \\ Dom|_\epsilon(r) & = & \{\epsilon \mid \epsilon \in Dom(r) \wedge \epsilon \in \text{EffVar}\} \\ Range|_\rho(r) & = & \{r(\rho) \mid \rho \in Dom|_\rho(r)\} \\ Range|_\epsilon(r) & = & \bigcup\{r(\epsilon) \mid \epsilon \in Dom|_\epsilon(r)\} \end{array}$$

We also make use of restricted mappings $r|_\rho(\varphi)$ and $r|_\epsilon(\varphi)$ to map only region or effect variables in φ:

$$\begin{array}{lcl} r|_\rho(\varphi) & = & \{r(\rho) \mid \rho \in (\varphi \cap Dom|_\rho(r))\} \\ r|_\epsilon(\varphi) & = & \bigcup\{r(\epsilon) \mid \epsilon \in (\varphi \cap Dom|_\epsilon(r))\} \end{array}$$

6.2 Refined Operational Semantics

The soundness proof uses the following property of the Tofte and Talpin typing: for every expression e, all regions accessed during evaluation of e are present in the current region environment. In other words, the evaluation of e will not access a random region that can't be determined from the direct context of e. This property is intuitive for expressions without applications, since regions are accessed only through region variables and therefore appear in the region environment. The property is less intuitive when e contains applications, since a called function may access regions that e doesn't directly use. However, the latent effect set of the called function specifies region variables through which regions are accessed during the call, and this effect set is part of e's effect. Because latent effects can contain effect variables, not all regions accessed during evaluation of e are "region variable"-bound. Some regions are "effect-variable"-bound in the environment of e.

In Section 5.1 we used the above-mentioned property implicitly to motivate the handling of quantified effect variables. In the proof we use a corollary of the property, which expresses that at every application, the caller and the callee agree on the set of regions that are potentially accessed during the call. In the presence of quantified region and effect variables, this property is not syntactic, but it can be expressed by the following general lemma.

Lemma 6.1 Given the evaluation of an expression e in region environment r

$$s, r, n \vdash e : (\mu_1 \xrightarrow{\varphi} \mu_2, \rho) \to a, s'$$

with the resulting closure value

$$s'(a) = \langle \lambda x.e' : (\mu_1' \xrightarrow{\varphi'} \mu_2', \rho'), r', n' \rangle$$

then the set of regions accessed by calling this closure is known to the calling context:

$$r'(\varphi') = r(\varphi)$$

For region polymorphic closures, the relation is $r'(\varphi') \subseteq r(\varphi)$. Region and effect instantiation reestablishes the equality. Thus the equality always holds for ordinary closures.

Proof: By induction on the evaluation of the closed expression containing e (requires explicit type variable instantiation). □

As discussed above, Lemma 6.1 depends on effect variable instantiation, which is not needed in the standard operational semantics for the completion language given in Figure 4. The extended operational semantics given in Figure 9 makes types and effects explicit: in particular effect variable quantification and instantiation is explicit. Concrete region environments are extended with mappings from effect variables to sets of regions. For simplicity, we assume that effect variables are only quantified at `letrec` constructs, but not at `let`. Extending the proof to `let` effect variable quantification for non-expansive let-expressions ([Tof90]) does not pose new technical difficulties, but the operational semantics must be modified in unintuitive ways. Note that the refinements presented here do not affect the way programs are executed. An implementation can be based solely on the original operational semantics. The rules have the general form:

$$s, n, r \vdash e : \mu, \varphi \to a, s'$$

where s is the store prior to the evaluation of the expression e, n is the value environment, r the concrete region environment containing region and effect variables, μ and φ are the static Tofte and Talpin type and effect for e, a is the address of the result of the evaluation, and s' is the store after evaluating e. The domains used by the operational semantics are given below. Note that region environments always map region variables to a single region, and effect variables to sets of regions.

$$
\begin{array}{rcl}
\text{Region} & = & \text{Int} \times \text{Color} \\
\text{Lam} & = & \text{Var} \times \text{e} \times \mu \\
\text{RegionState} & = & \text{unallocated} + \text{deallocated} + \\
& & (\text{Offset} \xrightarrow{\text{fin}} \text{Clos} + \text{RegClos}) \\
\text{Store} & = & \text{Region} \xrightarrow{\text{fin}} \text{RegionState} \\
\text{Clos} & = & \text{Lam} \times \text{Env} \times \text{RegEnv} \\
\text{RegClos} & = & \text{Lam} \times \text{RegionVar}^* \times \text{EffectVar}^* \times \text{Env} \times \text{RegEnv} \\
\text{Env} & = & \text{Var} \xrightarrow{\text{fin}} \text{Region} \times \text{Offset} \\
\text{RegEnv} & = & \text{RegionVar} + \text{EffectVar} \xrightarrow{\text{fin}} \text{Region} + \mathcal{P}(\text{Region})
\end{array}
$$

The following paragraphs describe and motivate the refinements to the rules of the operational semantics in Figure 9. Effect variable instantiation is made explicit in the [REGAPP] rule. The instantiation vector $\vec{\varphi}$ is found analogously to the region variable instantiations by type and effect inference. Region polymorphic closures contain an extra vector of quantified effect variables. This information is used in the [REGAPP] rule to extend the region environment of the instantiated closure. The effect instantiation is computed as described in Section 5.1.3 and equation (1). The effect sets φ'_i are obtained from the arrow-effects $(\epsilon'_i.\varphi'_i)$ of the quantified effect variable ϵ'_i.

Closures include the type of the associated lambda abstraction, as is apparent from the [LETREC], [REGAPP], [ABS], and [APP] rules. This information is used in the proof of the [APP] rule to reason about the syntactic type of the callee.

Concrete regions are subscripted with colors. Colors are chosen in the same manner as in the region-based closure analysis, creating the necessary correspondence between concrete and abstract regions (see Definition 6.2).

The proof requires two distinct `letregion` constructs as shown in rules [LETREGION] and [LETREGION_TT]. The first construct (`letregion`) behaves exactly as described in Section 2, where the new region is initially unallocated and is deallocated at the end. The second construct `letregion_tt` has the same semantics as in [TT94], i.e. a region is initially allocated but empty, and still allocated at the end. There are no free/alloc constructs for such regions. The reason for the second construct is the finite nature of our color abstraction. Recall that in the region-based closure analysis, the color abstracting the new region in a `letregion` construct is chosen to be different from all other "region variable"-bound colors (to correctly capture the aliasing between region variables). However, we cannot choose the color to be distinct from colors bound by effect variables, because this would potentially require an infinite number of distinct colors. The color for the new region may thus already be bound to an effect variable. Our constraints track a single region state per color and region environment at each program point. We therefore cannot require that the state of the freshly chosen region be unallocated, because it must be the same as the states of equally colored regions mapped by effect variables. Note that we have chosen unification of states to happen directly at `letregion` constructs. This choice is arbitrary and it is possible to delay the unification of the region states (e.g. until applications). Delaying state unification may lead to slightly better annotations, but the current approach is preferred for purposes of readability and simplicity in the proof. The solution to the constraints for a particular Tofte/Talpin target program determines the kind of each `letregion` construct.

Our proof does not attempt to make the *reuse* of regions formally safe. The proof requires `letregion` constructs to use fresh regions. This is arguably a weakness with respect to the Tofte/Talpin system, where the reuse of regions is provably safe [TT93]. However, we prove that once a region is deallocated, no more reads or writes to the memory held by that region are made.

6.3 Relations between abstract and concrete entities

The following paragraphs define relations between abstract and concrete entities that are used throughout the proof.

Definition 6.2 A concrete region environment r and an abstract region environment R match (written $r \;\mathsf{sat}\; R$), if they have the same domain and aliasing structure, and the color annotation of concrete regions corresponds to the colors in the abstract region environment.

$$
\begin{array}{l}
r \;\mathsf{sat}\; R \stackrel{\text{def}}{\equiv} \\
\quad \mathit{Dom}(R) = \mathit{Dom}(r) \wedge \\
\quad R(\rho) = R(\rho') \iff r(\rho) = r(\rho') \wedge \\
\quad R(\rho) = \mathit{color}(r(\rho)) \wedge \\
\quad R(\epsilon) = \mathit{colors}(r(\epsilon))
\end{array}
$$

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

$$\frac{\begin{array}{l} n(f) = a \\ s(a) = \langle \lambda x.e : \mu', \vec{\rho}', \epsilon'.\vec{\varphi}', n_0, r_0 \rangle \quad \text{where } \mu' = (\tau', \rho') \\ n' = n_0[f \leftarrow a] \\ a_1 = (r(\rho), o) \qquad o \notin \mathit{Dom}(s(r(\rho))) \\ r' = r_0[\vec{\rho}' \leftarrow r(\vec{\rho})] \\ r'' = r'[\epsilon'_i \leftarrow (r|_\rho(\varphi_i) - r'|_\rho(\varphi'_i)) \cup r|_\epsilon(\varphi_i)] \\ s_1 = s[a_1 \leftarrow \langle \lambda x.e : \mu', n', r'' \rangle] \end{array}}{s, n, r \vdash f[\vec{\rho}, \vec{\varphi}] \,@\, \rho : \mu, \{\rho, \rho'\} \rightarrow a_1,\ s_1} \qquad [\mathit{REGAPP}]$$

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

$$\frac{\begin{array}{l} s, n, r \vdash e_1 : \mu_1, \varphi_1 \rightarrow a_1, s_1 \quad \text{where } \mu_1 = (\mu_2 \xrightarrow{\varphi_a} \mu, \rho) \\ s_1, n, r \vdash e_2 : \mu_2, \varphi_2 \rightarrow a_2, s_2 \\ s_2(a_1) = \langle \lambda x.e : \mu'_1, n_0, r_0 \rangle \quad \text{where } \mu'_1 = (\mu'_2 \xrightarrow{\varphi_l} \mu', \rho') \\ s_2, n_0[x \leftarrow a_2], r_0 \vdash e : \mu', \varphi_l \rightarrow a_3, s_3 \end{array}}{s, n, r \vdash e_1\ e_2 : \mu, \varphi_1 \cup \varphi_2 \cup \varphi_a \cup \{\rho\} \rightarrow a_3, s_3} \qquad [\mathit{APP}]$$

$$\frac{\begin{array}{l} a_1 = (r(\rho), o) \qquad o \notin \mathit{Dom}(s(r(\rho))) \\ n' = n[f \leftarrow a_1] \\ s_1 = s[a_1 \leftarrow \langle \lambda x.e_1 : \mu_1, \vec{\rho}, \epsilon.\vec{\varphi}, n, r \rangle] \quad \text{where } \mu_1 \text{ is the type of } \lambda x.e_1 \\ s_1, n', r \vdash e_2 : \mu_2, \varphi_2 \rightarrow a, s_2 \end{array}}{s, n, r \vdash \texttt{letrec}\ f[\vec{\rho}, \epsilon.\vec{\varphi}](x) \,@\, \rho = e_1\ \texttt{in}\ e_2 : \mu_2, \varphi_2 \cup \{\rho\} \rightarrow a, s'} \qquad [\mathit{LETREC}]$$

$$\frac{\begin{array}{l} z_d \text{ fresh} \qquad d = \text{minimum color not in } \mathit{color}(\mathit{Range}|_\rho(r)) \\ s_0 = s[z_d \leftarrow \mathit{unallocated}] \\ s_0, n, r[\rho \leftarrow z_d] \vdash e : \mu, \varphi \cup \{\rho\} \rightarrow a_1, s_1 \\ s_1(z_d) = \mathit{deallocated} \end{array}}{s, n, r \vdash \texttt{letregion}\ \rho\ \texttt{in}\ e : \mu, \varphi \rightarrow a_1, s_1|_{\mathit{Dom}(s)}} \qquad [\mathit{LETREGION}]$$

$$\frac{\begin{array}{l} z_d \text{ fresh} \qquad d = \text{minimum color not in } \mathit{color}(\mathit{Range}|_\rho(r)) \\ s_0 = s[z_d \leftarrow \{\}] \\ s_0, n, r[\rho \leftarrow z_d] \vdash e : \mu, \varphi \cup \{\rho\} \rightarrow a_1, s_1 \\ s_1(z_d) \text{ is allocated} \end{array}}{s, n, r \vdash \texttt{letregion\_tt}\ \rho\ \texttt{in}\ e : \mu, \varphi \rightarrow a_1, s_1|_{\mathit{Dom}(s)}} \qquad [\mathit{LETREGION\_TT}]$$

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

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

Figure 9: Refined Operational Semantics.

Definition 6.3 A store s and address a match a set of abstract values V (written $s, a \text{ sat } V$), if V contains an abstraction of the concrete value stored at address a in s, and the environment of the concrete closure matches the region-based closure analysis $[\![\cdot]\!]$ (defined below).

$$
\begin{array}{l}
s, a \text{ sat } V \stackrel{\text{def}}{\equiv} \\
\quad a = (o, z_c) \\
\quad z_c \in \mathit{Dom}(s) \ \wedge \ \mathit{state}(s, z_c) = \mathbf{A} \Longrightarrow \\
\quad\quad s(a) = \langle \lambda x.e, r', n' \rangle \ \wedge \\
\quad\quad \exists \langle \lambda x.e, R' \rangle \in V \text{ s.t.} \\
\quad\quad\quad s, r', n' \text{ sat } [\![\cdot]\!] \ R'
\end{array}
$$

Definition 6.4 A store s, concrete region environment r, and concrete value environment n match an abstract region environment R and the region-based closure analysis (written $s, r, n \text{ sat } [\![\cdot]\!] \ R$), if the abstract and concrete region environments match, and if for every variable x in the concrete environment, $[\![x]\!] \ R'$ contains an abstraction of the concrete value.

$$
\begin{array}{l}
s, r, n \text{ sat } [\![\cdot]\!] \ R \stackrel{\text{def}}{\equiv} \\
\quad r \text{ sat } R \ \wedge \\
\quad \forall (x \in \mathit{Dom}(n)) \ \exists R' \text{ s.t.} \\
\quad\quad (R|_{\mathit{Dom}(R')} = R' \ \wedge \ s, n(x) \text{ sat } [\![x]\!] \ R')
\end{array}
$$

Definition 6.5 A store s and a concrete region z_c with color c match a state variable $S_{e,R}[c]$ (written $s, z_c \text{ sat } S_{e,R}[c]$), if the state of the region z_c in the store s corresponds to the solution for $S_{e,R}[c]$.

$$
\begin{array}{l}
s, z_c \text{ sat } S_{e,R}[c] \stackrel{\text{def}}{\equiv} \\
\quad z_c \in \mathit{Dom}(s) \Longrightarrow \mathit{state}(s, z_c) = \mathcal{S}(S_{e,R}[c])
\end{array}
$$

Definition 6.6 Finally, a state s and concrete region environment r match an abstract region environment R, state vector $S_{e,R}$, and effect set φ (written $s, r \text{ sat } R, S_{e,R}, \varphi$), if r and R match, the states of all regions in φ match the solution of the constraints for $S_{e,R}$, and if the color of a region is bound to an effect variable in φ, then the region is in the allocated state.

$$
\begin{array}{l}
s, r \text{ sat } R, S_{e,R}, \varphi \stackrel{\text{def}}{\equiv} \\
\quad \varphi \subseteq \mathit{Dom}(R) \ \wedge \\
\quad r \text{ sat } R \ \wedge \\
\quad \forall (z_c \in r(\varphi)) \ \ s, z_c \text{ sat } S_{e,R}[\mathit{color}(z_c)] \\
\quad \forall (c \in R|_\epsilon(\varphi)) \ \ \mathcal{S}(S_{e,R}[c]) = \mathbf{A}
\end{array}
$$

6.4 Soundness Theorem

The proof of our soundness theorem is aided by several lemmas presented below. Lemma 6.7 expresses that a set of abstract values matching a concrete value can be augmented without changing the relation.

Lemma 6.7 $s, a \text{ sat } V \ \wedge \ V \subseteq V' \Longrightarrow s, a \text{ sat } V'$

Proof: Follows from the definition of $s, a \text{ sat } V$. □

A store s can be extended with regions and/or offsets without affecting the validity of the environment abstraction. This fact is expressed by Lemma 6.8.

Lemma 6.8 If $s, r, n \text{ sat } [\![\cdot]\!] \ R$ and s' is a region and/or offset extension of s, then $s', r, n \text{ sat } [\![\cdot]\!] \ R$.

Proof: Store extensions cannot recapture dangling references because new regions are always chosen to be fresh. □

A region may change from the unallocated state to the allocated state, or from the allocated to the deallocated state without affecting the validity of the environment abstraction. This fact is expressed by Lemma 6.9.

Lemma 6.9 If $\mathit{state}(s, z) = \mathbf{U}$ and $s, r, n \text{ sat } [\![\cdot]\!] \ R$ then $s[z \leftarrow \{\}], r, n \text{ sat } [\![\cdot]\!] \ R$. Similarly, if $\mathit{state}(s, z) = \mathbf{A}$ and $s, r, n \text{ sat } [\![\cdot]\!] \ R$ then $s[z \leftarrow \mathit{deallocated}], r, n \text{ sat } [\![\cdot]\!] \ R$.

Proof: Allocation of a region cannot recapture dangling references, because the region has never before been in the allocated state and regions are not reused. □

The next Lemma (6.10) is used in the proof to establish the relation between the store and the constraints after an inductive step on a subexpression.

Lemma 6.10 Given the following relations and constraints

(1)	$s, r \text{ sat } R, S^1_{e,R}, \varphi$	
(2)	$s', r \text{ sat } R, S^2_{e,R}, \varphi_1$	
(3)	$\varphi_1 \subseteq \varphi$	
(4)	$Dom(s) = Dom(s')$	
(5)	$\forall(z_c \in Dom(s))$ s.t. $z_c \notin r(\varphi_1)$ $state(s, z_c) = state(s', z_c)$	
(6)	$\forall(c \in (R(\varphi) - R(\varphi_1)) \cup R\|_\epsilon(\varphi))\ S^1_{e,R}[c] = S^2_{e,R}[c]$	

we conclude

$s', r \text{ sat } R, S^2_{e,R}, \varphi$

Proof:

(7)	$\varphi \subseteq Dom(R)$	by 1
(8)	$r \text{ sat } R$	by 1
(9)	$\forall(z_c \in r(\varphi))\ s, z_c \text{ sat } S^1_{e,R}[c]$	by 1
(10)	$\forall(z_c \in r(\varphi_1))\ s', z_c \text{ sat } S^2_{e,R}[c]$	by 2
(11)	$\forall(c \in R\|_\epsilon(\varphi))\ \mathcal{S}(S^1_{e,R}[c]) = \mathbf{A}$	by 1,def. 6.6
(12)	$\forall(c \in R\|_\epsilon(\varphi))\ \mathcal{S}(S^2_{e,R}[c]) = \mathbf{A}$	by 11,6

To show: $\forall(z_c \in r(\varphi))\ s', z_c \text{ sat } S^2_{e,R}[c]$.

(13)	$c \in (R(\varphi) - R(\varphi_1)) \Longrightarrow z_c \notin r(\varphi_1)$	by 3,8
(14)	$\forall(z_c \in r(\varphi) - r(\varphi_1))\ c \in (R(\varphi) - R(\varphi_1)) \Longrightarrow s', z_c \text{ sat } S^2_{e,R}[c]$	by 5,13,6,9
(15)	let $z_c \in r(\varphi) - r(\varphi_1)$ s.t. $c \notin (R(\varphi) - R(\varphi_1))$	
(16)	$c \in R(\varphi_1)$	by 15
(17)	$\exists z'_c \in r(\varphi_1)$ s.t. $z'_c \neq z_c$	by 16,15
(18)	$z_c \in r\|_\epsilon(\varphi) \ \vee\ z'_c \in r\|_\epsilon(\varphi)$	by 17,8,3
(19)	$c \in R\|_\epsilon(\varphi)$	by 18,8
(20)	$state(s, z'_c) = state(s', z'_c)$	by 10,1,6,19
(21)	$state(s, z'_c) = state(s, z_c)$	by 1
(22)	$state(s', z_c) = state(s', z'_c)$	by 20,21,15,5
(23)	$s', z_c \text{ sat } S^2_{e,R}[c]$	by 22,10
(24)	$\forall(z_c \in r(\varphi) - r(\varphi_1))\ c \notin (R(\varphi) - R(\varphi_1)) \Longrightarrow s', z_c \text{ sat } S^2_{e,R}[c]$	by 15,23

The result follows from 10,14,24, and 12. □

The soundness of our analysis is stated by Theorem 6.11, which says that if the evaluation of an expression e_k results in an address a and a new store s', then the value stored at a in s' is correctly modeled by the abstract closure analysis. Furthermore, the states of regions in store s' is as predicted by the solution of the constraints.

Theorem 6.11 Assume that $\mathcal{S}$ is the solution to the constraints for a closed expression e, of which e_k is a subexpression. Given the result of the region-based closure analysis $[\![\cdot]\!]$ and the following premises,

(1)	$s, r, n \vdash e_k : \mu, \varphi \to a, s'$	
(2)	$s, r, n \text{ sat } [\![\cdot]\!]\ R$	
(3)	$s, r \text{ sat } R, S^{\text{in}}_{e_k,R}, \varphi$	

we conclude

$s', a \text{ sat } [\![e_k]\!]\ R$

$s', r \text{ sat } R, S^{\text{out}}_{e_k,R}, \varphi$

Proof: The proof is by induction on the structure of e_k. Note that

(4)	$r \text{ sat } R$	by 2
(5)	$\varphi \subseteq Dom(R)$	by 3

There are 9 cases. We prove each one in turn.

Case 1 The operational semantics rule for [VAR] is

(6) $s, r, n \vdash x : \mu, \varphi \rightarrow n(x), s$
where $\varphi = \emptyset$

By the definition of the region-based closure analysis and the constraint generation, we have

(7) $[\![x]\!]\ R = \bigcup\{[\![x]\!]\ R' \mid R|_{Dom(R')} = R'\}$

Therefore

(8) $\exists R'$ s.t. $R|_{Dom(R')} = R' \ \wedge\ s, n(x)\ \mathsf{sat}\ [\![x]\!]\ R'$ by 2

(9) $[\![x]\!]\ R' \subseteq [\![x]\!]\ R$ by 7

(10) $s, n(x)\ \mathsf{sat}\ [\![x]\!]\ R$ by 8,9,Lemma 6.7

(11) $s, r\ \mathsf{sat}\ R, S^{\text{out}}_{e_k,R}, \emptyset$ by def 6.6

The result for [VAR] follows from 10 and 11. □

Case 2 The operational semantics rule for [ABS] is

(12) $s, r, n \vdash \lambda x.e@\rho : \mu, \varphi \rightarrow a, s'$
where $a = (r(\rho), o)$ and $\varphi = \{\rho\}$

(13) $s' = s[a \leftarrow \langle \lambda x.e : \mu, r, n\rangle]$

By the definition of the region-based closure analysis and the constraint generation, we have

(14) $[\![\lambda x.e : \mu]\!]\ R = \{\langle \lambda x.e : \mu, R\rangle\}$

(15) $S^{\text{in}}_{e_k,R}[R(\rho)] = \mathbf{A}$

(16) $S^{\text{in}}_{e_k,R}[R(\rho)] = S^{\text{out}}_{e_k,R}[R(\rho)]$

Therefore

(17) $\mathit{state}(s, r(\rho)) = \mathbf{A}$ by 3,12,15

(18) $s', r\ \mathsf{sat}\ R, S^{\text{out}}_{e_k,R}, \varphi$ by 3,13,12,16

(19) $s', a\ \mathsf{sat}\ [\![\lambda x.e : \mu]\!]\ R$ by 2,12,14

The result for [ABS] follows from 18 and 19.

Case 3 The operational semantics rule for [LETREGION] is

(20) $z_{d'}$ fresh (to avoid recapture of dangling references)

(21) $d' =$ minimum color $\notin \mathit{colors}(\mathit{Range}|_{\rho}(r))$

(22) $r_1 = r[\rho \leftarrow z_{d'}]$

(23) $s_0 = s[z_{d'} \leftarrow \mathit{unallocated}]$

(24) $s_0, r_1, n \vdash e_1 : \mu, \varphi \cup \{\rho\} \rightarrow a_1, s_1$

(25) $s, r, n \vdash \texttt{letregion}\ \rho\ \texttt{in}\ e_1 : \mu, \varphi \rightarrow a_1, s_1|_{Dom(s)}$

By the definition of the region-based closure analysis and the constraint generation, we have

(26) $d =$ minimum color $\notin \mathit{Range}|_{\rho}(R)$

(27) $R_1 = R[\rho \leftarrow d]$

(28) $[\![\texttt{letregion}\ \rho\ \texttt{in}\ e_1]\!]\ R = [\![e_1]\!]\ R_1$

(29) $\forall c \in R(\varphi).S^{\text{in}}_{e_k,R}[c] = S^{\text{in}}_{e_1,R_1}[c]$

(30) $\forall c \in R(\varphi).S^{\text{out}}_{e_1,R_1}[c] = S^{\text{out}}_{e_k,R}[c]$

(31) $c_k \Longrightarrow S^{\text{in}}_{e_1,R_1}[d] = \mathbf{U}\ \wedge\ S^{\text{out}}_{e_1,R_1}[d] = \mathbf{D}$

(32) $\neg c_k \Longrightarrow S^{\text{in}}_{e_1,R_1}[d] = \mathbf{A}\ \wedge\ S^{\text{out}}_{e_1,R_1}[d] = \mathbf{A}$

The choice variable c_k associated with the letregion construct expresses the kind of letregion chosen by the constraint solver: $c_k \Rightarrow$ `letregion`, and $\neg c_k \Rightarrow$ `letregion_tt`. The outline of the proof for [LETREGION] is as follows: Let $\varphi_1 = \varphi \cup \{\rho\}$. We first show that

$s_0, r_1, n\ \mathsf{sat}\ [\![\,\cdot\,]\!]\ R_1$

$s_0, r_1\ \mathsf{sat}\ R_1, S^{\text{in}}_{e_1,R_1}, \varphi_1$

By induction, using $s_0, r_1, n \vdash e_1 : \mu, \varphi_1 \rightarrow a_1, s_1$, we derive

$s_1, a_1\ \mathsf{sat}\ [\![e_1]\!]\ R_1$

$s_1, r_1\ \mathsf{sat}\ R_1, S^{\text{out}}_{e_1,R_1}, \varphi_1$

We then show that

$s_1|_{Dom(s)}, a_1 \ \mathsf{sat}\ [\![\texttt{letregion}\,\rho\,\texttt{in}\,e_1]\!]\ R$

$s_1|_{Dom(s)}, r \ \mathsf{sat}\ R, S^{\text{out}}_{e_k,R}, \varphi$

and the result follows immediately.

(33) $d = d'$ — by 4,21,26

(34) $r_1 \ \mathsf{sat}\ R_1$ — by 33,4,27,22

(35) $s_0, r, n \ \mathsf{sat}\ [\![\cdot]\!]\ R$ — by 2,23,Lemma 6.8

(36) $s_0, r_1, n \ \mathsf{sat}\ [\![\cdot]\!]\ R_1$ — by 35,34,27

(37) $\varphi \cup \{\rho\} \subseteq Dom(R[\rho \leftarrow d])$ — by 5

(38) $\mathcal{S}(S^{\text{in}}_{e_1,R_1}[d]) = \mathbf{U} \ \wedge\ \mathcal{S}(S^{\text{out}}_{e_1,R_1}[d]) = \mathbf{D}$ — by 31,letreg. choice

(39) $s_0, z_d \ \mathsf{sat}\ S^{\text{in}}_{e_1,R_1}[d]$ — by 33,23,38

Now we establish that there are no regions with color d in region environment R.

(40) Assume $\exists z'_c \in r|_\epsilon(\varphi)$ s.t. $c = d$

(41) $d \in R|_\epsilon(\varphi)$ — by 40,4

(42) $\mathcal{S}(S^{\text{in}}_{e_k,R}[d]) = \mathbf{A}$ — by 41,3,def. 6.6

(43) $\mathcal{S}(S^{\text{in}}_{e_1,R_1}[d]) = \mathbf{A}$ — by 42,29

(44) $\nexists z'_c \in r|_\epsilon(\varphi)$ s.t. $c = d$ — by 40,43,38,contrad.

(45) $s_0, r_1 \ \mathsf{sat}\ R_1, S^{\text{in}}_{e_1,R_1}, \varphi$ — by 3,23,44,21

(46) $s_0, r_1 \ \mathsf{sat}\ R_1, S^{\text{in}}_{e_1,R_1}, \varphi_1$ — by 34,37,45,39

(47) $s_1, a_1 \ \mathsf{sat}\ [\![e_1]\!]\ R_1$ — by 24,36,46,induct

(48) $s_1, r_1 \ \mathsf{sat}\ R_1, S^{\text{out}}_{e_1,R_1}, \varphi_1$ — by 24,36,46,induct

It remains to be shown that the abstract value of the letregion construct abstracts the result, and that the store is in the state specified by the constraints.

(49) $s_1, a_1 \ \mathsf{sat}\ [\![\texttt{letregion}\,\rho\,\texttt{in}\,e_1]\!]\ R$ — by 47,28

(50) $Dom(s_1) - Dom(s) = \{z_d\}$ — by 23,33

(51) $s_1|_{Dom(s)}, a_1 \ \mathsf{sat}\ [\![\texttt{letregion}\,\rho\,\texttt{in}\,e_1]\!]\ R$ — by 49,50,T&T typing

(52) $s_1, r \ \mathsf{sat}\ R, S^{\text{out}}_{e_1,R_1}, \varphi$ — by 48,$\varphi \subseteq \varphi_1$,22,27

(53) $s_1|_{Dom(s)}, r \ \mathsf{sat}\ R, S^{\text{out}}_{e_k,R}, \varphi$ — by 52,30

The result for [LETREGION] follows from 51 and 53.

Case 4 The operational semantics rule for [LETREGION_TT] is

(54) $z_{d'}$ fresh (to avoid recapture of dangling references)

(55) $d' =$ minimum color $\notin colors(Range|_\rho(r))$

(56) $r_1 = r[\rho \leftarrow z_{d'}]$

(57) $s_0 = s[z_{d'} \leftarrow \{\}]$

(58) $s_0, r_1, n \vdash e_1 : \mu, \varphi \cup \{\rho\} \rightarrow a_1, s_1$

(59) $s, r, n \vdash \texttt{letregion\_tt}\ \rho\ \texttt{in}\ e_1 : \mu, \varphi \rightarrow a_1, s_1|_{Dom(s)}$

By the definition of the region-based closure analysis and the constraint generation, we have

(60) $d =$ minimum color $\notin Range|_\rho(R)$

(61) $R_1 = R[\rho \leftarrow d]$

(62) $[\![\texttt{letregion}\,\rho\,\texttt{in}\,e_1]\!]\ R = [\![e_1]\!]\ R_1$

(63) $\forall c \in R(\varphi).S^{\text{in}}_{e_k,R}[c] = S^{\text{in}}_{e_1,R_1}[c]$

(64) $\forall c \in R(\varphi).S^{\text{out}}_{e_1,R_1}[c] = S^{\text{out}}_{e_k,R}[c]$

(65) $c_k \Longrightarrow S^{\text{in}}_{e_1,R_1}[d] = \mathbf{U} \ \wedge\ S^{\text{out}}_{e_1,R_1}[d] = \mathbf{D}$

(66) $\neg c_k \Longrightarrow S^{\text{in}}_{e_1,R_1}[d] = \mathbf{A} \ \wedge\ S^{\text{out}}_{e_1,R_1}[d] = \mathbf{A}$

The choice variable c_k associated with the letregion construct again expresses the kind of letregion chosen by the constraint solver: $c_k \Rightarrow \texttt{letregion}$, and $\neg c_k \Rightarrow \texttt{letregion\_tt}$. The proof for [LETREGION_TT] is very similar to [LETREGION], except for the state of region $z_{d'}$: Let $\varphi_1 = \varphi \cup \{\rho\}$.

(67)	$d = d'$	by 4,55,60
(68)	$r_1 \ \mathsf{sat}\ R_1$	by 67,4,56,61
(69)	$s_0, r, n \ \mathsf{sat}\ [\![\cdot]\!]\ R$	by 2,57,Lemma 6.8
(70)	$s_0, r_1, n \ \mathsf{sat}\ [\![\cdot]\!]\ R_1$	by 69,68,61
(71)	$\varphi \cup \{\rho\} \subseteq \mathit{Dom}(R[\rho \leftarrow d])$	by 5
(72)	$S^{\text{in}}_{e_1,R_1}[d] = \mathbf{A} \ \wedge\ S^{\text{out}}_{e_1,R_1}[d] = \mathbf{A}$	by 66,letreg. choice
(73)	$s_0, z_d \ \mathsf{sat}\ S^{\text{in}}_{e_1,R_1}[d]$	by 67,57,72

Now we do the inductive step on e_1.

(74)	$s_0, r \ \mathsf{sat}\ R, S^{\text{in}}_{e_1,R_1}, \varphi$	by 3,63,57
(75)	$s_0, r_1 \ \mathsf{sat}\ R_1, S^{\text{in}}_{e_1,R_1}, \varphi_1$	by 74,73
(76)	$s_1, a_1 \ \mathsf{sat}\ [\![e_1]\!]\ R_1$	by 59,70,75,induct.
(77)	$s_1, r_1 \ \mathsf{sat}\ R_1, S^{\text{out}}_{e_1,R_1}, \varphi_1$	by 59,70,75,induct.

It remains to be shown that the abstract value matches the concrete value and that the store satisfies the constraints.

(78)	$s_1, a_1 \ \mathsf{sat}\ [\![\texttt{letregion\_tt}\ \rho\ \texttt{in}\ e_1]\!]\ R$	by 76,62
(79)	$\mathit{Dom}(s_1) - \mathit{Dom}(s) = \{z_d\}$	by 57,67
(80)	$s_1\vert_{\mathit{Dom}(s)}, a_1 \ \mathsf{sat}\ [\![\texttt{letregion\_tt}\ \rho\ \texttt{in}\ e_1]\!]\ R$	by 78,79,T&T typing
(81)	$s_1, r \ \mathsf{sat}\ R, S^{\text{out}}_{e_1,R_1}, \varphi$	by 77,$\varphi \subseteq \varphi_1$,56,61
(82)	$s_1\vert_{\mathit{Dom}(s)}, r \ \mathsf{sat}\ R, S^{\text{out}}_{e_k,R}, \varphi$	by 81,64

The result for [LETREGION_TT] follows from 80 and 82.

Case 5 The operational semantics rule for [APP] is

(83) $s, r, n \vdash e_1 : \mu_1, \varphi_1 \rightarrow a_1, s_1$ where $\mu_1 = (\mu_2 \xrightarrow{\varphi_b} \mu, \rho)$

(84) $s_1, r, n \vdash e_2 : \mu_2, \varphi_2 \rightarrow a_2, s_2$

(85) $s_2(a_1) = \langle \lambda x.e_b : \mu_1', n_0, r_0 \rangle$ where $\mu_1' = (\mu_2' \xrightarrow{\varphi_b'} \mu', \rho')$

(86) $s_2, r_0, n_0[x \leftarrow a_2] \vdash e_b : \mu', \varphi_b' \rightarrow a_3, s_3$

(87) $s, r, n \vdash e_1\ e_2 : \mu, \varphi \rightarrow a_3, s_3$ where $\varphi = \varphi_1 \cup \varphi_2 \cup \varphi_b \cup \{\rho\}$

By the definition of the region-based closure analysis and the constraint generation, we have

(88)	For each $\langle \lambda x.e_b : \mu_1', R_0 \rangle \in [\![e_1]\!]\ R$
(89)	$[\![e_b]\!]\ R_0 \subseteq [\![e_1\ e_2]\!]\ R$
(90)	$[\![e_2]\!]\ R \subseteq [\![x]\!]\ R_0$
(91)	$A = R(\varphi_b) = R_0(\varphi_b')$
(92)	$\forall (c \in R(\varphi_1))\ S^{\text{in}}_{e_k,R}[c] = S^{\text{in}}_{e_1,R}[c] \ \wedge\ S^{\text{out}}_{e_1,R}[c] = S^{1}_{e_k} R[c]$
(93)	$\forall (c \in R(\varphi) - R(\varphi_1) \cup R\vert_\epsilon(\varphi))\ S^{\text{in}}_{e_k,R}[c] = S^{1}_{e_k,R}[c]$
(94)	$\forall (c \in R(\varphi_2))\ S^{1}_{e_k,R}[c] = S^{\text{in}}_{e_2,R}[c] \ \wedge\ S^{\text{out}}_{e_2,R}[c] = S^{2}_{e_k,R}$
(95)	$\forall (c \in R(\varphi) - R(\varphi_2) \cup R\vert_\epsilon(\varphi))\ S^{1}_{e_k,R}[c] = S^{2}_{e_k,R}[c]$
(96)	$\forall (c \in A)\ S^{2}_{e_k,R}[c] = S^{\text{in}}_{e_b,R_0}[c] \ \wedge\ S^{\text{out}}_{e_b,R_0}[c] = S^{\text{out}}_{e_k,R}[c]$
(97)	$\forall (c \in R_0\vert_\epsilon(\varphi_b'))\ S^{\text{in}}_{e_b,R_0}[c] = \mathbf{A}$
(98)	$\forall (c \in R(\varphi) - A \cup R\vert_\epsilon(\varphi))\ S^{2}_{e_k,R}[c] = S^{\text{out}}_{e_k,R}[c]$
(99)	$S^{2}_{e_k,R}[R(\rho)] = \mathbf{A}$

The proof for [APP] proceeds as follows: We use induction and Lemma 6.10 on expressions e_1 and e_2 to establish the state relation prior to evaluating the function body e_b. Then the state relation and the environment abstraction is established in the context of e_b and its abstract region environment R_0. Finally, the state relation after evaluating e_b is mapped back to the calling context e_k.

(100)	$\varphi_1 \subseteq \varphi$	by 87
(101)	$s, r \text{ sat } R, S^{\text{in}}_{e_1,R}, \varphi_1$	by 3,92,100
(102)	$s_1, a_1 \text{ sat } [\![e_1]\!]\ R$	by 83,2,101,induct
(103)	$s_1, r \text{ sat } R, S^{\text{out}}_{e_1,R}, \varphi_1$	by 83,2,101,induct
(104)	$s_1, r \text{ sat } R, S^{1}_{e_k,R}, \varphi$	by 3,103,100,92, 93,Lemma 6.10
(105)	$s_1, r, n \text{ sat } [\![\cdot]\!]\ R$	by 2,83,Lemmas 6.8, 6.9

Now similarly for e_2.

(106)	$\varphi_2 \subseteq \varphi$	by 87
(107)	$s_1, r \text{ sat } R, S^{\text{in}}_{e_2,R}, \varphi_2$	by 104,94,106
(108)	$s_2, a_2 \text{ sat } [\![e_2]\!]\ R$	by 84,105,107,induct
(109)	$s_2, r \text{ sat } R, S^{\text{out}}_{e_2,R}, \varphi_2$	by 84,105,107,induct
(110)	$s_2, r \text{ sat } R, S^{2}_{e_k,R}, \varphi$	by 104,109,106,94, 95,Lemma 6.10

In order to do the inductive step on e_b, we show

$s_2, r_0, n_0[x \leftarrow a_2] \text{ sat } [\![\cdot]\!]\ R_0$

$s_2, r_0 \text{ sat } R_0, S^{\text{in}}_{e_b,R_0}, \varphi'_b$

(111)	$\mathit{state}(s_2, r(\rho)) = \mathbf{A}$	by 110,99
(112)	$\langle \lambda x.e_b : \mu'_1, R_0\rangle \in [\![e_1]\!]\ R$	by 85,102,111
(113)	$r_0 \text{ sat } R_0$	by 102,112
(114)	$s_2, r_0, n_0 \text{ sat } [\![\cdot]\!]\ R_0$	by 102,Lemmas 6.8,6.9
(115)	$s_2, r_0, n_0[x \leftarrow a_2] \text{ sat } [\![\cdot]\!]\ R_0$	by 108,90,114
(116)	$\varphi_b \subseteq \varphi$	by 87
(117)	$r(\varphi_b) = r_0(\varphi'_b)$	by 83,85,Lemma 6.1
(118)	$R(\varphi_b) = R_0(\varphi'_b)$	by 117,4,113
(119)	$\forall(z_c \in r(\varphi_b))\ s_2, z_c \text{ sat } S^{2}_{e_k,R}[c]$	by 110,def. 6.6,116
(120)	$\forall(z_c \in r_0(\varphi'_b))\ s_2, z_c \text{ sat } S^{\text{in}}_{e_b,R_0}[c]$	by 119,117,118,91,96
(121)	$\forall(c \in R_0\vert_\epsilon(\varphi'_b))\ \mathcal{S}(S^{\text{in}}_{e_b,R_0}[c]) = \mathbf{A}$	by 97
(122)	$s_2, r_0 \text{ sat } R_0, S^{\text{in}}_{e_b,R_0}, \varphi'_b$	by 113,120,121

Using induction on e_b, we obtain:

(123)	$s_3, a_3 \text{ sat } [\![e_b]\!]\ R_0$	by 86,115,122,induct
(124)	$s_3, r_0 \text{ sat } R_0, S^{\text{out}}_{e_b,R_0}, \varphi'_b$	by 86,115,122,induct
(125)	$s_3, a_3 \text{ sat } [\![e_1\ e_2]\!]\ R$	by 123,89, Lemma 6.7

It remains to be shown that $s_3, r \text{ sat } R, S^{\text{out}}_{e_k,R}, \varphi$.

(126)	$\forall(z_c \in r_0(\varphi'_b))\ s_3, z_c \text{ sat } S^{\text{out}}_{e_b,R_0}[c]$	by 124,def 6.6
(127)	$\forall(z_c \in r(\varphi_b))\ s_3, z_c \text{ sat } S^{\text{out}}_{e_k,R}[c]$	by 126,117,118,96
(128)	$s_3, r \text{ sat } R, S^{\text{out}}_{e_k,R}, \varphi_b$	by 127,110,98
(129)	$s_3, r \text{ sat } R, S^{\text{out}}_{e_k,R}, \varphi$	by 110,128,116,117, 86,98,Lemma 6.10

The result for [APP] follows from 125 and 129.

Case 6 The operational semantics rule for [LETREC] is

(130) $a_1 = (r(\rho), o) \qquad o \notin \mathit{Dom}(s(r(\rho)))$

(131) $n' = n[f \leftarrow a_1]$

(132) $s_1 = s[a_1 \leftarrow \langle \lambda x.e_1 : \mu_1, \vec{\rho}, \epsilon.\vec{\varphi}, n, r\rangle]$ where μ_1 is the type of $\lambda x.e_1$

(133) $s_1, r, n' \vdash e_2 : \mu_2, \varphi_2 \rightarrow a_2, s_2$

(134) $s, r, n \vdash \texttt{letrec}\ f[\vec{\rho}, \epsilon.\vec{\varphi}](x)@\rho = e_1\ \texttt{in}\ e_2 : \mu_2, \varphi \rightarrow a_2, s_2$
where $\varphi = \varphi_2 \cup \{\rho\}$

By the definition of the region-based closure analysis and the constraint generation, we have

(135)	$[\![\texttt{letrec}\, f[\vec{\rho}, \vec{\epsilon.\varphi}](x)@\rho = e_1 \texttt{ in } e_2]\!]\, R = [\![e_2]\!]\, R$	
(136)	$\langle \lambda x.e_1 : \mu', \vec{\rho}, \vec{\epsilon.\varphi}, R\rangle = [\![f]\!]\, R$	
(137)	$S^{\text{in}}_{e_k,R}[R(\rho)] = \mathbf{A}$	
(138)	$\forall(c \in R(\varphi_2))\; S^{\text{in}}_{e_k,R}[c] = S^{\text{in}}_{e_2,R}[c]$	
(139)	$\forall(c \in R(\varphi_2))\; S^{\text{out}}_{e_2,R}[c] = S^{\text{out}}_{e_k,R}[c]$	
(140)	$\forall(c \in R(\varphi) - R(\varphi_2) \cup R\vert_\epsilon(\varphi))\; S^{\text{in}}_{e_k,R}[c] = S^{\text{out}}_{e_k,R}[c]$	

We first prove that the region for the polymorphic closure is allocated.

(141)	$\rho \in \varphi$	by 134
(142)	*state*$(s, r(\rho)) = \mathbf{A}$	by 3,137,141

Now we establish the environment relation for n' and use an inductive step on e_2.

(143)	s_1, r, n sat $[\![\cdot]\!]\, R$	by 2,132,Lemma 6.8
(144)	s_1, a_1 sat $[\![f]\!]\, R$	by 143,136,132
(145)	s_1, r, n' sat $[\![\cdot]\!]\, R$	by 143,144,131
(146)	s_1, r sat $R, S^{\text{in}}_{e_k,R}, \varphi$	by 3,132
(147)	$\varphi_2 \subseteq \varphi$	by 134
(148)	s_1, r sat $R, S^{\text{in}}_{e_2,R}, \varphi_2$	by 146,138,134
(149)	s_2, a_2 sat $[\![e_2]\!]\, R$	by 133,145,148,induct
(150)	s_2, r sat $R, S^{\text{out}}_{e_2,R}, \varphi_2$	by 133,145,148,induct
(151)	s_2, a_2 sat $[\![\texttt{letrec}\, f[\vec{\rho}, \vec{\epsilon.\varphi}]\ldots]\!]\, R$	by 149,135
(152)	s_2, r sat $R, S^{\text{out}}_{e_k,R}, \varphi_2$	by 150,139
(153)	s_2, r sat $R, S^{\text{out}}_{e_k,R}, \varphi$	by 146,152,147,133, 140,Lemma 6.10

The result for [LETREC] follows from 151 and 153.

Case 7 The operational semantics rule for [REGAPP] is

(154)	$n(f) = a$	
(155)	$s(a) = \langle \lambda x.e_1 : \mu', \vec{\rho}\,', \vec{\epsilon'.\varphi'}, n_0, r_0\rangle$ where $\mu' = (\tau', \rho')$	
(156)	$n' = n_0[f \leftarrow a]$	
(157)	$r' = r_0[\rho'_i \leftarrow r(\rho_i)]$ where $\rho'_i = \vec{\rho}\,'[i]$, $\rho_i = \vec{\rho}[i]$	
(158)	$r'' = r'[\epsilon'_i \leftarrow (r\vert_\rho(\varphi_i) - r'\vert_\rho(\varphi'_i)) \cup r\vert_\epsilon(\varphi_i)]$ where $\epsilon'_i = \vec{\epsilon}\,'[i]$, $\varphi_i = \vec{\varphi}[i]$	
(159)	$s_1 = s[a_1 \leftarrow \langle \lambda x.e_1 : \mu', n', r''\rangle]$	
(160)	$s, r, n \vdash f[\vec{\rho}, \vec{\varphi}]@\rho : \mu, \varphi \rightarrow a_1, s_1$ where $\varphi = \{\rho, \rho'\}$	

By the definition of the region-based closure analysis and the constraint generation, we have

(161)	$\{\langle \lambda x.e_1 : \mu', \vec{\rho}\,', \vec{\epsilon'.\varphi'}, R_0\rangle\} = [\![f]\!]\, R$	
(162)	$R' = R_0[\rho'_i \leftarrow R(\rho_i)]$	
(163)	$R'' = R'[\epsilon'_i \leftarrow (R\vert_\rho(\varphi_i) - R'\vert_\rho(\varphi'_i)) \cup R\vert_\epsilon(\varphi_i)]$	
(164)	$[\![f[\vec{\rho}, \vec{\varphi}]@\rho]\!]\, R = \{\langle \lambda x.e_1 : \mu', R''\rangle\}$	
(165)	$S^{\text{in}}_{e_k,R}[R(\rho)] = \mathbf{A}$	
(166)	$S^{\text{in}}_{e_k,R}[R(\rho')] = \mathbf{A}$	
(167)	$\forall(c \in R(\varphi))\; S^{\text{in}}_{e_k,R}[c] = S^{\text{out}}_{e_k,R}[c]$	

The main point in this case is to establish the relations r'' $\mathtt{sat}$ R'' and s_1, r'', n' $\mathtt{sat}$ $[\![\cdot]\!]$ R''. Note that the effect sets φ'_i are part of the arrow-effect $\epsilon'_i.\varphi'_i$ associated with the quantified effect variable ϵ'_i in the type τ'. We have:

(168)	r_0 $\mathtt{sat}$ R_0	by 2,155,161
(169)	$\mathit{state}(s, r(\rho')) = \mathbf{A}$	by 3,166
(170)	$\mathit{state}(s, r(\rho)) = \mathbf{A}$	by 3,165
(171)	$r''\vert_{\mathit{Dom}(r_0)} = r_0 = r\vert_{\mathit{Dom}(r_0)}$	by 157,158,`letrec`
(172)	$R''\vert_{\mathit{Dom}(R_0)} = R_0 = R\vert_{\mathit{Dom}(R_0)}$	by 162,163,`letrec`
(173)	$\{\rho'_1, \dots, \rho'_n\} \cap \mathit{Dom}\vert_\rho(r_0) = \emptyset$	by 171
(174)	$\{\rho'_1, \dots, \rho'_n\} \cup \mathit{Dom}\vert_\rho(r_0) = \mathit{Dom}\vert_\rho(r'')$	by 157,158
(175)	$\{\epsilon'_1, \dots, \epsilon'_m\} \cap \mathit{Dom}\vert_\epsilon(r_0) = \emptyset$	by 171
(176)	$\{\epsilon'_1, \dots, \epsilon'_m\} \cup \mathit{Dom}\vert_\epsilon(r_0) = \mathit{Dom}\vert_\epsilon(r'')$	by 157,158
(177)	$\forall(i \in \{1, \dots, n\})\; r''(\rho'_i) = r(\rho_i)$	by 157,158
(178)	$\forall(i \in \{1, \dots, n\})\; R''(\rho'_i) = R(\rho_i)$	by 162,163

We first prove $\forall(\rho, \rho' \in \mathit{Dom}\vert_\rho(R''))\; R''(\rho) = R''(\rho') \iff r''(\rho) = r''(\rho')$.

(179)	$\forall(\rho, \rho' \in \mathit{Dom}\vert_\rho(R_0))\; R''(\rho) = R''(\rho') \iff r''(\rho) = r''(\rho')$	by 171,172,173,168
(180)	$\forall(i, j \in \{1, \dots, n\})\; R''(\rho'_i) = R''(\rho'_j) \iff R(\rho_i) = R(\rho_j)$	by 178
(181)	$\forall(i, j \in \{1, \dots, n\})\; R''(\rho'_i) = R''(\rho'_j) \iff r(\rho_i) = r(\rho_j)$	by 180,4
(182)	$\forall(i, j \in \{1, \dots, n\})\; R''(\rho'_i) = R''(\rho'_j) \iff r''(\rho'_i) = r''(\rho'_j)$	by 181,177

179 and 182 leave us with the case

$$\forall(\rho \in \mathit{Dom}\vert_\rho(R_0))\; \forall(i \in \{1, \dots, n\})\; R''(\rho) = R''(\rho'_i) \iff r''(\rho) = r''(\rho'_i)$$

(183)	Let $\rho \in \mathit{Dom}\vert_\rho(R_0), i \in \{1, \dots, n\}$	
(184)	Case $\Longrightarrow$: Assume $R''(\rho) = R''(\rho'_i) = c$	
(185)	$R(\rho) = c$	by 183,184,172
(186)	$R(\rho_i) = c$	by 183,184,178
(187)	$r(\rho) = r(\rho_i)$	by 185,186,4
(188)	$r''(\rho) = r''(\rho'_i)$	by 187,183,171,177
(189)	$R''(\rho) = R''(\rho'_i) \Longrightarrow r''(\rho) = r''(\rho'_i)$	by 184,188
(190)	Case $\Longleftarrow$: Assume $r''(\rho) = r''(\rho'_i) = z_c$	
(191)	$r(\rho) = z_c$	by 183,171,190
(192)	$r(\rho_i) = z_c$	by 183,177,190
(193)	$R(\rho) = R(\rho_i)$	by 191,192,4
(194)	$R''(\rho) = R''(\rho'_i)$	by 193,183,172,178
(195)	$r''(\rho) = r''(\rho'_i) \Longrightarrow R''(\rho) = R''(\rho'_i)$	by 190,194
(196)	$\forall(\rho \in \mathit{Dom}\vert_\rho(R_0))\; \forall(i \in \{1, \dots, n\})\; R''(\rho) = R''(\rho'_i) \iff r''(\rho) = r''(\rho'_i)$	by 183,195
(197)	$\forall(\rho, \rho' \in \mathit{Dom}\vert_\rho(R''))\; R''(\rho) = R''(\rho') \iff r''(\rho) = r''(\rho')$	by 174,179,182,196

Now we prove $\forall(\rho \in \mathit{Dom}\vert_\rho(R''))\; R''(\rho) = \mathit{color}(r''(\rho))$.

(198)	$\forall(\rho \in \mathit{Dom}\vert_\rho(R_0))\; R''(\rho) = \mathit{color}(r''(\rho))$	by 171,172,168
(199)	$\forall(i \in \{1, \dots, n\})\; R''(\rho'_i) = \mathit{color}(r''(\rho'_i))$	by 177,178,4
(200)	$\forall(\rho \in \mathit{Dom}\vert_\rho(R''))\; R''(\rho) = \mathit{color}(r''(\rho))$	by 198,199,174

To establish r'' $\mathtt{sat}$ R'', it remains to be shown that $\forall(\epsilon \in \mathit{Dom}\vert_\epsilon(R''))\; R''(\epsilon) = \mathit{colors}(r''(\epsilon))$.

(201)	$\forall(\epsilon \in \mathit{Dom}\vert_\epsilon(R_0))\; R''(\epsilon) = \mathit{colors}(r''(\epsilon))$	by 171,172,168
(202)	Let $i \in \{1, \dots, m\}$	
(203)	Assume $c \in R''(\epsilon'_i)$	
(204)	$c \in (R\vert_\rho(\varphi_i) - R'\vert_\rho(\varphi'_i)) \cup R\vert_\epsilon(\varphi_i)$	by 203,202,163
(205)	Case 1: $c \in R\vert_\rho(\varphi_i) \;\wedge\; c \notin R'\vert_\rho(\varphi'_i)$	
(206)	$\exists z_c \in r\vert_\rho(\varphi_i)$ s.t. $\mathit{color}(z_c) = c$	by 4,205
(207)	$\nexists z_c \in r'\vert_\rho(\varphi'_i)$ s.t. $\mathit{color}(z_c) = c$	by 200,205,158,163
(208)	$\exists z_c \in (r\vert_\rho(\varphi_i) - r'\vert_\rho(\varphi'_i) \cup r\vert_\epsilon(\varphi_i))$ s.t. $\mathit{color}(z_c) = c$	by 206,207

(209) Case 2: $c \in R|_\epsilon(\varphi_i)$

(210) $\exists z_c \in r|_\epsilon(\varphi_i)$ s.t. $color(z_c) = c$ by 4,209

(211) $c \in R''(\epsilon'_i) \Longrightarrow c \in colors(r''(\epsilon'_i))$ by 203,208,210

(212) Assume $c \in colors(r''(\epsilon'_i))$

(213) $\exists z_c \in r|_\rho(\varphi_i) - r'|_\rho(\varphi'_i) \cup r|_\epsilon(\varphi_i)$ s.t. $color(z_c) = c$ by 212,158

(214) Case 1: $z_c \in r|_\rho(\varphi_i) \ \wedge \ z_c \notin r'|_\rho(\varphi'_i)$

(215) $c \in R|_\rho(\varphi_i)$ by 214, 4

(216) $c \notin R'|_\rho(\varphi'_i)$ by 200,214,163,158

(217) $c \in R|_\rho(\varphi_i) - R'|_\rho(\varphi'_i)$ by 215,216

(218) $c \in R''(\epsilon'_i)$ by 217,163

(219) Case 2: $z_c \in r|_\epsilon(\varphi_i)$

(220) $c \in R|_\epsilon(\varphi_i)$ by 219,4

(221) $c \in R''(\epsilon'_i)$ by 220,163

(222) $c \in colors(r''(\epsilon'_i)) \Longrightarrow c \in R''(\epsilon'_i)$ by 212,218,221

(223) $\forall (i \in \{1, \ldots, m\})\ R''(\epsilon'_i) = colors(r''(\epsilon'_i))$ by 202,211,222

(224) $\forall (\epsilon \in Dom|_\epsilon(R''))\ R''(\epsilon) = colors(r''(\epsilon))$ by 201,223,176

It follows that

(225) r'' sat R'' by 197,200,224

We now show s_1, r'', n' sat $[\![\cdot]\!]\ R''$ and the result for [REGAPP] follows easily.

(226) $s, n(f)$ sat $[\![f]\!]\ R$ by 2

(227) s, r_0, n_0 sat $[\![\cdot]\!]\ R_0$ by 171

(228) s_1, r_0, n_0 sat $[\![\cdot]\!]\ R_0$ by 227,159, Lemma 6.8

(229) s_1, r'', n_0 sat $[\![\cdot]\!]\ R''$ by 228,225,171,172,def 6.4

(230) s_1, a sat $[\![f]\!]\ R$ by 154,226,159

(231) s_1, r'', n' sat $[\![\cdot]\!]\ R''$ by 229,230,156

(232) s_1, a_1 sat $[\![f[\vec{\rho}, \vec{\varphi}]@\rho]\!]\ R$ by 159,164,231

(233) s_1, r sat $R, S^{\text{out}}_{e_k,R}, \varphi$ by 3,167

The result for [REGAPP] follows from 232 and 233.

Case 8 The operational semantics rule for [FREE_AFTER] is

(234) $s, n, r \vdash e_1 : \mu_1, \varphi_1 \rightarrow a_1, s_1$

(235) $r(\rho) = z_c$

(236) $state(s_1, z_c) = \mathbf{A}$

(237) $s_2 = s_1[z_c \leftarrow deallocated]$

(238) $s, n, r \vdash \texttt{free\_after}\ \rho\ e_1 : \mu_1, \varphi \rightarrow a_1, s_2$ where $\varphi = \varphi_1 \cup \{\rho\}$

By the definition of the constraint generation and obvious extension of the region-based closure analysis, we have

(239) $[\![\texttt{free\_after}\ \rho\ e_1]\!]\ R = [\![e_1]\!]\ R$

(240) $\forall (c \in R(\varphi_1))\ S^{\text{in}}_{e_k,R} = S^{\text{in}}_{e_1,R} \wedge\ S^{\text{out}}_{e_1,R} = S^1_{e_k,R}$

(241) $\forall (c \in R(\varphi) - R(\varphi_1) \cup R|_\epsilon(\varphi))\ S^{\text{in}}_{e_k,R} = S^1_{e_k,R}$

(242) $\forall (c \in R(\varphi)$ s.t. $c \neq R(\rho))\ S^1_{e_k,R} = S^{\text{out}}_{e_k,R}$

(243) $\langle S^1_{e_k,R}[R(\rho)], c_k, S^{\text{out}}_{e_k,R}[R(\rho)] \rangle_d$

(244) $R(\rho) \in R|_\epsilon(\varphi) \Longrightarrow c_k = false$

Note that $c_k = true$ since the `free_after` construct actually appears in the completion. We therefore have:

(245) $c_k = true$

(246) $R(\rho) \notin R|_\epsilon(\varphi)$ by 244,245

(247) $\mathcal{S}(S^1_{e_k,R}[R(\rho)]) = \mathbf{A}$ by 243,245,def 5.2

(248) $\mathcal{S}(S^{\text{out}}_{e_k,R}[R(\rho)]) = \mathbf{D}$ by 243,245,def 5.2

We first do the inductive step and then establish the new store relation.

(249) $s, r \ \mathsf{sat}\ R, S^{\text{in}}_{e_1,R}, \varphi_1$ by 3,240,238

(250) $s_1, a_1 \ \mathsf{sat}\ [\![e_1]\!]\ R$ by 234,2,249,induct

(251) $s_1, r \ \mathsf{sat}\ R, S^{\text{out}}_{e_1,R}, \varphi_1$ by 234,2,249,induct

(252) $s_1, a_1 \ \mathsf{sat}\ [\![\texttt{free\_after}\ \rho\ e_1]\!]\ R$ by 250,239

(253) $s_1, r \ \mathsf{sat}\ R, S^{1}_{e_k,R}, \varphi_1$ by 251,240

(254) $s_1, r \ \mathsf{sat}\ R, S^{1}_{e_k,R}, \varphi$ by 3,253,238,234, 241,Lemma 6.10

(255) $\mathit{state}(s_1, r(\rho)) = \mathbf{A}$ by 247,254

(256) $s_2, r(\rho) \ \mathsf{sat}\ S^{\text{out}}_{e_k,R}[R(\rho)]$ by 237,248

(257) $\forall (z_c \in r(\varphi) \text{ s.t. } \mathit{color}(z_c) \neq R(\rho))\ s_2, z_c \ \mathsf{sat}\ S^{\text{out}}_{e_k,R}[\mathit{color}(z_c)]$ by 254,242

(258) $\nexists z'_c \in r(\varphi) \text{ s.t. } z'_c \neq r(\rho)\ \wedge\ \mathit{color}(z'_c) = R(\rho)$ by 4,246

(259) $\forall (z_c \in r(\varphi))\ s_2, z_c \ \mathsf{sat}\ S^{\text{out}}_{e_k,R}[\mathit{color}(z_c)]$

(260) $257, 258, 256$

(261) $\forall (c \in R|_{\epsilon}(\varphi))\ \mathcal{S}(S^{\text{out}}_{e_k,R}[c]) = \mathbf{A}$ by 254,242,246

(262) $s_2, r \ \mathsf{sat}\ R, S^{\text{out}}_{e_k,R}, \varphi$ by 238,4,259,261

The result for [FREE_AFTER] follows from 252 and 262.

Case 9 The operational semantics rule for [ALLOC_BEFORE] is

(263) $r(\rho) = z_c$

(264) $\mathit{state}(s, z_c) = \mathbf{U}$

(265) $s_0 = s[z_c \leftarrow \{\}]$

(266) $s_0, n, r \vdash e_1 : \mu_1, \varphi_1 \rightarrow a_1, s_1$

---

(267) $s, n, r \vdash \texttt{alloc\_before}\ \rho\ e_1 : \mu_1, \varphi \rightarrow a_1, s_1$
where $\varphi = \varphi_1 \cup \{\rho\}$

By the definition of the constraint generation and obvious extension of the region-based closure analysis, we have

(268) $[\![\texttt{alloc\_before}\ \rho\ e_1]\!]\ R = [\![e_1]\!]\ R$

(269) $\forall (c \in R(\varphi) \text{ s.t. } c \neq R(\rho))\ S^{\text{in}}_{e_k,R} = S^{1}_{e_k,R}$

(270) $\langle S^{\text{in}}_{e_k,R}[R(\rho)], c_k, S^{1}_{e_k,R}[R(\rho)] \rangle_a$

(271) $\forall (c \in R(\varphi_1))\ S^{1}_{e_k,R} = S^{\text{in}}_{e_1,R} \wedge\ S^{\text{out}}_{e_1,R} = S^{\text{out}}_{e_k,R}$

(272) $\forall (c \in R(\varphi) - R(\varphi_1) \cup R|_{\epsilon}(\varphi))\ S^{1}_{e_k,R} = S^{\text{out}}_{e_k,R}$

(273) $R(\rho) \in R|_{\epsilon}(\varphi) \Longrightarrow c_k = \mathit{false}$

Note that $c_k = \mathit{true}$ since the `alloc_before` construct actually appears in the completion. We therefore have:

(274) $c_k = \mathit{true}$

(275) $R(\rho) \notin R|_{\epsilon}(\varphi)$ by 273,274

(276) $\mathcal{S}(S^{\text{in}}_{e_k,R}[R(\rho)]) = \mathbf{U}$ by 270,274,def 5.1

(277) $\mathcal{S}(S^{1}_{e_k,R}[R(\rho)]) = \mathbf{A}$ by 270,274,def 5.1

We first establish the new store relation.

(278) $\mathit{state}(s, r(\rho)) = \mathbf{U}$ by 3,276

(279) $s_0, r(\rho) \ \mathsf{sat}\ S^{1}_{e_k,R}[R(\rho)]$ by 265,263,277

(280) $\forall (z_c \in r(\varphi) \text{ s.t. } \mathit{color}(z_c) \neq R(\rho))\ s_0, z_c \ \mathsf{sat}\ S^{1}_{e_k,R}[\mathit{color}(z_c)]$ by 3,269

(281) $\nexists z'_c \in r(\varphi) \text{ s.t. } z'_c \neq r(\rho)\ \wedge\ \mathit{color}(z'_c) = R(\rho)$ by 4,275

(282) $\forall (z_c \in r(\varphi))\ s_0, z_c \ \mathsf{sat}\ S^{1}_{e_k,R}[\mathit{color}(z_c)]$ by 279,280,281

(283) $\forall (c \in R|_{\epsilon}(\varphi))\ \mathcal{S}(S^{1}_{e_k,R}[c]) = \mathbf{A}$ by 3,269,275

(284) $s_0, r \ \mathsf{sat}\ R, S^{1}_{e_k,R}, \varphi$ by 267,4,282,283

Now we can do the inductive step.

(285)	s_0, r, n sat $[\![\cdot]\!]\ R$	by 2,Lemma 6.9
(286)	s_0, r sat $R, S^{\text{in}}_{e_1,R}, \varphi_1$	by 284,271,267
(287)	s_1, a_1 sat $[\![e_1]\!]\ R$	by 266,285,286,induct
(288)	s_1, r sat $R, S^{\text{out}}_{e_1,R}, \varphi_1$	by 266,285,286,induct
(289)	s_1, a_1 sat $[\![\texttt{alloc\_before}\ \rho\ e_1]\!]\ R$	by 287,268
(290)	s_1, r sat $R, S^{\text{out}}_{e_k,R}, \varphi_1$	by 288,271
(291)	s_1, r sat $R, S^{\text{out}}_{e_k,R}, \varphi$	by 284,290,267,266, 272,Lemma 6.10

The result for [ALLOC_BEFORE] follows from 289 and 291.

7 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 4). The next step adds allocation and deallocation choice points and generates the allocation constraints (Section 5). 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. The implementation is roughly 5,500 lines plus the roughly 8,500 lines of code in the Tofte/Taplin base implementation.

Our annotations are orthogonal to the storage mode analysis mentioned in [TT94] and described in more detail in [Tof94] and in Section 2.6. 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. 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 10). 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 example, a straightforward tail-recursive factorial function, has a similar pattern. The improvement in this case is from $O(n)$ to $O(1)$ space complexity.

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, Samsort, Fibonacci, and Randlist. The memory usage graphs are shown in Figures 11, 12, 14, and 15, 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 11) has a curious feature: at times the memory usage drops below the amount needed to store the list! Our measurements count only heap memory usage. The evaluation stack is not counted, a measurement methodology consistent with [TT94]. Quicksort is not unusual in the behavior of using the evaluation stack to store values that would seem to belong in the heap. 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.

Results for a larger suite of programs are presented in Table 3. The source code for all of these programs may be found at `http://kiwi.cs.berkeley.edu/~nogc/examples/`.

	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.

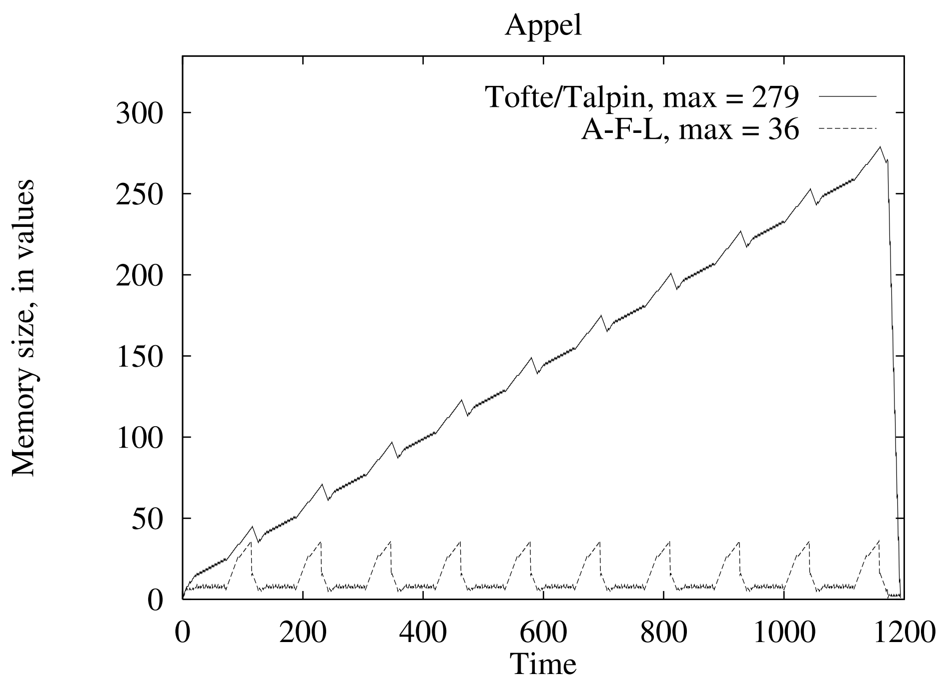


Figure 10: Memory usage in Appel example [App92] ($n = 10$).

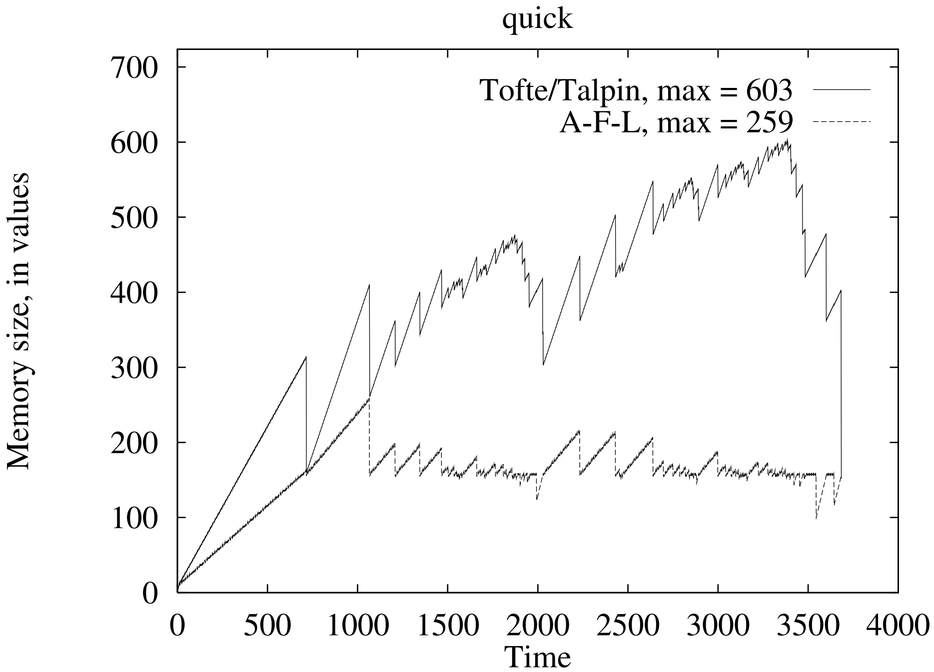


Figure 11: Memory usage in Quicksort example
(sort 50 element list of random integers).

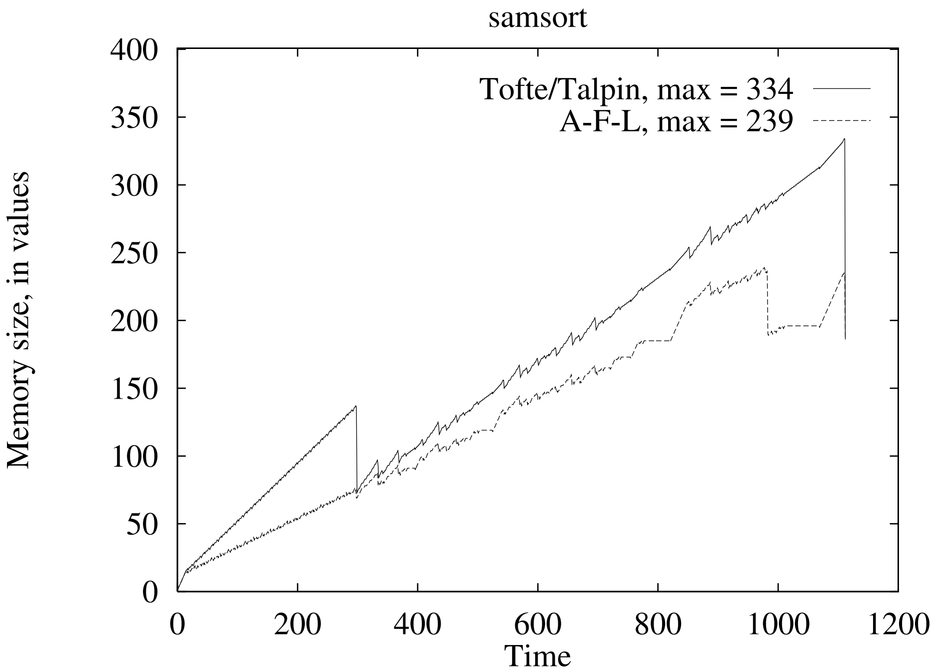


Figure 12: Memory usage in Samsort example
(smooth applicative mergesort 20 element list of random integers).

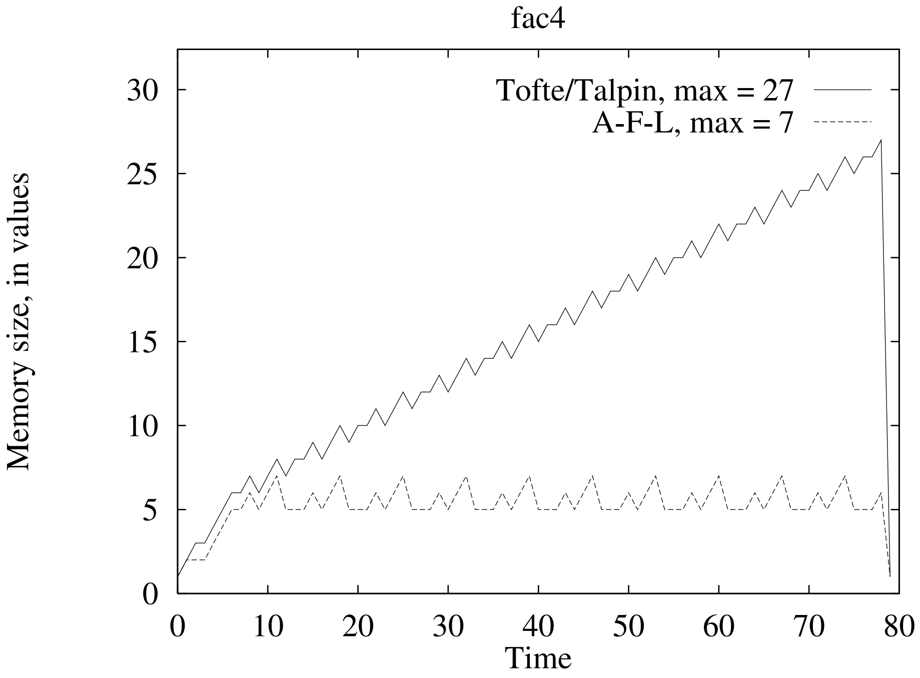


Figure 13: Memory usage in Fac4 example
(tail-recursive factorial of 10).

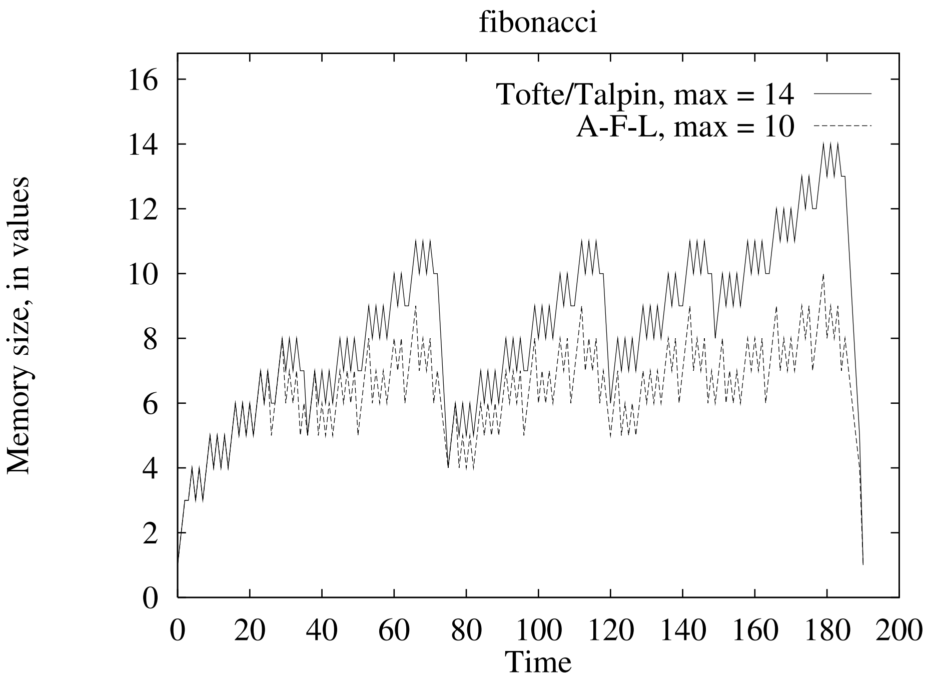


Figure 14: Memory usage in Fibonacci example
(recursive fibonacci of 6).

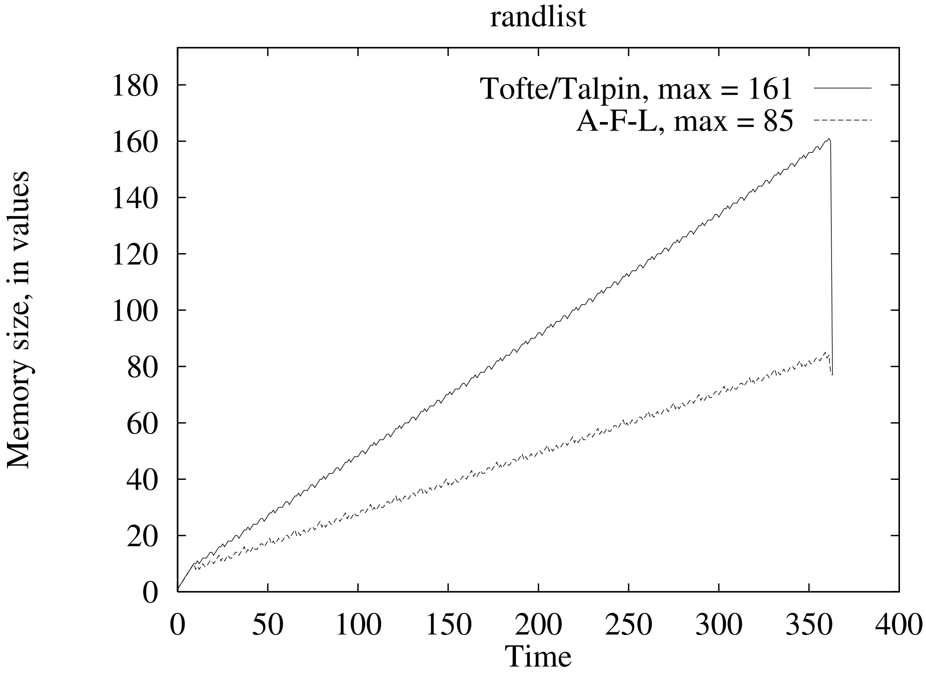


Figure 15: Memory usage in Randlist example
(generate 25 element list of random integers).

program name	total regions	total values	final regions	max regions (T-T)	max regions (A-F-L)	max values (T-T)	max values (A-F-L)	region ratio T-T/A-F-L	value ratio T-T/A-F-L
Appel	1005	1194	1	121	28	279	36	4.32	7.75
alias	9	11	4	7	5	7	4	1.40	1.75
append	19	27	11	6	6	12	11	1.00	1.09
big	52633	70163	1753	779	271	5018	2259	2.87	2.22
copy	16	33	10	10	10	19	13	1.00	1.46
count	48	77	11	7	4	34	32	1.75	1.06
crunch	347	684	31	42	12	377	47	3.50	8.02
downfrom	2115	3912	901	13	10	1807	1805	1.30	1.00
fac1	66	66	1	25	25	14	14	1.00	1.00
fibonacci	190	190	1	20	15	14	10	1.33	1.40
map	73	121	32	11	9	56	55	1.22	1.02
mergesort2	35519	46056	752	777	264	1767	1291	2.94	1.37
quick	19606	27334	752	770	75	4012	1259	10.27	3.19
randlist	289	363	77	90	12	161	85	7.50	1.89
relax	42727	73124	301	618	19	30913	409	32.53	75.58
samsort	819	1112	186	91	20	334	239	4.55	1.40

Table 3: Summary of results for larger suite.

7.1 Constraints

The current implementation handles quantified effect variables slightly differently than described in Section 5.1.5, for simplicity of implementation. The implementation does not explicitly represent effect variables in the region environments. Instead, it categorizes all effect variables into one of three classes: *unquantified, quantified monotonic,* and *quantified antimonotonic.* The class is determined by first finding the point in the program where the effect variable was quantified, and then inspecting the type quantified at that point. If there is no such point, it is unquantified. If there is such a point, and the effect variable does not appear in an anti-monotonic position in the type, then it is quantified monotonic. If the variable does appear in an antimonotonic position in the type, then it is quantified antimonotonic. The classification of effect variables is a syntactic property of the program; no analysis is required, which was the primary motivation for this approach.

At an application point, we examine the arrow effect for both the function subexpression in the application (i.e. the "caller effect" φ) and at the abstraction where the closure was generated (i.e. the "callee effect" φ'). If any of the effect variables in the caller or callee effect are quantified antimonotonic, then all region variables in both effects are constrained to be A, in both the application and the abstraction.

If none of these effect variables are quantified antimonotonic, then we conjecture that the set of colors obtained by applying the caller's region environment (i.e. R) to the caller effect is a superset of the corresponding set of colors obtained by applying the callee's region environment R' to the callee effect. In other words, we conjecture that:

$$R(\varphi) \supseteq R'(\varphi')$$

All the colors in $R(\varphi) \setminus R'(\varphi')$ are constrained to A as well.

We can only conjecture that this implementation is sound. Certainly, it works for all the examples we have tried. We plan to replace it with the global effect flow analysis described in Section 5.1.5. We expect the newer analysis to be more precise as well as better grounded theoretically, with a minimal increase in execution time. It is likely that implementing the full system in which abstract region environments map both region variables and effect variables would lead to an exponential blowup in the number of states generated in the region-based closure analysis.

Also, we have not yet implemented the phase of the constraint solver which deals with inconsistent partial labellings. In practice, we find that only highly contrived counterexamples lead to inconsistent partial labellings. For all other programs we tried, the constraint solver found a solution. Also, in all of these cases, there was no additional improvement to the solution that was apparent.

7.2 Comparison with storage mode analysis

Both our system and the storage mode analysis (Section 2.6) performed by the base Tofte/Talpin system are designed to improve memory utilization. It is instructive to compare the two.

First, the two optimizations are orthogonal. Storage mode analysis "resets" the region at various times during its lifetime. Our system allocates the region at some point (hopefully the latest possible) before the dynamic lifetime, and frees it at some point (hopefully the earliest) after the dynamic lifetime. The two optimizations do not interfere with each other. All of our measurements were peformed with storage mode analysis enabled.

It has been our experience that sometimes the storage mode annotation is effective by itself, sometimes our analysis is effective even without the storage mode annotation, and sometimes it is possible to fool both analyses. We illustrate this point by comparing the effectiveness of the two approaches in performing tail call optimization of variants of an iterative factorial function.

The first variant is successfully optimized by the storage mode analysis:

Example 7.1

```
letrec fac1 (p) =
    let n = snd p in
        let acc = fst p in
            if n = 0
                then p
                else fac1 (n * acc, n - 1)
        end
    end
in fac1(1, 10)
```

```
letregion r101
in
  letrec
      fac1[r102,r103,r104] (p:(((int,r102)*(int,r103)),r104)) attop r101 =
        let n:(int,r103) = snd p
          in let acc:(int,r102) = fst p
             in letregion r105
                in
                  if letregion r106 in (n=(0 attop r106)) attop r105 end
                  then p
                  else
                    letregion r107
                    in
                      fac1[ sat r102, sat r103, sat r104] attop r107
                      (((n*acc) sat r102,
                        letregion r108 in (n-(1 attop r108)) sat r103 end
                       ) sat r104
                      )
                    end
                end
             end
          end
      ) at r101
  in letregion r109,r110
     in
       fst
         letregion r111
         in
           fac1[ atbot r0, atbot r109, atbot r110] attop r111
           (((1 attop r0),(10 attop r109)) attop r110)
         end
     end
  end
end
```

Figure 16: Translation of *fac1* example.

The translation of this program is shown in Figure 16. Because p appears in the then branch, the input and output regions of the function are unified. In the translation of this program, the definition of *fac1* quantifies three region variables: one region holds both the input and output n, another holds both the input and output acc, and the third holds the pair.

It turns out that in this case, the storage mode analysis is able to determine that all three stores in the else branch can be done with an atbot annotation, meaning that they are reset to size one on each iteration.

If the order of the two arguments were reversed as in Example 7.2, then the store of the new value of n is forced to an attop annotation, because the $n * acc$ computation follows the $n - 1$ computation, so that the old value of n is still live. Thus, storage grows linearly with the number of iterations.

Example 7.2

```
letrec fac2 (p) =
    let n = fst p in
        let acc = snd p in
            if n = 0
                then p
                else fac2 (n - 1, n * acc)
        end
    end
in fac2 (10, 1)
```

Unfortunately, our system cannot improve Example 7.2 because the input and output regions of *fac2* are unified. Each of these regions is live during the entire *fac2* computation; there is no opportunity to free them earlier.

In our experience with the full system, we have found that separating regions as much as possible often improves results. With storage mode analysis alone, however, separating regions often makes things worse. Consider, for example, changing Example 7.1 so that the then branch performs a copy of p:

Example 7.3

```
letrec fac3 (p) =
    let n = snd p in
        let acc = fst p in
            if n = 0
                then (n + 0, acc + 0)
                else fac1 (n * acc, n - 1)
        end
    end
in fac1 (1, 10)
```

In general, separating regions improves the accuracy of the region inference algorithm, but in this case it also eliminates the opportunity for storage mode analysis to optimize the tail call. In Example 7.3, *fac3* uses separate regions to store the inputs and the outputs. It also makes use of recursive region polymorphism, so that the arguments to the recursive call are in another set of regions.

Now, the argument regions become dead as soon as the new arguments for the recursive call are built. However, there is no way for storage mode analysis to exploit this fact; it can only optimize when there is a store to a region. In *fac3*, there is no store to the regions containing the argument.

Our analysis, on the other hand, easily discovers that the argument regions become dead before the recursive call, and inserts deallocation annotations immediately after the last use of each argument value. Thus, it successfully optimizes the tail call recursion, allowing *fac3* to run in constant space.

The ability of our system to optimize the tail recursion of Example 7.3 survives switching the order of the arguments.

This example illustrates a number of points. First, it shows that our system is incomparable to storage mode analysis; neither optimization subsumes the other. Second, it shows that it is capable of making the overall optimization quality less sensitive to small program changes. At least in this example, it makes the dictum “separate regions where possible” much more useful. With storage mode annotation alone, this dictum is problematical, because sometimes it inhibits another optimization that was working better when the regions were unified.

It is difficult to generalize the experience with the factorial program, but the idioms are common enough that the effect should be significant.

```
letrec f = fn x =>
   if x = 1 then 41 else
   if x = 2 then 82 else
   if x = 3 then 123 else
   if x = 4 then 164 else
   if x = 5 then 205 else
   if x = 6 then 246 else
   if x = 7 then 30 else
   if x = 8 then 71 else
   if x = 9 then 112 else
   if x = 10 then 153 else
   0
in
   f 5
end
```

Figure 17: Source of *scale10* example.

7.3 Remote experimentation

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

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

The Web server allows experimenters to enter arbitrary programs, then reports the annotation performed, and also generates graphs similar to Figures 10-15.

7.4 Performance

This section describes some measurements of the performance of our system, in particular its scaling behavior. Table 4 presents the measurements. For each program, the following information is given:

- Number of textual lines of code.
- Number of *(expression, region environment)* pairs generated in the extended closure analysis.
- Number of constraints generated.
- Number of state variables.
- Number of choice variables.
- Time (in seconds) performing closure analysis.
- Time (in seconds) generating the constraints.
- Time (in seconds) solving the constraints.
- Total time spent (sum of the previous three plus housekeeping).

Overall, we find that our algorithm scales similarly to the Tofte/Talpin system. The implementations of both Tofte/Talpin system and our extensions are prototypes, containing numerous possibilities for improvement.

The programs *scale1, scale10,* and *scale100* are synthetic programs designed to characterize the scaling behavior of our system. The source for *scale10* is shown in Figure 17; the other two programs are defined similarly, varying the number of lines of code. This example exercises a performance problem in our implementation: it generates constraints for more regions than needed (in particular, all `letregion`-bound variables in scope within the current abstraction). Thus, the number of constraints generated is quadratic in the size of the program. A more faithful implementation of the constraint generation process described in Section 5.3 would yield a linear increase in the number of constraints for this program.

The worst case time complexity of the analysis is exponential. One simple attempt to elicit exponential scaling was unsuccessful because of arbitrary limits imposed by the prototype implementation of the Tofte/Talpin region inference algorithm.

One factor contributing to the complexity of our analysis is its interprocedural nature. The completion of a function depends on the context in which it is used. A straightforward intraprocedural version of our analysis would give very poor results, since a function would never be able to free or allocate an argument or result region without knowing whether its caller needed it to remain allocated.

A substantial amount of time is spent in the constraint solver, which is partly explained by the fact that the constraint generation and solution process is currently memory-bound. We feel that switching to an incremental solver (i.e. interleaving the processes of refining the partial solution and generating constraints) would improve both the speed and the amount of memory required.

All measurements were performed on a DEC AXP 3000/300 with 64MB of main memory, using SML/NJ 1.07.8. Times reported are user times according to the `checkCPUTimer` system function.

program name	number of lines	number of nodes	number of constraints	number of state variables	number of choice variables	time in closure analysis	time generating constraints	time solving constraints	total time
Appel	20	397	2393	2075	363	0.22	0.28	1.01	1.51
alias	23	40	95	49	75	0.01	0.01	0.02	0.05
append	26	113	329	226	96	0.01	0.01	0.01	0.03
copy	7	48	243	182	104	0.02	0.03	0.06	0.11
count	12	33	163	122	116	0.01	0.02	0.03	0.06
crunch	24	214	1770	1282	498	0.16	0.18	0.65	0.99
downfrom	11	107	679	479	247	0.06	0.06	0.18	0.30
fac1	10	38	152	112	107	0.01	0.02	0.03	0.06
fac2	10	38	150	110	107	0.01	0.02	0.03	0.06
fac3	10	118	546	437	137	0.06	0.06	0.17	0.29
fibonacci	10	166	991	824	144	0.09	0.11	0.41	0.61
map	13	101	628	425	224	0.05	0.06	0.17	0.28
mergesort2	100	2975	22682	19052	1546	3.08	3.65	22.47	29.22
quick	91	2772	16714	13081	917	8.00	2.46	11.81	22.29
randlist	32	165	873	645	236	0.11	0.10	0.25	0.46
relax	44	720	4700	3688	605	0.51	0.55	2.19	3.26
samsort	83	1030	8415	5664	1975	2.14	1.01	4.65	7.83
scale1	6	16	59	42	42	0.01	0.01	0.01	0.03
scale10	15	79	671	492	546	0.06	0.07	0.21	0.35
scale100	105	709	37976	31722	32316	5.27	23.11	39.59	68.25

Table 4: Summary of performance measurements.

8 Related Work

This section describes some of the relevant related work.

We have presented our system as an alternative to garbage collection. We would like to avoid some of the well known problems of garbage collection, including the large heap requirements, pauses (hence unsuitable for real-time or interactive applications), and interoperability. These problems have also been addressed by trying to improve garbage collection.

The problem of a large heap requirement is a consequence of *stop and copy* collectors [FY69]. In their simplest form, these collectors work by allocating all new storage in a contiguous *space*, i.e. by incrementing an allocation pointer. When the space is exhausted, all the live (i.e. reachable from the roots) data is copied into another space. Obviously, the memory requirement is at least twice as large as the reachable data. However, if the memory were exactly twice the size of the reachable data, then garbage collection would be required for every allocation. Garbage overhead decreases with as the ratio increases. A typical value for the ratio is five.

Generational garbage collection [LH83] significantly reduces the average time per garbage collection, but does not change the fundamental time/space tradeoff. The key idea of generational garbage collection is to separate objects by their lifetime; short-lived objects are treated separately from long-lived ones. Modern generational garbage collectors [App90] exhibit good overall performance, often with an overhead of under 40% (including the overhead of maintaing garbage collector invariants, and reasonable cache performance [DTM94]. However, it retains many of the problems mentioned above: pauses, memory consumption, and interoperability.

There are a number of techniques for reducing or eliminating the garbage collection pauses, including the incremental mark-sweep collector of Dijkstra et al [DLM+78] and the incremental copying collector of Baker [Bak78]. The latter technique relies on some additional computation (called a *read barrier*) for read operations. On modern architectures, it is generally agreed that the cost of a read barrier is prohibitive. Another efficiency problem is that, unlike in a traditional stop-and-copy collector, the older reachable data is interleaved with newly allocated data, reducing the locality of the data, thus making the cache performance worse.

Another approach to avoiding pauses is for the program to maintain a log of changes to memory, which is consumed by a collector thread (i.e. process sharing the same address space) [NO93]. This approach reduces the pauses, but does not eliminate them altogether. Also, the technique does not address the problems of memory usage or interoperability.

Conservative garbage collection [BW88] does address the interoperability issue. With this type of collector (based on mark and sweep rather than copying), the compiler need maintain no invariants for the garbage collector. Without any such invariants, the collector is not capable of precisely distinguishing between pointers and non-pointers. Rather, a conservative approximation is used, which may classify some non-pointers as pointers thereby forcing some garbage to be retained. All pointers are identified as pointers, which is required for correctness.

All of the techniques described so far are fully dynamic; they do not rely on any information about the individual program (with the possible exception of the knowledge that certain invariants are maintained). In the remainder of this section, we will discuss static analysis techniques which attempt to improve the memory behavior of programs.

One class of static analyses detects when a particular value is no longer reachable, and return it immediately to the free list. In a mark and sweep collecter, this can be valuable in delaying garbage collection, and thus reducing garbage collection overhead. In a copying garbage collector, there is no explicit free list. However, if the explicit deallocation is paired with a subsequent allocation (of a compatible number of bytes), the two operations can be merged into a *reuse*, avoiding the overhead of both the deallocation and the allocation. One example of this analysis is the reference count analysis of Hudak [Hud86], implemented in the Russel compiler by Hederman [Hed88]. In Hederman's implementation, the reference count analysis associates an abstract reference count with each program point and variable. When such a reference count passes from one to zero, the value bound to the variable may be deallocated.

The result of this optimization is to reduce the frequency of garbage collections. Garbage collection itself, with its attendant problems, is still required.

There are a number of other analyses that have the same goal, to identify some of the values which are no longer live, including sharing analysis [HJ90], which determines that some storage cells are not shared, so that when the last reference to the cell disappears it can be immediately reclaimed. All other cells are subject to garbage collection.

Another promising avenue is linear logic [Gir87]. Expressions with linear types are guaranteed to have only one reference to each value. If it is possible to infer that a function is linear, then other optimizations may be possible. Lafont proposed interaction nets [Laf90] as language design based on linear logic. Lafont claims that interaction nets can be implemented without garbage collection.

Ruggieri and Murtagh [RM88] proposed an analysis with the aim of eliminating garbage collection altogether. In their analysis, all heap-allocated data is divided into a stack of sub-heaps, one for each procedure activation record. When the procedure exits, all of the data in the associated activation record is deallocated. This runtime layout differs from a traditional stack in that allocations can be performed in any of the sub-heaps, not just the one at the top of the stack.

The most closely related work to ours is of course the Tofte/Talpin region inference [TT94], which is described in detail in Section 2.

9 Discussion and Conclusions

It remains an open question whether our system is a practical approach to memory management. The complexity of the region-based 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 $\Omega(n)$ less or better) than when annotated using the Tofte/Talpin system alone.

References

[App90] Andrew Appel. A runtime system. *Lisp and Symbolic Computation*, 3(4):343–380, November 1990.

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

[Bak78] Henry Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[BW88] Hans-J. Boehm and Mark Weiser. Garbage collection in an uncooperative enviroment. *Software Practice and Experience*, 18(9):214–221, September 1988.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proc. of the 4th Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

[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.

[DM82] L. Damas and Robin Milner. Principal type schemes for functinoal programs. In *Proc. of the 9th Annual ACM Symposium on Principles of Programming Languages*, pages 207–212, January 1982.

[DTM94] Amer Diwan, David Tarditi, and Eliot Moss. Memory subsystem performance of programs using copying garbage collection. In *Proc. of the 21st Annual ACM Symposium on Principles of Programming Languages*, pages 1–14, January 1994.

[Fra91] P. Fradet. Syntactic detection of single-threading using continuations. In *Functional Programming Languages and Computer Architecture, 5th ACM Conference (LNCS 523)*, pages 241–258, August 1991.

[FY69] Robert R. Fenichel and Jerome C. Yochelson. A LISP garbage-collector for virtual-memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Hed88] Lucy Hederman. Compile time garbage collection using reference count analysis. Master's thesis, Rice University, Department of Computer Science, 1988.

[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.

[Hen93] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2), April 1993.

[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.

[Hud86] Paul Hudak. A sematic model of reference counting and its abstraction. In *ACM Symposium on LISP and Functional Languages*, pages 351–363, January 1986.

[Laf90] Yves Lafont. Interaction nets. In *Proc. of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 95–108, January 1990.

[LH83] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[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.

[Sch85] D.A. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems*, 5(2):299–310, April 1985.

[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.

[TJ92] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region, and effect inference. *Journal of Functional Programming*, 2(3), 1992.

[Tof90] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1), November 1990.

[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.