

A direct approach to control-flow sensitive region-based memory management*

Fritz Henglein
The IT University of
Copenhagen
DENMARK
henglein@it.edu

Henning Makhholm
Dept. of Computer Science
University of Copenhagen
DENMARK
henning@makhholm.net

Henning Niss
Dept. of Computer Science
University of Copenhagen
DENMARK
hniss@diku.dk

ABSTRACT

Region-based memory management can be used to control dynamic memory allocations and deallocations safely and efficiently. Existing (direct-style) region systems that statically guarantee region safety—no dereferencing of dangling pointers—are based on refinements of Tofte and Talpin’s seminal work on region inference for managing heap memory in stacks of regions.

We present a unified Floyd-Hoare Logic inspired region type system for reasoning about and inferring region-based memory management, using a sublanguage of imperative region commands. Our system expresses and performs control-sensitive region management without requiring a stack discipline for allocating and deallocating regions. Furthermore, it captures storage mode analysis and late allocation/early deallocation analysis in a single, expressive, unified logical framework. Explicit region aliasing in combination with reference-counted regions provides flexible, context-sensitive early memory deallocation and simultaneously dispenses with the need for an integrated region alias analysis.

In this paper we present the design of our region type system, illustrate its practical expressiveness, compare it to existing region analyses, demonstrate how this eliminates the need for previously required source code rewritings for good memory performance, and describe automatic inference of region commands that give consistently better (or at least equally good) memory performance as existing inference techniques.

1. INTRODUCTION

Region-based memory management achieves efficiency by bulk allocation and deallocation of objects in memory. This is a well-known technique for improving efficiency of programs [5, 8, 9]. However, it is still as error prone and unsafe as the traditional C-like `malloc/free` framework. In particular, it is still hard to verify that a program does not contain space leaks and does not deallocate a region before the last use of data allocated in it. Tofte and Talpin [15] introduced a *region inference* system

*The order of authors on this paper carries no significance.

Permission to make digital or hard copies of part or all of this work or personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

PPDP 01 Florence, Italy

© ACM 2001 1-58113-388-x/01/09...\$5.00

and algorithm that inserts region-based deallocation operations automatically, eliminating the potential for human error.

In the Tofte/Talpin system (TT) the central expression is **let-region** ρ in $e[\rho]$. Operationally, upon entry a new region is allocated and bound to the lexically scoped variable ρ , expression $e[\rho]$ is evaluated, the region is deallocated, and the result of $e[\rho]$ is returned to the context of the evaluation (and the variable ρ disappears as its scope is left).

The *region safety* of the annotated program (*i.e.*, the property that it never accesses data after they have been deallocated) is guaranteed by a *region type system* where ordinary types are annotated with the region variables that control the values’ lifetimes. In TT, placing $e[\rho]$ in the **letregion** construct is permitted only if ρ occurs neither in the typing assumptions for $e[\rho]$ nor in its result type. This means that data can only be deallocated when they are manifestly unobservable by *any* context of $e[\rho]$, not just by the context $e[\rho]$ happens to have in the given program. Furthermore, the **letregion** construct must be aligned with the program’s expression hierarchy, so all region allocations and deallocations follow a stack discipline in unison with the original program’s expression structure.

The original inspiration for Tofte and Talpin’s region calculus came from work on *effect systems* [4, 12]. Recent research shows how such systems can be represented in the monadic lambda calculus [17, 10].

The TT system is efficient for smaller programs [15], but requires a number of extra analyses to give reasonable behavior for larger programs [2]. The problem is that in practice object lifetimes do not follow a stack discipline. In the ML Kit [13], a compiler for Standard ML using region-based memory management, a *storage mode analysis* is used to determine when it is safe to deallocate the data in a region—prior to deallocation of the region itself. Aiken, Fähndrich, and Levien [1] (AFL) employ another technique that separates region allocation and deallocation from introduction of region variables. The technique postpones allocation of a region until just before its first access, and deallocates it just after the last access. This leads to *late allocation* and *early deallocation* of regions. Characteristic of these variations is that they are built on top of the basic TT region framework.

Most previous work on region-based memory management has focused on typed lambda calculi, in particular Standard ML. That means integrating first-class functions and lexical closures into the framework from the outset. Being able to handle these is attractive theoretically (and for ML programmers), but it also entails a number of difficult technical problems, some of which still await a full solution. For example, it is not known whether the TT type system has principal types; a fact due directly to the way the system handles regions that become trapped in lexical closures.

In this paper we instead pursue the goal of constructing the best possible region architecture for *first-order* programs. We see several advantages to this approach. Most prosaic is the simple fact that it gets us longer with a given amount of work. Ignoring higher-order functions allows us to investigate different ways of managing regions without losing sight of their essential properties amidst the complex machinery needed to support higher-order programming. Also, there is a growing interest in using regions in other programming paradigms such as object-oriented or logic programming. In these contexts there is little point in forsaking a possibility simply because it will not work for ML.

We present a system that includes the first-order fragments of the earlier proposals as special cases and is arguably superior to them with respect to first-order programs. Our basic idea is to dispense with the hierarchic **letregion** construct and instead embed an *imperative* region sublanguage in the base language. Access to memory is managed through region variables pointing to memory regions. A region variable can be updated by assigning it a new region or a region from another region variable, and by releasing it. When a region variable is released, the region it points to is deallocated if and only if no other variable points to the same region. We use classic *reference counting* to determine when this situation occurs.¹

Functions are parameterized by three kinds of region variables: *constant region parameters*, *input region parameters*, and *output region parameters*. Data can be read from and written to constant region parameters, but the binding of the region parameter to its actual region cannot be updated in the body of the function. (In the TT system all region parameters are constant in this sense.) Input region parameters are updateable region variables that are passed into the function by the caller; the function takes responsibility for releasing them. Output region parameters are also updateable and are assigned by the function and returned to the caller.

We use a Floyd-Hoare Logic inspired region type system to keep track of the changes in the set of assigned region variables and the relation between region variables and data values. In the type system region variables are used to decorate the types of values. For example, $\Gamma, x:(\text{int}_B, \rho)$ expresses that x is bound to an address containing an infinite precision integer in the region currently bound to ρ . The region variable can be thought of as an access capability: As long as its binding does not change, the region will still exist, so it is safe to access x . However, if ρ 's binding is lost—by explicitly un-binding it or by passing it as an input region to a function—it cannot be used for access any more. This may happen even though x still exists in the value environment, in which case the loss of ρ is marked by changing the type environment to $\Gamma, x:(\text{int}_B, \top)$. The \top intuitively means that the address that x is bound to *may* not contain meaningful data anymore.

Our type system is expressive enough that it captures and extends TT- and AFL-style region annotated programs, including (mono-morphic) region resetting. At the same time it leads to simplicity since we need only be concerned with one framework, and not as for the Kit and AFL with both the original region system and extra analyses on top. Altogether, these principles make it less necessary, although still possible, to rewrite source programs to obtain good region performance. Such rewritings were previously required as illustrated below.

¹It is well known that reference counting cannot be used as the only memory-management strategy for a language that allows cyclic values to be formed, such as Standard ML with references. That problem does not apply to our system because region variables never occur *inside* regions, so there can be no cyclic region references.

Outline

Section 2 presents existing region techniques and describes how they handle the so-called Game of Life. Section 3 informally presents our framework, based on an imperative region sublanguage. Sections 4 and 5 contain our formal development of a region type system for an ML subset with imperative region management. Section 6 presents a negative result: There does not always exist a “best” way to add region annotations to the program. This result is extended to existing improvements of the TT system. Section 7 discusses implementation techniques, including an outline of a region inference algorithm for our system. Finally, Section 8 concludes.

2. BACKGROUND

For existing region systems there are known difficulties associated with iterative algorithms. The classic example is simulating the Game of Life for n generations, but the problems it raises apply to iteration of any function with a nontrivial domain.

The standard way of programming an iteration in a functional language is to use tail recursion:

```

fun nextgen(g) = ⟨read g; create and return new generation⟩
fun life(n, g) = if n = 0 then g
                else life(n - 1, nextgen(g))

```

We shall leave the details of *nextgen* unspecified here and in the following discussion (see the Kit distribution for an implementation; <http://www.it-c.dk/research/mlkit>). Furthermore we make the simplifying assumption that a single region holds all the pieces of a generation description, and ignore other non-essential details in the discussion.

2.1 Tofte/Talpin

In the basic TT system [15, 16], the Game of Life would be annotated as follows:

```

fun nextgen[ρ](g) = ⟨read g from ρ; create new gen. at ρ⟩
fun life[ρn, ρg](n, g) =
  if n = 0 then g
  else letregion ρnl
        in life[ρnl, ρg](n - 1) at ρnl, nextgen[ρg](g)

```

There are two major problems here. First, the recursive call to *life* is not a tail call anymore because it takes some work to deallocate a region at the end of **letregion**. This means that all the ρ_n^l regions will pile up on the call stack during the iteration and be deallocated only when the final result has been found.

Second, the *nextgen* function is forced to be a *region endomorphism*, that is, it must construct its result in the same region that contains its input. This means that the program has a serious space leak: all the intermediate-generation data will be deallocated only when the result of the iteration is deallocated.

2.2 The ML Kit

The ML Kit’s region implementation [2, 13] is based on the original TT system. After a TT region inference, a *storage mode analysis* infers when it is possible to *reset* regions safely. Resetting a region means deallocating all data in the region, while not deallocating the region itself.

With the Kit system one can rewrite the original Life program as follows to obtain better region behavior:

```

fun copy(g) = ⟨read g; make fresh copy⟩
fun lifel(p as (n, g)) = if n = 0 then p
                        else lifel(n - 1, copy(nextgen(g)))
fun life(p) = #2(lifel p)

```

where *copy* (whose body is omitted here for brevity) takes apart a generation description and constructs a fresh, identical copy. The ML Kit will then produce the region annotations

```

fun nextgen[ρ, ρ′](g) = ⟨read g from ρ; new gen. at ρ′⟩
fun copy[ρ′, ρ](g) = ⟨read g from ρ′; fresh copy sat ρ⟩
fun life′[ρn, ρg](p as (n, g)) =
  if n = 0 then p
  else life′[ρn, ρg] ((n − 1) atbot ρn,
    letregion ρ′g
    in copy[ρ′g, atbot ρg]
      (nextgen[ρg, ρ′g](g)))
fun life[ρn, ρg] p = #2 (life′[ρn, ρg] p)

```

Letting *life′* return the innermost *n* along with the result forces region inference to place all of the *n*’s in the same region ρ_n. A memory leak in ρ_n is prevented by the **atbot** allocation, whose effect is that the region ρ_n is reset prior to placing *n* − 1 in it. This solves the first of the problems with the original code.

The second problem is solved with the introduction of the *copy* function. This makes it possible to construct the new generation in a separate region ρ′; once the old generation is not needed anymore, the new generation is copied into ρ_g with the **atbot** mode which frees the old generation.

This solution does make it possible for *life* to run in constant space (assuming that the size of a single *g* is bounded), but it is by no means obvious that precisely these changes to the original program would improve the space behavior. Furthermore, inserting such region optimizations in the program impede maintainability because they obscure the intended algorithm. Finally, the CPU cycles spent on *copy* may be considered “waste” because they do not contribute to the computation *per se*.

Having region parameters in function definitions potentially leads to *region aliasing*. For example, in a (hypothetical) call *life′*[ρ, ρ](*n, g*), the region variables ρ_n and ρ_g in the function body will denote the same region. Therefore, one has to be careful when resetting regions. The analyses in the ML Kit solve this problem by conservatively approximating the set of aliased region variables [13, Section 12.2]. Our solution, as described later, is to explicitly count the number of references to a region at runtime.

2.3 Aiken/Fähndrich/Levien

The AFL system [1] extends the TT system in another direction², decoupling dynamic region allocation and deallocation from the introduction of region variables in the interpretation of the **letregion** construct.

In the AFL system, entry into a **letregion** block introduces a region variable, but does not allocate a region for it. During evaluation of the body of **letregion**, a region variable goes through precisely three states: unallocated, allocated, and finally deallocated. After a TT region inference, a constraint-based analysis—guided by a higher-order data-flow analysis for region variables—is used to insert explicit region allocation and deallocation commands which update the state of the region variables.

With this system the Life example can be improved by rewriting the original program to

```

fun copy(g) = ⟨read g; make fresh copy⟩
fun life(n, g) = if n = 0 then copy(g)
  else life(n − 1, nextgen(g))

```

(the only difference from the original program being that the base

²In [1] the system is presented as an extension to the Kit system. We think it is fruitful to view the two extensions as orthogonal.

case returns a fresh copy of its input rather than the input itself). This program is analyzed as³

```

fun nextgen[ρ, ρ′](g) =
  [[alloc ρ′]] ⟨read g from ρ; new gen. at ρ′⟩ [[free ρ]]
fun copy[ρ, ρ′](g) =
  [[alloc ρ′]] ⟨read g from ρ; fresh copy at ρ′⟩ [[free ρ]]
fun life[ρn, ρg, ρ′](n, g) =
  if n = 0 then [[free ρn]] copy[ρg, ρ′](g)
  else letregion ρ′n, ρ′g
    in life[ρ′n, ρ′g, ρ′] ([[alloc ρ′n]] (n − 1) at ρ′n [[free ρn]],
      nextgen[ρg, ρ′g](g))

```

Because deallocation of each region is done explicitly and not by **letregion**, the body of **letregion** is a tail call context, and the regions containing the old *n* and *g* can be freed as soon as *n* − 1 and *nextgen*(*g*) have been computed. Without rewriting the original program this would not be the case, because a function must either *always* free one of its input regions or *never* do it.

2.4 Walker/Crary/Morrisett

The development of the Calculus of Capabilities (CC) by Walker, Crary and Morrisett [3, 18] was motivated by the goal of using region-based memory management to certify the memory safety of object code. This leads to two unique characteristics of the CC system. First, it does not come with a region inference algorithm, only an execution model and a region type system; Walker et al. suggest that the TT or AFL region inference can be used as a front end for their system. Second, the region type system is surprisingly strong and expressive, and offers possibilities that by far transcend what the AFL region inference is capable of utilizing.

The strength of the CC system is mandated by the fact that it must work in a low-level context where transfer of control into and out of functions must be reasoned about separately. Therefore the system is defined in terms of a continuation-passing-style language. The need to be able to type CPS-transformed versions of TT-annotated program translates into severe demands on the type system. When the CPS transformation encounters a function call in the direct-style original program, the caller’s entire context must be reified as a continuation function. This means that the CC type system must be able to pack into a function type all the region-state information that other systems blithely assume can survive a function call.

The CC type system thus ends up being highly expressive and complex, generalizing region variables to “capability variables” which are subject to System-F-style polymorphism and bounded quantification. It can express very subtle control-flow dependencies and opens vast possibilities for interaction between region-based memory management and higher-order programming whose true limits are not yet well understood.

Our system appears to have comparable expressive power to CC when the latter is only applied to CPS-transformed first-order programs. Most of what our system allows will be accepted by CC after CPS-transformation—the exception being reference counting for regions, which can be added to the CC system without conceptual difficulties. Conversely, we believe that any CC-correct annotation of a CPS-transformed first-order program can be mapped back to an equivalent program that our system accepts.

Expressing a region type system in direct-style (as TT, the Kit, AFL and we do) has the advantage of being closer to what programmers actually write—and experience strongly suggests that

³The syntax here is not identical with the one used in [1]; for example, AFL write “**free_after** ρ *e*” for what we write as “**[[free** ρ]]”.

programmers need to be able to understand the basics of what the region type system does, or they won't be able to make good use of it—but also comes at a technical price. For example, our model and typing rule for function calls are roughly twice as complex as it would be in a CPS formulation, because a direct-style function call includes two transfers of control instead of just one.

3. IMPERATIVE REGION MANAGEMENT

Our fundamental idea is to embed an *imperative* sublanguage for manipulating region variables into the “base” language. This language has four kinds of operations.

- $\llbracket \text{new } \rho \rrbracket$: allocates a new region with reference count 1 and assigns it to ρ ; ρ must not be assigned a region before this.
- $\llbracket \text{release } \rho \rrbracket$: decrements the reference count of the region assigned to ρ and then makes ρ unassigned; the region is deallocated if (and only if) the reference count drops to 0.
- $\llbracket \rho' := \text{alias } \rho \rrbracket$: assigns the region in ρ to ρ' and increments its reference count; ρ' must be unassigned and ρ must be assigned a region.
- $\llbracket \rho' := \rho \rrbracket$: this *renaming* operation is equivalent to the sequence $\llbracket \rho' := \text{alias } \rho \rrbracket \llbracket \text{release } \rho \rrbracket$. It has no net effect on the reference count of the region originally bound to ρ . (One can think of this as a *linear* variant of region assignment in the previous case.)

Region operations can be inserted at arbitrary places in the source program. When the base language is functional, as is the case in our formal development later in the paper, this means that one or more region operations can be attached to each expression as a pre- or post-operation. Thus, for example, an effect similar to the TT system's “**letregion** ρ **in** e ” can be simulated in our system by “ $\llbracket \text{new } \rho \rrbracket e \llbracket \text{release } \rho \rrbracket$ ”, but **new** and **release** operations are not required to match up in this way.

The internal ordering of region operations is not subject to any hierarchical discipline. The “scope” of a region variable, if one insists on using such a concept, reaches from when it is assigned using a **new**, **alias**, or renaming operation and until it is consumed by **release** or renaming. The scope need not have a “nice” shape (the same region variable can toggle arbitrarily back and forth between bound and unbound), and there are no ties between the scope of different region variables.

The two exceptions to the general rule that the region sublanguage is independent of the structure of the base language are conditionals and function call.

In a conditional, the two branches of a conditional must lead to the same region variables being assigned at the end, but they need not do it by the same means. Thus both of

```
if ... then ...  $\llbracket \text{release } \rho \rrbracket \llbracket \text{new } \rho \rrbracket$  ... else ...
if ... then  $\llbracket \text{new } \rho' \rrbracket$  ...  $\llbracket \text{release } \rho \rrbracket$  else  $\llbracket \rho' := \rho \rrbracket$  ...
```

are legal but

```
if ... then ...  $\llbracket \text{release } \rho \rrbracket$  else ...
```

is not.

Function calls are more complex (and the reader may want to refer to the example in Sections 3.1 and 3.2 while digesting the general discussion that follows). The region annotation for a function call specifies three lists of *actual region parameters*:

$$[c; \rho_1, \dots, \rho_k; i; \rho'_1, \dots, \rho'_m; o; \rho''_1, \dots, \rho''_n]$$

Here the ρ_i 's are the *constant* parameters, the ρ'_i 's are the *input* region parameters, and the ρ''_i 's are the *output* region parameters.

The constant regions and input regions must be bound before the function call. When the called function starts executing, the bindings are transferred to region variables listed as formal region parameters in the function definition. No other region variables are bound at the beginning of the function body. The function body may not change the bindings of its formal constant parameters, but may do anything with the formal input region parameters. After the function body has been executed, the bound region variables must be exactly the formal constant parameters plus a set of formal output region parameters. The bindings of these are then transferred back to the caller's actual constant regions and output regions.

If any region variable that is neither an actual constant parameter nor an actual input parameter is bound before the function call, it is hidden from the called function but reappears in the caller after the call. It must not have the same name as an actual output parameter.

An actual input region parameter must not be identical to another actual input parameter or actual constant parameter in the same call. Likewise, each actual output region parameter must be distinct from other actual output parameters and from the constant parameters. However, two actual constant region parameters can be identical. These rules make sure that the region reference counts are maintained correctly: Any region aliasing must be done by explicit **alias** operations, and previously bound region variables may not be rebound without first being explicitly released. The relaxation for constant actuals is sound because the called function may not change the bindings of the corresponding formals; thus the bindings of the constant actuals is always the same before and after the call.

Note that an actual input region parameter loses its binding during the call (but can get a new one if it is used as an actual output parameter too).

The following examples illustrate the power of input and output regions. Seeing the need for constant regions requires knowledge of our type system, so we defer the rationale to the comments to our typing rule for function calls.

3.1 Game of Life in our system

With imperative region management it is possible to handle the Game of Life with no rewriting at all. In our system we get:

```
fun nextgen [i; \rho; o; \rho'] (g) =
   $\llbracket \text{new } \rho' \rrbracket$   $\langle \text{read } g \text{ from } \rho; \text{new gen. at } \rho' \rangle \llbracket \text{release } \rho \rrbracket$ 
fun life [i; \rho_n, \rho_g; o; \rho'] (n, g) =
  if n = 0 then  $\llbracket \text{release } \rho_n \rrbracket$  g  $\llbracket \rho' := \rho_g \rrbracket$ 
  else life [i; \rho'_n, \rho'_g; o; \rho']
      (( $\llbracket \text{new } \rho'_n \rrbracket$  (n - 1) at  $\rho'_n$   $\llbracket \text{release } \rho_n \rrbracket$ ,
        nextgen [i; \rho_g; o; \rho'_g] (g))
```

where each iteration of *life* decides for itself whether to release the region it gets as its second parameter or to return it back to the caller.

The $\llbracket \rho' := \rho_g \rrbracket$ operation serves the same purpose as the *copy* operation in the AFL solution, but has no runtime cost. Indeed, even the $\llbracket \rho' := \rho_g \rrbracket$ operation itself is superfluous in this particular example; as an alternative all occurrences of ρ' could simply be changed to ρ_g . Then ρ_g would appear as a formal output region as well as a formal input region, but that does not matter; there is no *a priori* relation between the input and output regions.

3.2 Context-sensitive early deallocation

To see the benefits of the $\llbracket \rho' := \text{alias } \rho \rrbracket$ operation, consider the function

```

fun twolife ( $g, n, m$ ) = let  $g' = \textit{life}(n, g)$ 
                         $g'' = \textit{life}(m, g')$ 
                        in ( $g', g''$ )

```

In our system this can be annotated as

```

fun twolife [i:  $\rho_g, \rho_n, \rho_m$ ; o:  $\rho'_g, \rho''_g$ ]( $g, n, m$ ) =
  let  $g' = \textit{life}$  [i:  $\rho_n, \rho_g$ ; o:  $\rho'_g$ ]( $n, g$ )
     $g'' = \llbracket \rho'_g := \textit{alias } \rho'_g \rrbracket \textit{life}$  [i:  $\rho_m, \rho'_g$ ; o:  $\rho''_g$ ]( $m, g'$ )
  in ( $g', g''$ )

```

The second call to *life* is prevented from deallocating g' because the binding in ρ'_g keeps the region alive after *life* releases ρ'_g . All other invocations of *life* still deallocate the generation input.

This behavior is not possible in the AFL and CC systems: If any call to *life* needs to preserve the region where some of its inputs live, *no* call can be allowed to deallocate its argument.

The ML Kit system can handle certain instances of this situation through a device called *storage-mode polymorphism*, which lets a function either reset one of its region parameters or not reset it, according to instructions passed by the caller.

The *twolife* example is not handled well by the ML Kit, however, since the Kit-friendly *life* implementation from Section 2.2 depends on being allowed to reset the region with the generation data.

A combination of the AFL and Kit solutions would be able to obtain a behavior similar to our system for *twolife*. It would need to be guided by an explicit annotation from the programmer, though, because by default the Kit attempts to reset a region only when something gets allocated in it.

3.3 Simulation of the Kit and AFL systems

It is immediately clear that our system can express everything allowed by the (first-order fragment of the) TT system. We now argue that a similar relation holds between our system and the Kit and AFL systems.

Region resetting in the Kit can be simulated in our system simply by the combination “ $\llbracket \textit{release } \rho \rrbracket \llbracket \textit{new } \rho \rrbracket$ ”. The Kit prohibits resetting if any live value has a type that mentions the region variable. When this is not the case, our region type system will also allow the **release-new** combination. Resetting a region parameter in the Kit requires that the actual parameter was not live at the call site, in which case our system would allow the parameter to be converted from a constant parameter to a pair of (identically named) input and output regions such that the function can still reset the region.

What our system *cannot* simulate directly is the storage-mode polymorphism we mentioned in Section 3.2. We expect that most storage-polymorphic region parameters can be simulated by a combination of an input region for use before the resetting and a constant parameter or output region for use after the resetting. However, this does not always work in cases like

```

fun nonneg [ $\rho$ ]( $x$ ) = if  $x < 0_{\mathbb{B}}$  at  $\rho$  then ( $0 \textit{ sat } \rho$ ) else  $x$ 

```

where the region is only reset in some execution paths through the function. Region renaming in the **else** branch might work here, but only if all callers can accept that *nonneg* decides which region the result will be allocated in.

Simple cases of early deallocation in the AFL system is easily modeled by our input region parameters. It is less clear whether our system can model precisely all the effects of the AFL system’s handling of region aliasing. Consider, for example, the following program fragment with AFL annotations:

```

fun  $f$  [ $\rho_x, \rho_y$ ]( $x, y$ ) = let  $b = x > y$   $\llbracket \textit{free } \rho_x \rrbracket$ 
                        in  $\langle \textit{code not depending on } x \textit{ and } y \rangle$ 

```

```

fun  $g$  [ $\rho$ ]( $x$ ) =  $f$  [ $\rho, \rho$ ]( $x, x$ )
fun  $h$  [ $\rho$ ]( $x$ ) = letregion  $\rho'$ 
                in  $f$  [ $\rho, \rho'$ ]( $x, (x + 1 \llbracket \textit{alloc } \rho' \rrbracket)$  at  $\rho'$ )  $\llbracket \textit{free } \rho' \rrbracket$ 

```

The best emulation of this in our system is

```

fun  $f$  [i:  $\rho_x, \rho_y$ ]( $x, y$ ) =
  let  $b = x > y$   $\llbracket \textit{release } \rho_x \rrbracket \llbracket \textit{release } \rho_y \rrbracket$ 
  in  $\langle \textit{code not depending on } x \textit{ and } y \rangle$ 
fun  $g$  [i:  $\rho$ ]( $x$ ) =  $\llbracket \rho' := \textit{alias } \rho \rrbracket f$  [i:  $\rho, \rho'$ ]( $x, x$ )
fun  $h$  [i:  $\rho$ ]( $x$ ) =  $f$  [i:  $\rho, \rho'$ ]( $x, (x + 1 \llbracket \textit{new } \rho' \rrbracket)$  at  $\rho'$ )

```

In our version, f will delete both of its input regions after the comparison, whereas the AFL version cannot deallocate both ρ_x and ρ_y lest the same region would be deallocated twice when f is called from g . We conjecture that the pattern in this example holds generally: that an AFL annotation can always be translated into a program in our system that gives at least as good lifetimes of all memory blocks.

4. SOURCE AND TARGET LANGUAGES

For our formal development we consider a small ML subset called Fun. The primitive data types are integers in unboxed (int) and boxed (int_B) versions. Our main reason for having boxed integers is that they make it easy for small examples to require heap allocation; one can also think of them as infinite-precision integers. We also include constructed data in the form of binary pairs and lists. The language contains the usual variables, **let**-expressions, constructors and destructors for data, and functions. Fun is “almost higher order” in that functions are first-class values, but it is not possible to create closures. Thus function values behave similarly to C’s “function pointers”.

Fun is the source language in which the programmer writes her programs. Region inference (Section 7) takes a Fun-program and emits a region annotated program. The resulting program is written in the target language RegFun, which is Fun with our imperative region sublanguage embedded and explicit region annotations on each allocation.

Figure 1 shows the syntax of RegFun, from which the syntax of Fun can also be inferred by ignoring the region constructs. An evaluation semantics for RegFun is given in [6].

5. A REGION TYPE SYSTEM

We now define a region type system for RegFun. As for the original TT region type system, its purpose is to only accept programs that are memory safe. The region type system can thus be used to guide the design (and ultimately help prove the correctness) of region inference algorithms, and as a contract between the region inference and later phases of the implementation.

Figure 2 shows the semantic objects used in the type system. The various constructions will be explained along with the rules that use them. In addition to the operations, we use the set $\textit{frv}(\mu)$ of free region variables of a region type μ (where, by definition, $\textit{frv}(\sigma) = \emptyset$). By point-wise extension, the free variables of an environment Γ is called $\textit{frv}(\Gamma)$.

Our typing judgments keep track of available regions and data. The judgments are inspired by Floyd-Hoare Logic and take the form of pre- and post-descriptions of the runtime state. The main typing judgment has the form

$$\Psi \vdash \{ \Delta / \Gamma \} e : \mu \{ \Delta' / \Gamma' \}$$

Here Δ and Δ' are the sets of updateable bound region variables before and after the evaluation. They are complemented by Ψ

$R \in \langle \text{RegOpr} \rangle$	$::= \text{new } \rho \mid \rho := \text{alias } \rho \mid \text{release } \rho \mid \rho := \rho$	(region operations)
$e \in \langle \text{Expr} \rangle$	$::= \llbracket R \rrbracket e \mid e \llbracket R \rrbracket$	(region operations)
	$\mid \text{let } x = e \text{ in } e \mid x$	(variables and bindings)
	$\mid n \mid (e \otimes e) \mid \text{if } e \neq 0 \text{ then } e \text{ else } e \mid n_{\text{B}} \text{ at } \rho \mid (e \otimes e) \text{ at } \rho$	(integer operations)
	$\mid (e, e) \text{ at } \rho \mid \pi_1 e \mid \pi_2 e$	(pairs)
	$\mid [] \text{ at } \rho \mid e :: e \text{ at } \rho \mid \text{case } e \text{ of } [] \Rightarrow e \text{ or } x :: x \Rightarrow e$	(lists)
	$\mid e \llbracket \xi \rrbracket e$	(function calls)
$F \in \langle \text{FunDef} \rangle$	$::= \text{fun } f \llbracket \xi \rrbracket x = e$	(function definitions)
$d \in \langle \text{DeclSeq} \rangle$	$::= \langle \text{empty} \rangle \mid d F$	(declarations)
$P \in \langle \text{Program} \rangle$	$::= \text{let } d \text{ in } e$	(programs)
$\xi \in \langle \text{RegParms} \rangle$	$::= \text{c}; \vec{\rho}; \text{i}; \vec{\rho}; \text{o}; \vec{\rho}$	
$\rho \in \langle \text{RegVar} \rangle$	$f, x \in \langle \text{VarName} \rangle$	$n \in \langle \text{IntConst} \rangle$
		$n_{\text{B}} \in \langle \text{BigNumConst} \rangle$
		$\otimes \in \langle \text{Operator} \rangle$

Figure 1: Abstract syntax of RegFun (an ML subset with explicit region annotations)

$\mu \in \langle \text{RegionType} \rangle$	$::= \text{int} \mid (\text{int}_{\text{B}}, p) \mid (\mu \times \mu, p)$	
	$\mid (\mu \text{ list}, p) \mid \sigma$	
$\sigma \in \langle \text{FunType} \rangle$	$::= \forall \vec{\rho}. (\exists \vec{\rho}. \mu) \rightarrow (\exists \vec{\rho}. \mu)$	
$p \in \langle \text{Place} \rangle$	$::= \perp \mid \rho \mid \top$	
$\Gamma \in \langle \text{VarEnv} \rangle$	$::= 0 \mid \Gamma, x; \mu \mid \Gamma, *; \mu$	
$\Psi, \Delta \in \langle \text{RegSets} \rangle$	$= P_{\text{fin}}(\langle \text{RegVar} \rangle)$	

$\vec{\rho}$ is an ordered sequence of region variables

$\Delta \uplus \Delta' \stackrel{\text{def}}{=} \Delta \cup \Delta'$	(when $\Delta \cap \Delta' = \emptyset$)
$\Gamma(x) \stackrel{\text{def}}{=} \text{case } \Gamma \text{ of}$	$\begin{cases} \Gamma', x; \mu \mapsto \mu \\ \Gamma', x'; \mu \mapsto \Gamma'(x) & (x \neq x') \\ \Gamma', *; \mu \mapsto \Gamma'(x) \\ 0 & \mapsto (\text{undefined}) \end{cases}$

$\llbracket \rho_i \rrbracket_i$ converts a sequence to a set (when the ρ_i 's are distinct)

Figure 2: Semantic objects and associated operations used in the region type system

which is the set of formal constant parameters in the current function. They behave mostly like the region variables in Δ , except that their bindings cannot change. Therefore, Ψ stays the same throughout an expression and needs only occur once in a judgment. Ψ and Δ (or Ψ and Δ') are always disjoint; their union is the set of bound (or “live”) regions variables before (or after) evaluating e .

Though the actual *values* in the runtime environment ideally do not change while evaluating an expression, embedded region operations may change the bindings of the region variables that allow access to them, such that their region-annotated type must be updated. Therefore the judgment contains two different type environments Γ and Γ' which both describe the same runtime environment, but relative to the region-variable bindings before and after the evaluation of e , respectively. The typing rules maintain the invariant that Γ and Γ' have the same structure—that is, the only differences are in the $\langle \text{Place} \rangle$'s inside types. The idea of tracking state changes in types is not new; an early example is the tpestates system [11]. However, the tpestates system is not based on a specification with pre- and post-descriptions as above, instead it is based on a flow analysis of the program.

The special $\langle \text{Place} \rangle \top$ is used to mark the types of values that are not accessible through any currently-bound variable. For example, the occurrence of \top in

$$\vdash \left\{ \left\{ \rho, \rho' \right\} / x : (\text{int}_{\text{B}}, \rho') \right\} \\ (x + 1_{\text{B}} \text{ at } \rho') \text{ at } \rho \llbracket \text{release } \rho' \rrbracket : (\text{int}_{\text{B}}, \rho) \\ \left\{ \left\{ \rho \right\} / x : (\text{int}_{\text{B}}, \top) \right\}$$

signals that, after evaluation of the expression, the value of x is no longer accessible. Attempting a subsequent operation involving access to the value of x would be rejected by the type system.

Intuitively, our region type system is very aggressive about deallocating regions: anything can be deallocated at any point.

This is possible if the context of an expression does not depend on the deallocated data—they are semantically dead. If the context requires data from the deallocated region then our type system gives a type error at the point of the data access. This deallocation strategy is different from the strategy in the Tofte/Talpin system that only allows deallocation of unobservable regions.

Region operations

$$\Psi \vdash \{ \Delta / \Gamma \} R \{ \Delta' / \Gamma' \}$$

We start by defining the effect of region operations on the set of available regions and on the region-annotated types of live values:

$$\frac{\rho \notin \Psi \uplus \Delta}{\Psi \vdash \{ \Delta / \Gamma \} \text{new } \rho \{ \Delta \uplus \{ \rho \} / \Gamma \}} \quad (1)$$

$$\frac{\rho \in \Psi \uplus \Delta \quad \rho' \notin \Psi \uplus \Delta}{\Psi \vdash \{ \Delta / \Gamma[\rho/\rho'] \} \rho' := \text{alias } \rho \{ \Delta \uplus \{ \rho' \} / \Gamma \}} \quad (2)$$

$$\frac{\rho \notin \text{frv}(\Gamma)}{\Psi \vdash \{ \Delta \uplus \{ \rho \} / \Gamma \} \text{release } \rho \{ \Delta / \Gamma \}} \quad (3)$$

$$\frac{\rho' \notin \Psi \uplus \Delta}{\Psi \vdash \{ \Delta \uplus \{ \rho \} / \Gamma \} \rho' := \rho \{ \Delta \uplus \{ \rho' \} / \Gamma[\rho'/\rho] \}} \quad (4)$$

Except for $\rho \notin \text{frv}(\Gamma)$ in the third rule, the side conditions on the rules correspond to the preconditions listed for each region operation in Section 3. The side condition $\rho \notin \text{frv}(\Gamma)$ in the third rule is meant to handle the following situation:

$$\text{let } x = \llbracket \text{new } \rho \rrbracket (1_{\text{B}} \text{ at } \rho) \llbracket \text{release } \rho \rrbracket \\ \text{in } \llbracket \text{new } \rho \rrbracket (x + 1_{\text{B}} \text{ at } \rho) \text{ at } \rho$$

This fragment should be rejected by the type system since the region created in the body of the **let** is actually a fresh region different

from the region used in the binding to x . And indeed the side condition rejects the fragment. To type $\llbracket \text{new } \rho \rrbracket (1_{\mathbb{B}} \text{ at } \rho) \llbracket \text{release } \rho \rrbracket$ one would have to use a subtyping step (below), which would give x the type $(\text{int}_{\mathbb{B}}, \top)$, making it inaccessible in the body of the **let** expression.

Expressions

$$\Psi \vdash \{\Delta / \Gamma\} e : \mu \{\Delta' / \Gamma'\}$$

Expressions with region operations

The rule for a prefixed region operation is natural:

$$\frac{\Psi \vdash \{\Delta_1 / \Gamma_1\} R \{\Delta_2 / \Gamma_2\} \quad \Psi \vdash \{\Delta_2 / \Gamma_2\} e : \mu \{\Delta_3 / \Gamma_3\}}{\Psi \vdash \{\Delta_1 / \Gamma_1\} \llbracket R \rrbracket e : \mu \{\Delta_3 / \Gamma_3\}} \quad (5)$$

but the corresponding attempt to handle a postfix:

$$\frac{\Psi \vdash \{\Delta_1 / \Gamma_1\} e : \mu \{\Delta_2 / \Gamma_2\} \quad \Psi \vdash \{\Delta_2 / \Gamma_2\} R \{\Delta_3 / \Gamma_3\}}{\Psi \vdash \{\Delta_1 / \Gamma_1\} e \llbracket R \rrbracket : \mu \{\Delta_3 / \Gamma_3\}} \quad (\text{WRONG})$$

does not work! Because R is executed *after* the evaluation of e it might affect regions variables that appear in μ . For example, the rule above would allow constructions such as

$$\pi_1 ((5, 7) \text{ at } \rho \llbracket \text{release } \rho \rrbracket \llbracket \text{new } \rho \rrbracket)$$

Instead of this rule we have to thread the μ through the typing of R so the latter can rewrite it appropriately. Similar cases also arise elsewhere in the system, so we invent a general solution: An environment Γ can contain *anonymous entries*, denoted by $*:\mu$, which hold the region-annotated types of intermediate results that are only needed during the evaluation of the current expression/operation. The intuition is that a Γ models the contents of the value stack if the expression were to be evaluated on a stack machine. Thus we write

$$\frac{\Psi \vdash \{\Delta_1 / \Gamma_1\} e : \mu_0 \{\Delta_2 / \Gamma_2\} \quad \Psi \vdash \{\Delta_2 / \Gamma_2, *:\mu_0\} R \{\Delta_3 / \Gamma_3, *:\mu\}}{\Psi \vdash \{\Delta_1 / \Gamma_1\} e \llbracket R \rrbracket : \mu \{\Delta_3 / \Gamma_3\}} \quad (6)$$

This explains the unusual definition of $\langle \text{VarEnv} \rangle$ in Figure 2 and the corresponding lookup operation.

Region subtyping

We define a partial order \sqsubseteq on places by

$$\perp \sqsubseteq \rho \sqsubseteq \top$$

and its natural pointwise extensions to μ 's (not allowing \sqsubseteq inside function types) and Γ 's. The subtyping rule

$$\frac{\Psi \vdash \{\Delta_1 / \Gamma'_1\} e : \mu_0 \{\Delta_2 / \Gamma_2\} \quad \Gamma_1 \sqsubseteq \Gamma'_1 \quad \Gamma_2, *:\mu_0 \sqsubseteq \Gamma'_2, *:\mu' \quad \text{frv}(\Gamma'_1) \subseteq \Psi \uplus \Delta_1 \quad \text{frv}(\Gamma'_2, *:\mu') \subseteq \Psi \uplus \Delta_2}{\Psi \vdash \{\Delta_1 / \Gamma_1\} e : \mu' \{\Delta_2 / \Gamma_2\}} \quad (7)$$

now allows region variables inside region types to be changed to \top . The effect of this is that the new type does not allow the values it describes to be accessed. On the other hand, if all instances of a region variable are changed to \top the variable is not free in the environment anymore; this allows it to be released.

Similarly, a place can start out as \perp and be changed to a region variable at a later time when the desired region variable comes into existence. That allows constructions such as

$$\text{let } x = [] \text{ at } \rho \text{ in } (\text{life } [i; \dots; o; \rho'](\dots) :: x) \text{ at } \rho$$

where *life* creates the region that contains the list elements *after* the $[]$ has been constructed. The type of x can start out as $((\dots, \perp) \text{ list}, \rho)$ and just before the **cons** operation it can be reinterpreted as $((\dots, \rho') \text{ list}, \rho)$.

Variables and variable bindings

Variables and variable bindings are standard:

$$\frac{\Psi \vdash \{\Delta_1 / \Gamma_1\} e : \mu \{\Delta_2 / \Gamma_2\} \quad \Psi \vdash \{\Delta_2 / \Gamma_2, x:\mu\} e' : \mu' \{\Delta_3 / \Gamma_3, x:\mu''\}}{\Psi \vdash \{\Delta_1 / \Gamma_1\} \text{let } x = e \text{ in } e' : \mu' \{\Delta_3 / \Gamma_3\}} \quad (8)$$

$$\frac{}{\Psi \vdash \{\Delta / \Gamma\} x : \Gamma(x) \{\Delta / \Gamma\}} \quad (9)$$

where the environment lookup $\Gamma(x)$ ignores anonymous entries.

Operations on unboxed integers

With unboxed integers no memory management or regions are needed at all, and all we have to do is mechanically pass around the environments. Note that the two branches of a conditional must have the same net effects on the regions and region types

$$\frac{}{\Psi \vdash \{\Delta / \Gamma\} n : \text{int } \{\Delta / \Gamma\}} \quad (10)$$

$$\frac{\Psi \vdash \{\Delta_1 / \Gamma_1\} e : \text{int } \{\Delta_2 / \Gamma_2\} \quad \Psi \vdash \{\Delta_2 / \Gamma_2\} e' : \text{int } \{\Delta_3 / \Gamma_3\}}{\Psi \vdash \{\Delta_1 / \Gamma_1\} (e \otimes e') : \text{int } \{\Delta_3 / \Gamma_3\}} \quad (11)$$

$$\frac{\Psi \vdash \{\Delta_1 / \Gamma_1\} e : \text{int } \{\Delta_2 / \Gamma_2\} \quad \Psi \vdash \{\Delta_2 / \Gamma_2\} e' : \mu \{\Delta_3 / \Gamma_3\} \quad \Psi \vdash \{\Delta_2 / \Gamma_2\} e'' : \mu \{\Delta_3 / \Gamma_3\}}{\Psi \vdash \{\Delta_1 / \Gamma_1\} \text{if } e \neq 0 \text{ then } e' \text{ else } e'' : \mu \{\Delta_3 / \Gamma_3\}} \quad (12)$$

Operations on boxed integers

Boxed integers are our main example of a primitive data type that must be heap allocated. Each constructor expression specifies where its result should be placed, and the rules check that all involved regions are live.

$$\frac{\rho \in \Psi \uplus \Delta}{\Psi \vdash \{\Delta / \Gamma\} n_{\mathbb{B}} \text{ at } \rho : (\text{int}_{\mathbb{B}}, \rho) \{\Delta / \Gamma\}} \quad (13)$$

$$\frac{\Psi \vdash \{\Delta_1 / \Gamma_1\} e : \mu_0 \{\Delta_2 / \Gamma_2\} \quad \Psi \vdash \{\Delta_2 / \Gamma_2, *:\mu_0\} e' : (\text{int}_{\mathbb{B}}, \rho') \{\Delta_3 / \Gamma_3, *:(\text{int}_{\mathbb{B}}, \rho)\} \quad \rho, \rho', \rho'' \in \Psi \uplus \Delta_3}{\Psi \vdash \{\Delta_1 / \Gamma_1\} (e \otimes e') \text{ at } \rho'' : (\text{int}_{\mathbb{B}}, \rho'') \{\Delta_3 / \Gamma_3\}} \quad (14)$$

In the latter rule, the side condition $\rho, \rho' \in \Psi \uplus \Delta_3$ is, strictly speaking, redundant, because the typing rules maintain the invariant that $\text{frv}(\Gamma) \subseteq \Psi \cup \Delta$ (or, respectively, $\text{frv}(\Gamma', *:\mu) \subseteq \Psi \cup \Delta'$). We choose to leave them in for readability. The condition $\rho'' \in \Psi \uplus \Delta_3$ is essential.

Pairs

The rules for pairs are simple. Observe that none of the rules check that the pair's *components* are readable. It does no harm to put a dangling pointer into a pair and later extract it. What matters is that the pointer does not dangle if we try to read through it; this is

independent of whether it has been stored in a pair in the meantime.

$$\frac{\Psi \vdash \{\Delta_1/\Gamma_1\} e : \mu_0 \{\Delta_2/\Gamma_2\} \quad \Psi \vdash \{\Delta_2/\Gamma_2, *;\mu_0\} e' : \mu' \{\Delta_3/\Gamma_3, *;\mu\} \quad \rho \in \Psi \uplus \Delta_3}{\Psi \vdash \{\Delta_1/\Gamma_1\} (e, e') \text{ at } \rho : (\mu \times \mu', \rho) \{\Delta_3/\Gamma_3\}} \quad (15)$$

$$\frac{\Psi \vdash \{\Delta_1/\Gamma_1\} e : (\mu_1 \times \mu_2, \rho) \{\Delta_2/\Gamma_2\} \quad \rho \in \Psi \uplus \Delta_2}{\Psi \vdash \{\Delta_1/\Gamma_1\} \pi_i e : \mu_i \{\Delta_2/\Gamma_2\}} \quad (16)$$

Lists

Lists are the prototypical example of a recursive data type. The rules contain no surprises for readers who are familiar with the handling of lists in other region systems. This does not mean, however, that lists are trivial. In a sense, recursive types such as lists are only source of imprecision in our system: A program that does not use lists can always be annotated such that at runtime, each region is used for exactly one allocation⁴.

$$\frac{\{\rho\} \cup \text{frv}(\mu) \subseteq \Psi \uplus \Delta}{\Psi \vdash \{\Delta/\Gamma\} [] \text{ at } \rho : (\mu \text{ list}, \rho) \{\Delta/\Gamma\}} \quad (17)$$

$$\frac{\Psi \vdash \{\Delta_1/\Gamma_1\} e : \mu_0 \{\Delta_2/\Gamma_2\} \quad \Psi \vdash \{\Delta_2/\Gamma_2, *;\mu_0\} e' : (\mu \text{ list}, \rho) \{\Delta_3/\Gamma_3, *;\mu\} \quad \rho \in \Psi \uplus \Delta_3}{\Psi \vdash \{\Delta_1/\Gamma_1\} e :: e' \text{ at } \rho : (\mu \text{ list}, \rho) \{\Delta_3/\Gamma_3\}} \quad (18)$$

$$\frac{\Psi \vdash \{\Delta_1/\Gamma_1\} e : (\mu \text{ list}, \rho) \{\Delta_2/\Gamma_2\} \quad \Psi \vdash \{\Delta_2/\Gamma_2\} e' : \mu_3 \{\Delta_3/\Gamma_3\} \quad \Psi \vdash \{\Delta_2/\Gamma_2, x;\mu, x'\} (\mu \text{ list}, \rho) e'' : \mu_3 \{\Delta_3/\Gamma_3, x;\mu', x';\mu''\} \quad \rho \in \Psi \uplus \Delta_2}{\Psi \vdash \{\Delta_1/\Gamma_1\} \text{ case } e \text{ of } [] \Rightarrow e' \text{ or } x :: x' \Rightarrow e'' : \mu_3 \{\Delta_3/\Gamma_3\}} \quad (19)$$

Function calls

Our model for function call is illustrated by our syntax for function types:

$$\forall \vec{\rho}. (\exists \vec{\rho}' . \mu') \multimap (\exists \vec{\rho}'' . \mu'')$$

where $\vec{\rho}$ are the constant region parameters, $\vec{\rho}'$ are the input region parameters, μ' is the argument type, $\vec{\rho}''$ are the output region parameters, and μ'' is the result type. We do not allow function types to have free region variables, so there are implicit well-formedness conditions

$$\text{frv}(\mu') \subseteq \{\{\rho'_i\}_i\} \cup \{\{\rho_i\}_i\} \quad \text{and} \quad \text{frv}(\mu'') \subseteq \{\{\rho''_i\}_i\} \cup \{\{\rho_i\}_i\}$$

The intuition behind the existential quantifiers comes from dependent types: A region-annotated type can be viewed as depending on the region variables in it, and so the input (or output) to the function consists of an existential pair with some regions and a value whose type depends on the regions.

The peculiar function arrow “ \multimap ” comes from linear logic and supports the intuition that the input regions and argument value disappear from the caller’s context in the call; that is, unless the caller explicitly takes steps to save copies of them.

Finally, the universal quantification of the constant region parameters is borrowed from the TT type system. The constant regions may be present in the argument type as well as in the result type.

⁴This property does not hold for the TT system. It depends on being able to use region renamings to reconcile the region typing of **then** and **else** branches.

The entire notation supports a type-theoretic intuition about who selects the bindings of which region variables: The ρ_i ’s are selected by the caller because they are universally quantified. The ρ'_i ’s are selected by the called function because they are existentially quantified. The ρ''_i ’s are also existentially quantified, but the quantifier is in a contravariant position, so they are selected by the caller.

The purpose of having constant region parameters in our system is to allow the caller to control the bindings of some of the region variables in the result type. Without constant region parameters we would not be able to type calls such as

$$\text{let } x = 17_{\mathbf{B}} :: [] \text{ in } (f \ 0) :: x$$

The list construction requires that the return value from f must live in the same region as the previously-constructed x . If we did not have constant region parameters, f would need to have a type like

$$(\exists . \text{int}) \multimap (\exists \rho. (\text{int}_{\mathbf{B}}, \rho))$$

which would not give the caller any guarantee that the region produced by the function were identical to the region where x is allocated. Even if we passed ρ_x into the function

$$(\exists \rho_x. \text{int}) \multimap (\exists \rho_x. (\text{int}_{\mathbf{B}}, \rho_x))$$

it would be allowed to release the ρ_x input and create a new region to use as output, so we could not even be sure that x ’s value still existed after the call. (To prevent this situation, the typing rule below explicitly forbids any of the actual input regions to occur in the caller’s type environment).

With constant region parameters, however, we can give f the type

$$\forall \rho_x. (\exists . \text{int}) \multimap (\exists . (\text{int}_{\mathbf{B}}, \rho_x))$$

which allows the caller to specify where the results should be allocated.

The rule for function calls is the most complex rule in our system, and we therefore introduce it with an example. Consider the function f defined by

$$\text{fun } f \text{ [i: } \rho_i; \mathbf{o}: \rho_o] x = e_f$$

where f expects a boxed integer in ρ_i and returns a boxed integer in ρ_o (f may release ρ_i and create ρ_o , we do not know). We capture this with the function type:

$$f : \forall . (\exists \rho_i. (\text{int}_{\mathbf{B}}, \rho_i)) \multimap (\exists \rho_o. (\text{int}_{\mathbf{B}}, \rho_o))$$

To type

$$f \text{ [i: } \rho'_i; \mathbf{o}: \rho''_i] (17_{\mathbf{B}} \text{ at } \rho')$$

we first type the subexpressions using the typing rules above and get

$$\frac{\Psi \vdash \{\Delta \uplus \rho' / \Gamma\} f : \sigma \{\Delta \uplus \rho' / \Gamma\} \quad \Psi \vdash \{\Delta \uplus \rho' / \Gamma\} 17_{\mathbf{B}} \text{ at } \rho' : (\text{int}_{\mathbf{B}}, \rho') \{\Delta \uplus \rho' / \Gamma\}}{\Psi \vdash \{\Delta \uplus \rho' / \Gamma\} f \text{ [i: } \rho'_i; \mathbf{o}: \rho''_i] (17_{\mathbf{B}} \text{ at } \rho') : (\text{int}_{\mathbf{B}}, \rho'') \{\Delta \uplus \rho'' / \Gamma\}}$$

We next have to match the actual region parameters ρ' and ρ'' with the formal parameters ρ_i and ρ_o , resulting in substitutions θ' and θ'' . Then we match argument and result: $\theta'(\text{int}_{\mathbf{B}}, \rho')$ and $\theta''(\text{int}_{\mathbf{B}}, \rho'')$ with $(\text{int}_{\mathbf{B}}, \rho_i)$ and $(\text{int}_{\mathbf{B}}, \rho_o)$. The actual input parameter ρ' loses its binding in the call, so we remove it from $\Delta \uplus \rho'$; conversely ρ'' gets bound, and we add that to get the final Δ . We have then derived

$$\Psi \vdash \{\Delta \uplus \rho' / \Gamma\} f \text{ [i: } \rho'_i; \mathbf{o}: \rho''_i] (17_{\mathbf{B}} \text{ at } \rho') : (\text{int}_{\mathbf{B}}, \rho'') \{\Delta \uplus \rho'' / \Gamma\}$$

In the general case with multiple region parameters, constant regions, and potential region operations in the operand we get the following rule.

$$\frac{\Psi \vdash \{\Delta_1/\Gamma_1\} e : \mu_0 \{\Delta_2/\Gamma_2\} \quad \Psi \vdash \{\Delta_2/\Gamma_2, *;\mu_0\} e' : \mu' \{\Delta_3/\Gamma_3, *;\sigma\} \quad \Gamma_3; \Psi \vdash \{\Delta_3/\mu'\} \sigma[\xi] \{\Delta_4/\mu''\}}{\Psi \vdash \{\Delta_1/\Gamma_1\} e[\xi] e' : \mu'' \{\Delta_4/\Gamma_3\}} \quad (20)$$

where we have used an auxiliary relation, described below, to handle the matching of actual parameters to formal parameters.

Parameter matching

$$\boxed{\Gamma; \Psi \vdash \{\Delta'/\mu'\} \sigma[\xi] \{\Delta''/\mu''\}}$$

The first rule below does the actual matching of parameters via substitutions θ , θ' , and θ'' . The remaining rules allow the manipulation of the current sets of region variables to match the shape of the function type. The third rule ensures that the bindings in the type environment does not contain references to the regions passed to the function.

$$\frac{\sigma = \forall \bar{\rho}. (\exists \bar{\rho}', \mu') \multimap (\exists \bar{\rho}'', \mu'') \quad [\xi] = [\mathbf{c}; \theta(\bar{\rho}); \mathbf{i}; \theta'(\bar{\rho}'); \mathbf{o}; \theta''(\bar{\rho}'')] \quad \Psi = \theta(\{\rho_i\}_i) \quad \Delta' = \theta'(\{\rho'_i\}_i) \quad \Delta'' = \theta''(\{\rho''_i\}_i) \quad \mu_i = (\theta * \theta')(\mu') \quad \mu_o = (\theta * \theta'')(\mu'')}{0; \Psi \vdash \{\Delta'/\mu_i\} \sigma[\xi] \{\Delta''/\mu_o\}} \quad (20a)$$

$$\frac{0; \Psi \vdash \{\Delta'/\mu'\} \sigma[\xi] \{\Delta''/\mu''\}}{0; \Psi \uplus \{\rho\} \vdash \{\Delta'/\mu'\} \sigma[\xi] \{\Delta''/\mu''\}} \quad (20b)$$

$$\frac{0; \Psi \vdash \{\Delta'/\mu'\} \sigma[\xi] \{\Delta''/\mu''\} \quad \text{frv}(\Gamma) \subseteq \Psi}{\Gamma; \Psi \vdash \{\Delta'/\mu'\} \sigma[\xi] \{\Delta''/\mu''\}} \quad (20c)$$

$$\frac{\Gamma; \Psi \uplus \{\rho\} \vdash \{\Delta'/\mu'\} \sigma[\xi] \{\Delta''/\mu''\}}{\Gamma; \Psi \uplus \{\rho\} \vdash \{\Delta' \uplus \{\rho\} / \mu'\} \sigma[\xi] \{\Delta'' \uplus \{\rho\} / \mu''\}} \quad (20d)$$

Function definitions

$$\boxed{\Gamma \vdash F \Rightarrow x : \mu}$$

Typing a function definition is simply a matter of wrapping up the typing judgment for the body as a function type:

$$\frac{\{\rho_i\}_i \vdash \{\{\rho'_i\}_i / \Gamma, x; \mu'\} e : \mu'' \{\{\rho''_i\}_i / \Gamma'', x; \mu'''\} \quad \forall i, j : \rho_i \neq \rho'_j}{\Gamma \vdash \mathbf{fun} f[\mathbf{c}; \bar{\rho}; \mathbf{i}; \bar{\rho}'; \mathbf{o}; \bar{\rho}''] x = e \Rightarrow f : \forall \bar{\rho}. (\exists \bar{\rho}', \mu') \multimap (\exists \bar{\rho}'', \mu'')} \quad (21)$$

Declaration sequences

$$\boxed{\Gamma \vdash d \Rightarrow \Gamma}$$

Typing declarations is simply a matter of bookkeeping:

$$\frac{}{\Gamma \vdash \Rightarrow 0} \quad (22)$$

$$\frac{\Gamma \vdash d \Rightarrow \Gamma' \quad \Gamma \vdash F \Rightarrow x : \mu}{\Gamma \vdash d F \Rightarrow \Gamma', x; \mu} \quad (23)$$

Programs

$$\boxed{\vdash P}$$

The rule for programs sets up some simple boundary conditions:

$$\frac{\Gamma \vdash d \Rightarrow \Gamma \quad \emptyset \vdash \{\emptyset/\Gamma\} e : \mu \{\emptyset/\Gamma'\}}{\vdash \mathbf{let} d \mathbf{in} e} \quad (24)$$

Note that we require that all regions be deallocated when the program is finished.

6. NON-OPTIMALITY OF REGION ANNOTATIONS

We would like now to present a theorem

Theorem 1 (false) *For a given Fun program, there is a set of region annotations that is optimal in the sense that no other well-typed annotations for that program will deallocate any value sooner than the optimal one.*

and give an algorithm to find optimal region annotations. Unfortunately the claim cannot be true, as witnessed by the following program:

```
let fun f n = let p = (5B, 7B) in if n = 0 then p else f (n - 1)
fun g n = let (x, y) = f n ; z = x :: y :: [] in (use z)
in let x = π1 (f 42) in (space-critical section that uses x)
```

The function g is never called but must still be typed. The list construction means that x and y must have the same region annotations. Thus g must be able to call f in such a way that the components of the pair that f returns are guaranteed to be in the same region. Ignoring (for brevity) the region where the pair containing 5_B and 7_B is allocated, there are two fundamentally different ways to region-annotate f which allow this:

- a) $f : \forall \rho_1, \rho_2. (\exists . \text{int}) \multimap (\exists . (\text{int}_B, \rho_1) \times (\text{int}_B, \rho_2))$
 $\mathbf{fun} f[\mathbf{c}; \rho_1, \rho_2] n = \mathbf{let} p = (5_B \mathbf{at} \rho_1, 7_B \mathbf{at} \rho_2)$
 $\mathbf{in} \mathbf{if} n = 0 \mathbf{then} p$
 $\mathbf{else} f[\mathbf{c}; \rho_1, \rho_2] (n - 1)$
- b) $f : \forall . (\exists . \text{int}) \multimap (\exists \rho_3. (\text{int}_B, \rho_3) \times (\text{int}_B, \rho_3))$
 $\mathbf{fun} f[\mathbf{o}; \rho_3] n = \mathbf{let} p = [\mathbf{new} \rho_3] (5_B \mathbf{at} \rho_3, 7_B \mathbf{at} \rho_3)$
 $\mathbf{in} \mathbf{if} n = 0 \mathbf{then} p$
 $\mathbf{else} [\mathbf{release} \rho_3] f[\mathbf{o}; \rho_3] (n - 1)$

Each of these have different advantages. Version (a) contains a space leak in f itself if n is large, because the 5_B 's and 7_B 's allocated by each iteration are never deallocated. In version (b), the region containing 5_B and 7_B is released before the recursive call.

However, from the main program's point of view, version (a) is desirable, because then the two components of the pair can be allocated in different regions, and the second one deallocated before the space-critical section. With version (b), the main program must keep both elements around until x can be safely deallocated.

The net result of this is that neither typing (a) nor (b) is clearly superior to the other.

One way to resolve this dilemma would be to replicate the definition of f such that call sites requiring different region separations of the return value call different implementations of f . Then the call from g could use typing (b) while the main program would call a different implementation of f with typing

- c) $f : \forall . (\exists . \text{int}) \multimap (\exists \rho_1, \rho_2. (\text{int}_B, \rho_1) \times (\text{int}_B, \rho_2))$

This strategy would be sound and terminating even in the presence of recursion, because there is only a finite number of fundamentally different region annotations of each return type. However, the number of different versions of each function might grow exponentially with the size of the return type, so the strategy is not necessarily optimal in a practical sense.

This could be viewed as a weakness of our system, but it turns out that the previous systems have similar problems.

The CC system has exactly the same problem as our system.

The TT system does not have the problem; it allows the two typing variants

a') $f : \forall \rho_1, \rho_2. \text{int} \rightarrow (\text{int}_B, \rho_1) \times (\text{int}_B, \rho_2)$

b') $f : \forall \rho_3. \text{int} \rightarrow (\text{int}_B, \rho_3) \times (\text{int}_B, \rho_3)$

but in the TT system (a') is never inferior to (b'), so (a') can safely be used.

The ML Kit system faces a problem similar to the one in our system. It builds upon the TT system, but with region resetting present there *are* situations where (b') is preferable to (a'). Namely, with (a') the call from g will alias the region parameters ρ_1 and ρ_2 , which means that f cannot be allowed to reset ρ_2 before the allocation of 7_B . Thus the 7_B 's will pile up in ρ_2 when f is called from the main program. On the other hand, using (b') gives problems in the main program as outlined previously.

The AFL system has a related problem. Consider

```
let fun f (x,y) = let z = x + y in <space-critical section>
  fun g x = f(x,x)
in let x = 5B ; y = 7B
  in if <something> then f(x,y)
     else <space-critical section that uses x>
```

Again two TT typings may be considered for f :

a'') $\forall \rho_1, \rho_2. (\text{int}_B, \rho_1) \times (\text{int}_B, \rho_2) \rightarrow \dots$

b'') $\forall \rho_3. (\text{int}_B, \rho_3) \times (\text{int}_B, \rho_3) \rightarrow \dots$

In typing (a'') the two region parameters ρ_1 and ρ_2 are aliased in the call from g . This means that f can be allowed to free only one of them early, lest it would try to free the same region twice. Thus in the call from the main program, either x or y must stay allocated during the space-critical section. Typing (b'') does not have that problem, but would on the other hand require the main program to allocate 5_B and 7_B in the same region, keeping both allocated during the space-critical section in the **else** branch.

This kind of trade-offs in the Kit and AFL systems has apparently not been reported in the literature before. Their existence questions the fundamental design approach of these systems, which is to build on a classical TT region inference. An unspoken assumption here is that region annotations produced by a region inference that aims at a pure TT execution model will also be “good” as a baseline for more advanced execution models. Our examples show that in some situations the memory behavior can be better if one uses a baseline that would be inferior in a pure TT setting.

More generally, these findings emphasize Tofte and Hallenberg’s conclusion [14] that region inference cannot be left completely to the compiler: A practical region-based language implementation must allow the programmer to influence the region annotations more directly than by rewriting the source program. On the other hand, the region annotations for even trivial program parts easily become so numerous that it is impractical to expect the programmer to write them all by hand. Some middle way must be found where the programmer can nudge a mostly-automatic region inference in the right direction at critical points and get the boring details tended to in the rest of the program. How such manual annotations are best made is still an open question.

7. IMPLEMENTATION TECHNIQUES

We have produced a prototype Fun/RegFun system, available at <http://www.diku.dk/~hmiss/rbmm/regfun.tar.gz>. Below we discuss selected aspects of our implementation as well as implementation issues that we feel need more work.

7.1 Region inference

Region inference for the TT system traditionally works by first spreading fresh region variables over a standard typing for the program, and then unifying region variables until the region type system’s demands are satisfied. This principle does not extend well to our system because the region annotations of a variable’s type can change during its lifetime.

We therefore propose another strategy for producing region annotations for our system. Our basic method can be classified as *backwards abstract interpretation*. The method is not defined formally except as code in our prototype implementation (we know from the previous section that its theoretical properties cannot be impressive), but we give the flavor of the analysis by means of an example. We consider the analysis of the expression:

if $x > 0_B$ **then** $x + 7_B$ **else** x

Imagine that the inference has already discovered that when the expression has been evaluated the result should be in a region called ρ_1 and x is not used in the continuation. We thus have a partial typing judgement

$$\vdash \{?/?\} \text{ if...then...else...} : (\text{int}_B, \rho_1) \{ \{\rho_1\} / x : (\text{int}_B, \top) \}$$

In the rest of the example we will use a simplified notation where “ (int_B, p) ” is abbreviated to just “ p ” and the Δ parts of the judgement are omitted. During the inference, Δ is always implicitly equal to $\text{frv}(\Gamma) \setminus \Psi$ (and Ψ is empty in this example). The abbreviated boundary condition is thus

$$\vdash \{?\} \text{ if...then...else...} : \rho_1 \{x : \top\}$$

The analysis of the conditional begins by analyzing the **else** branch with the same end state. This results in

$$\vdash \{x : \rho_1\} x : \rho_1 \{x : \top\}$$

The **then** branch is also analyzed with this end state. The result of the addition must be delivered in ρ_1 and because ρ_1 is not mentioned elsewhere in the environment, it becomes the job of the “+” expression to create it. The operands must also be read from some regions. The weakest possible precondition is to assume that each operand is in its own region that can be released after the addition. So we select two fresh region variables ρ_2 and ρ_3 for them. We then insert appropriate region operations to get

$$(\dots + \dots \text{ [new } \rho_1]) \text{ at } \rho_1 \text{ [release } \rho_2] \text{ [release } \rho_3]$$

and proceed to analyze the second operand with boundary condition

$$\vdash \{?\} 7_B : \rho_3 \{x : \top, * : \rho_2\}$$

The analysis of the constant 7_B yields

$$\vdash \{x : \top, * : \rho_2\} \text{ [new } \rho_3] 7_B \text{ at } \rho_3 : \rho_3 \{x : \top, * : \rho_2\}$$

after which the first operand can be analyzed to get

$$\vdash \{x : \rho_2\} x : \rho_2 \{x : \top\}$$

We can now synthesize an entire annotated addition:

$$\vdash \{x : \rho_2\} (x + \text{ [new } \rho_3] 7_B \text{ at } \rho_3 \text{ [new } \rho_1]) \text{ at } \rho_1 : \rho_1 \{x : \top\}$$

$$\text{ [release } \rho_2] \text{ [release } \rho_3]$$

Now that both branches of the conditional have been analyzed, we need to unify their pre-environments $x : \rho_2$ and $x : \rho_1$. In this case we can let $x : \rho_1$ be the least upper bound and insert a $[\rho_2 := \rho_1]$ at the beginning of the **then** branch to fit it to the new precondition. (If one of the environments had required regions to be equal that

were different in the other, we would also have needed to insert **alias** operations).

The conditional is now partially analyzed as

$$\begin{aligned} &\mathbf{if} \dots \mathbf{then} \llbracket \rho_2 := \rho_1 \rrbracket (x + \llbracket \mathbf{new} \rho_3 \rrbracket 7_B \mathbf{at} \rho_3 \llbracket \mathbf{new} \rho_1 \rrbracket) \mathbf{at} \rho_1 \\ &\quad \llbracket \mathbf{release} \rho_2 \rrbracket \llbracket \mathbf{release} \rho_3 \rrbracket \\ &\mathbf{else} \ x \end{aligned}$$

and we need to analyze the condition in the context

$$\vdash \{?\} x > 0_B : \text{int } \{x:\rho_2\}$$

The analysis of $>$ and 0_B mimics that of $+$ and 7_B , but we then have to analyze

$$\vdash \{?\} x : \rho_4 \{x:\rho_1\}$$

Here we need ρ_4 and ρ_1 to be explicit aliases. We get

$$\vdash \{x:\rho_1\} x \llbracket \rho_4 := \mathbf{alias} \rho_1 \rrbracket : \rho_4 \{x:\rho_1\}$$

The entire annotated conditional becomes

$$\begin{aligned} \vdash \{x:\rho_1\} &\mathbf{if} (x \llbracket \rho_4 := \mathbf{alias} \rho_1 \rrbracket > \llbracket \mathbf{new} \rho_5 \rrbracket 0_B \mathbf{at} \rho_5) \\ &\quad \llbracket \mathbf{release} \rho_4 \rrbracket \llbracket \mathbf{release} \rho_5 \rrbracket \\ &\mathbf{then} \llbracket \rho_2 := \rho_1 \rrbracket \\ &\quad (x + \llbracket \mathbf{new} \rho_3 \rrbracket 7_B \mathbf{at} \rho_3 \llbracket \mathbf{new} \rho_1 \rrbracket) \mathbf{at} \rho_1 \\ &\quad \llbracket \mathbf{release} \rho_2 \rrbracket \llbracket \mathbf{release} \rho_3 \rrbracket \\ &\mathbf{else} \ x && : \rho_1 \{x:\top\} \end{aligned}$$

Though we showed in Section 6 that optimal region annotations for entire programs are not always possible, we conjecture that the backward analysis we have described here does produce optimal annotations if the region-annotated types of all the program’s functions are fixed in advance.

This assumption does, of course, not hold in our prototype implementation, where we use a global fixpoint iteration to decide on the region-annotated types of function. That algorithm makes no attempt to choose intelligently between the various choices in examples such as the one we gave in Section 6.

7.2 Region optimizations

Our experience with the prototype region inference is that it generally results in “fairly good” object lifetimes, even though we know it to not always be optimal.

However, it makes far from good use of the region-management operations themselves. For example, the ρ_4 alias above was clearly superfluous. We extend the basic region inference with a post-optimization pass doing the equivalent of copy propagation for the region sublanguage, which converts the annotated example expression to

$$\begin{aligned} &\mathbf{if} (x > \llbracket \mathbf{new} \rho_2 \rrbracket 0_B \mathbf{at} \rho_2) \llbracket \mathbf{release} \rho_2 \rrbracket \\ &\mathbf{then} (x + \llbracket \mathbf{new} \rho_3 \rrbracket 7_B \mathbf{at} \rho_3 \llbracket \mathbf{new} \rho_4 \rrbracket) \mathbf{at} \rho_4 \\ &\quad \llbracket \mathbf{release} \rho_1 \rrbracket \llbracket \mathbf{release} \rho_3 \rrbracket \llbracket \rho_1 := \rho_4 \rrbracket \\ &\mathbf{else} \ x \end{aligned}$$

One can also imagine other, more advanced, optimizations that result in more efficient use of the region operations. Examples include:

- a global variant of the just mentioned copy propagation, merging region parameters that are aliased at all call sites;
- an analysis that tries to replace region creations with **alias** operations if another region can be shown to be eventually deallocated at about the same time as the one to be created;
- conversion of region inputs to constant regions parameters whenever it can be shown that an input region cannot be

deallocated by the function—either because it is always reused as an output region, or because it is always kept alive by aliasing in the caller;

- elimination of region outputs when the region actually output is always created as an alias for a constant region parameter;
- elimination of constant region parameters that are not mentioned in the function body. Those parameters may be necessary for typing the function (because they appear in the region-annotated types of values that are read) but are not operationally needed by it (in the standard implementation model described below). Removing them will not preserve the RegFun program’s typeability, but it does preserve memory safety.

A particularly nice feature of region-based memory management is that such optimizations generally depend on only the imperative region sublanguage, and not on data-manipulation constructions except region-annotated allocation points. For example, an alias analysis for region variables should be vastly simpler than an alias analysis for data, because no structured values are involved.

7.3 Implementing the region operations

The intended implementation model for RegFun is the same as the one used in the ML Kit: Regions are dynamically extensible linked lists of fixed-size pages that can be deallocated in constant time. Adding reference counts to the implementation is trivial.

In this implementation model, a data pointer is an *absolute* addresses (though, for technical reasons, a formal semantics would modeled it as a region–offset pair). An interesting alternative would be to bind variables to relative addresses (offsets) only, without the region address they belong to. Both read and write accesses would require the corresponding region variable, which can be thought of as base registers for addressing.

It is not always necessary or desirable to use completely general implementations of each region operation. As a simple example, a data-flow analysis on the imperative region sublanguage may reveal that a particular **release** operation will always decrease the reference count to 0, in which case it can be implemented as an unconditional deallocation.

More importantly, experience with the ML Kit has shown that it often improves both time and memory behavior to use a special implementation model for “finite” regions where a static bound on the amount of allocated data can be inferred. We believe that the *multiplicity inference* which identifies the finite regions [2] would be relatively easy to adapt to our system, but it is not yet clear what the special treatment of finite regions should be. In the ML Kit a finite region is simply stack-allocated on the execution stack, but that solution can not always be used in our system where we have deliberately broken the ties between region lifetime and the execution stack. The same problem results from trying to use multiplicity inference with the AFL system, but the topic is not discussed in [1].

8. CONCLUSION AND FURTHER WORK

We have developed a region type system for reasoning about and managing region-based heap memory for direct-style functional programs. Our system is based on a simple imperative sublanguage of region operations. Primitive region operations consist of assigning a newly allocated region to a region variable, aliasing region variables (assigning a region from one region variable to another), renaming a region variable, releasing (unassigning) a

region variable and finally, all sequential compositions. A Floyd-Hoare style type system keeps track of the assigned region variables and the data in the heap that are accessible through them.

Our contributions consist of the use of reference counted regions, introduction of explicit region aliasing and renaming commands, and the design of a region type system that tracks assigned region variables in a control-flow sensitive fashion. The region type system constitutes a relatively simple, yet very expressive logic that lets us capture the original region management systems of Tofte and Talpin as well as its refinements, the ML Kit system with storage mode analysis and the allocation/deallocation analysis of Aiken, Fähndrich and Levien. Disregarding reference counting, it appears to be comparable in expressive power to the Walker, Crary and Morrisett's Calculus of Capabilities for continuation-passing style programs.

In practice this means that memory behavior can be analyzed precisely and directly at the source code level in our region type system in cases that either required rewriting the source code or additional analyses or both in the previous systems:

- Tail recursive functions remain tail recursive since regions allocated in the body of a recursive function can be deallocated inside its tail call(s).
- The region in which results are returned can be determined dynamically by executing explicit region aliasing or assignment commands under dynamic control; that is, the actual branch taken at runtime determines which region is bound to an output region variable.
- Reference counting simplifies the inference system and lets us deallocate regions earlier than TT, AFL or CC. If a region is unaliased at runtime (has reference count 1) it is deallocated by a **release** command, whereas it is not if another region variable still holds a handle to it. This means different call contexts of a function may result in different region deallocation behavior.

We have not provided a soundness proof for our type system here. A companion technical report [6] contains the formal developments and a proof of region safety. We are working on extending the region type system to cover lexical closures (fully higher-order functions) and strengthen it by advanced region subtyping. We are considering region inference where copy operations are inserted automatically. Practical use and experiments require extension of our prototype implementations to support larger subset of SML, support for recursive data types and other common language facilities, most of which do not require fundamentally new insights. As mentioned earlier competitive implementations require region management that is specialized to small sized regions and regions of static size. For this we need to carefully track region sizes, aliasing relations between region variables and, eventually, between individual references.

Eventually we wish to support real-time programming with dynamic memory management, analogous to the work of Hughes and Pareto [7] on sized types, which is based on a TT-style region type system.

9. REFERENCES

- [1] A. Aiken, M. Fähndrich, and R. Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Programming Language Design and Implementation (PLDI)*, volume 30(6) of *SIGPLAN Notices*, pages 174–185, 1995.
- [2] L. Birkedal, M. Tofte, and M. Vejstrup. From region inference to von Neumann machines via region representation inference. In *Principles of Programming Languages (POPL)*, pages 171–183, 1996.
- [3] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Principles of Programming Languages (POPL)*, pages 262–275, 1999.
- [4] D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *Lisp and Functional Programming (LFP)*, pages 28–38, 1986.
- [5] D. R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software—Practice and Experience*, 20(1):5–12, 1990.
- [6] F. Henglein, H. Makhholm, and H. Niss. A direct approach to control-flow sensitive region-based memory management. Technical report, 2001. (<http://www.diku.dk/~hniss/rbmm/ppdp-ext.ps.gz>).
- [7] J. Hughes and L. Pareto. Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. In *International Conference on Functional Programming (ICFP)*, volume 34(9) of *SIGPLAN Notices*, pages 70–81, 1999.
- [8] D. T. Ross. The AED free storage package. *Commun. ACM*, 10(8):481–492, 1967.
- [9] J. T. Schwartz. Optimization of very high level languages (parts I and II). *Comput. Lang.*, 1(2 & 3):161–194, 197–218, 1975.
- [10] M. Semmelroth and A. Sabry. Monadic encapsulation in ML. In *International Conference on Functional Programming (ICFP)*, volume 34(9) of *SIGPLAN Notices*, pages 8–17, 1999.
- [11] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.
- [12] J.-P. Talpin and P. Jouvelot. Polymorphic type, region, and effect inference. *J. Func. Prog.*, 2(3):245–271, 1992.
- [13] M. Tofte, L. Birkedal, M. Elsmann, N. Hallenberg, T. H. Olesen, P. Sestoft, and P. Bertelsen. Programming with regions in the ML Kit (for version 3). Technical Report DIKU-TR-98/25, Department of Computer Science, University of Copenhagen (DIKU), 1998.
- [14] M. Tofte and N. Hallenberg. Region-based memory management in perspective. In F. Henglein, J. Hughes, H. Makhholm, and H. Niss, editors, *Workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE)*, pages 23–30, 2001.
- [15] M. Tofte and J.-P. Talpin. Implementing the call-by-value lambda-calculus using a stack of regions. In *Principles of Programming Languages (POPL)*, pages 188–201, 1994.
- [16] M. Tofte and J.-P. Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, 1997.
- [17] P. Wadler. The marriage of effects and monads. In *International Conference on Functional Programming (ICFP)*, volume 34(1) of *SIGPLAN Notices*, pages 63–74, 1998.
- [18] D. Walker, K. Crary, and G. Morrisett. Typed memory management via static capabilities. *ACM Trans. Prog. Lang. Syst.*, 22(4):701–771, 2000.