Chapter 15

A Simple Region Inference Algorithm for a First-Order Functional Language

Manuel Montenegro, Ricardo Peña, Clara Segura [1]
Category: Research

Abstract: *Safe* is a first-order eager language with facilities for programmer controlled destruction and copying of data structures. It provides also *regions*, i.e. disjoint parts of the heap, where the program allocates data structures. The runtime system does not need a garbage collector and all allocation/deallocation actions are done in constant time. The language is aimed at inferring and certifying upper bounds for memory consumption in a Proof Carrying Code environment. Some of its analyses have been presented elsewhere [4, 6, 7]. In this paper we present a simple region inference algorithm for annotating programs with regions arguments and region types. Programmers are assumed to write programs and to declare datatypes without any reference to regions. The algorithm decides the regions that will be needed by every function and type. We show convincing examples of programs before and after region annotation, prove the correctness and optimality of the inference algorithm and give its asymptotic cost.

15.1 INTRODUCTION

Many imperative languages offer low level mechanisms to allocate and free heap memory, which the programmer may use in order to dynamically create and destroy pointer based data structures. These mechanisms give the programmer complete control over memory usage but are very error prone. Well known problems

[1] Dpto. Sistemas Informáticos y Computación, Univ. Complutense de Madrid, Spain; E-mail: montenegro@fdi.ucm.es, {ricardo,csegura}@sip.ucm.es.
Work supported by the MEC FPU grant AP2006-02154 and by the Madrid Region Government grant S-0505/TIC/0407 (PROMESAS).

that may arise are dangling references, undesired sharing between data structures with complex side effects as a consequence, and polluting memory with garbage.

On the other hand, functional languages usually consider memory management as a low level issue. Allocation is done implicitly and usually a garbage collector takes care of the memory exhaustion situation.

A semi-explicit approach to memory control is our functional language *Safe*, in which the programmer cooperates with the memory management system by providing some information about the intended use of data structures. For instance, the programmer may indicate that some particular data structure will not be needed in the future and that, as a consequence, it may be safely destroyed by the runtime system and its memory recovered. The language uses regions to locate data structures. It also allows controlling the degree of sharing between different data structures. A garbage collector is not needed.

More interesting is the definition of a type system [6, 7] guaranteeing that destruction facilities can be used in a safe way. In particular, it guarantees that dangling pointers are never created in the live heap. In this paper we present a simple region inference algorithm for annotating programs with regions arguments and region types. Programmers are assumed to write programs and to declare datatypes without any reference to regions. The algorithm decides the regions that will be needed by every function and type.

In Section 15.2 we summarize the language *Safe* and in Section 15.3 we present its big-step operational semantics. In Section 15.4 the region inference algorithm is presented. Section 15.5 sketches a correctness proof of the part of *Safe*'s type system regarding regions. Section 15.6 shows some examples whose regions have been inferred. Finally, Section 15.7 compares this work with other languages with memory management facilities.

15.2 LANGUAGE CONCEPTS AND INFERENCE EXAMPLES

Safe was designed in such a way that the compiler has a complete control on where and when memory allocation and deallocation actions will take place at runtime. The smallest memory unit is the **cell**, a contiguous memory space big enough to hold any data construction. A cell contains the mark of the constructor and a representation of the free variables to which the constructor is applied. These may consist, either of basic values, or of pointers to other constructions. It is allocated at constructor application time and can be deallocated by *destructive pattern matching*, a facility included in Safe. Allocation and deallocation of a single cell needs constant time. A **region** is a collection of cells. It is created empty and it may grow and shrink while it is active. Region deallocation frees all its cells and it is done in constant time.

A **data structure** (DS) is the set of cells obtained by starting at one cell considered as the root, and taking the transitive closure of the relation $C_1 \rightarrow C_2$, where C_1 and C_2 are cells of the same type T, and in C_1 there is a pointer to C_2. That means that, for instance in a list of type `[[a]]`, we consider as a DS all the cells belonging to the outermost list, but not those belonging to the individual in-

nermost lists. Each one of the latter constitute a separate DS. A DS completely resides in one region. A DS can be part of another DS, or two DSs can share a third one. The basic values —integers, booleans, etc.— do not allocate cells in regions. They live inside the cells of DSs, or in the stack.

These decisions are reflected in the way the type system deals with datatype definitions. Polymorphic algebraic data types are defined through **data** declarations as the following one:

```
data Tree a = Empty | Node (Tree a) a (Tree a)
```

The types assigned by the compiler to constructors include an additional argument indicating the region where the constructed values of that type are allocated. In the example, the compiler infers:

$$\textbf{data}\ \mathit{Tree}\ a\ @\rho = \mathit{Empty}@\rho \mid \mathit{Node}\ (\mathit{Tree}\ a\ @\rho)\ a\ (\mathit{Tree}\ a\ @\rho)@\rho$$

where ρ is the type of the region argument given to the constructors. After region inference, constructions appear in the annotated text with an additional argument `r` that will be bound at runtime to an actual region, as in `Node lt x rt @ r`. Constructors are polymorphic in region arguments, meaning that they can be applied to any actual region. But, due to the above type restrictions, and in the case of `Node`, this region must be the same where both the left tree `lt` and the right tree `rt` live.

Several regions can be inferred when nested types are used, as different components of the data structure may live in different regions. For instance, in the declaration

```
data Table a b = TBL [(a,b)]
```

the following three region types will be inferred for the `Table` datatype:

$$\textbf{data}\ \mathit{Table}\ a\ b\ @\rho_1\rho_2\ \rho_3 = \mathit{TBL}\ ([(a,b)@\rho_1]@\rho_2)@\rho_3$$

In that case we adopt the convention that the last region type in the list is the outermost one where the constructed values of the datatype are to be allocated.

After region inference, function applications are annotated with the additional region arguments which the function uses to construct DSs. For instance, in the definition

```
concat []     ys  = ys
concat (x:xs) ys  = x : concat xs ys
```

the compiler infers the type $\mathit{concat} :: \forall a\rho_1\rho_2.[a]@\rho_1 \to [a]@\rho_2 \to \rho_2 \to [a]@\rho_2$ and annotates the text as follows:

```
concat []     ys @ r = ys
concat (x:xs) ys @ r = (x : concat xs ys @ r) @ r
```

The region of the output list and that of the second input list must be the same due to the sharing between both lists introduced by the first equation. Functions

are also polymorphic in region types, i.e. they can accept as arguments any actual regions provided that they satisfy the type restrictions (for instance, in the case of `concat`, that the second and the output lists must live in the same region). Sometimes, several region arguments are needed as in:

```
partition y [] = ([],[])
partition y (x:xs) | x <= y = (x:ls,gs)
                   | x >  y = (ls  ,x:gs)
            where (ls,gs) = partition y xs
```

The inferred type (without quantifiers) is *partition* :: $a \rightarrow [a]@\rho_1 \rightarrow \rho_2 \rightarrow \rho_3 \rightarrow \rho_4 \rightarrow ([a]@\rho_2, [a]@\rho_3)@\rho_4$. The algorithm splits the output in as many regions as possible. This gives more general types and, as we will see, makes the garbage to be deallocated sooner.

The allocation and deallocation of regions is bound to function calls. A *working region*, denoted by the reserved identifier **self**, is allocated when entering the call and deallocated when exiting it. Inside the function, data structures not belonging to the output may be built there. When a function body is executing, the *live* regions are the working regions of all the active function calls leading to this one. The *live* regions in scope are those where the argument DSs live (for reading), those received as additional arguments (for reading and writing) and the own **self** region. The following example builds an intermediate tree not needed in the output:

```
treesort xs = inorder (makeTree xs)
```

where the inferred types are as follows:

$$\begin{array}{ll} \mathit{makeTree} & :: [a]@\rho_1 \rightarrow \rho_2 \rightarrow \mathit{Tree}\ a@\rho_2 \\ \mathit{inorder} & :: \mathit{Tree}\ a@\rho_1 \rightarrow \rho_2 \rightarrow [a]@\rho_2 \\ \mathit{treesort} & :: [a]@\rho_1 \rightarrow \rho_2 \rightarrow [a]@\rho_2 \end{array}$$

After region inference, the definition is annotated as follows:

```
treesort xs @ r = inorder (makeTree xs @ self) @ r
```

i.e. the intermediate tree is created in the **self** region and it is deallocated upon termination of `treesort`. Destruction facilities are also available in the language, but they are orthogonal to region inference. As an example, here is a constant space append funcion:

```
concatD []!     ys  = ys
concatD (x:xs)! ys  = x : concatD xs ys
```

The ! mark is the way programmers indicate that the left list must be destroyed. The type inferred is now *concatD* :: $[a]!@\rho_1 \rightarrow [a]@\rho_2 \rightarrow \rho_2 \rightarrow [a]@\rho_2$, where the ! mark in the type expresses that the corresponding argument is *condemned* by the function. This feature and the type system allowing to use it in a safe way have been explained in previous papers [6, 7]. The constant space consumption is due to that, at each recursive call, a cell is deleted by the pattern matching while a new one is allocated by the $(:)$ construction.

prog	$\rightarrow$	$\overline{data_i}^n; \overline{dec_j}^m; e$	
data	$\rightarrow$	**data** $T\ \overline{\alpha_i}^n\ @\ \overline{\rho_j}^m = \overline{C_k\ \overline{t_{ks}}^{n_k}\ @\ \rho_m}^l$	{recursive, polymorphic data type}
dec	$\rightarrow$	$f\ \overline{x_i}^n\ @\ \overline{r_j}^l = e$	{recursive, polymorphic function}
e	$\rightarrow$	a	{atom: literal c or variable x}
		$\mid x@r$	{copy}
		$\mid x!$	{reuse}
		$\mid f\ \overline{a_i}^n\ @\ \overline{r_j}^l$	{function application}
		$\mid$ **let** $x_1 = be$ **in** e	{non-recursive, monomorphic}
		$\mid$ **case** x **of** $\overline{alt_i}^n$	{read-only case}
		$\mid$ **case!** x **of** $\overline{alt_i}^n$	{destructive case}
alt	$\rightarrow$	$C\ \overline{x_i}^n \rightarrow e$	
be	$\rightarrow$	$C\ \overline{a_i}^n\ @\ r$	{constructor application}
		$\mid e$	

FIGURE 15.1. ***Core-Safe*** **language definition**

By using destructive pattern matching, the programmer may reduce the live memory, but this feature can sometimes produce programs rejected by the type system. The region mechanism will not however lead to rejecting programs. It always succeeds although, of course, it will not be able to detect all garbage. Section 15.4 explains how the algorithm works and shows that it is optimal in the sense that it assigns as many DS as possible to the **self** region of the function at hand. This implies that these DSs will be deallocated at function termination. So, their memory space could be immediately reused.

The examples shown above are written in *Full-Safe*. The region inference algorithm is performed at this level. There exists a desugared version, called *Core-Safe* in which regions and pattern matching are made explicit via the @ notation and **case**-expressions. Its syntax is shown in Figure 15.1. A program *prog* is a sequence of possibly recursive polymorphic data and function definitions followed by a main expression e, using them, whose value is the program result. The abbreviation $\overline{x_i}^n$ stands for $x_1 \cdots x_n$. Constructions are only allowed in **let** bindings, and atoms are used in function applications, **case/case!** discriminant, copy and reuse. After region inference, programs written in *Full-Safe* are translated into *Core-Safe*, in which the operational semantics and the type system are defined.

15.3 BIG-STEP OPERATIONAL SEMANTICS

In Fig. 15.2 we show the big-step operational semantics of the core language expressions. We use $v, v_i, \ldots$ to denote values, i.e. either heap pointers or basic constants, and $p, p_i, q, \ldots$ to denote heap pointers. We use $a, a_i, \ldots$ to denote atoms, i.e. either program variables or basic constants. The former are denoted by $x, x_i, \ldots$ and the latter by c, c_i etc. Finally, we use $r, r_i, \ldots$ to denote region variables.

A judgement of the form $E \vdash h, k, e \Downarrow h', k', v$ means that expression e is successfully reduced to normal form v under runtime environment E and heap h with $k+1$ regions, ranging from 0 to k, and that a final heap h' with $k'+1$ regions is produced as a side effect. Runtime environments E map program variables to

$$E \vdash h,k,c \Downarrow h,k,c \quad [Lit]$$

$$E[x \mapsto v] \vdash h,k,x \Downarrow h,k,v \quad [Var_1]$$

$$\frac{j \leq k \quad (h',p') = copy(h,p,j)}{E[x \mapsto p, r \mapsto j] \vdash h,k,x@r \Downarrow h',k,p'} \; [Var_2]$$

$$\frac{fresh(q)}{E[x \mapsto p] \vdash h \uplus [p \mapsto w],k,x! \Downarrow h \uplus [q \mapsto w],k,q} \; [Var_3]$$

$$\frac{(f\ \overline{x_i}^n @\ \overline{r_j}^m = e) \in \Sigma \quad [\overline{x_i \mapsto E(a_i)}^n, \overline{r_j \mapsto E(r'_j)}^m, self \mapsto k+1] \vdash h,k+1,e \Downarrow h',k'+1,v}{E \vdash h,k,f\ \overline{a_i}^n @\ \overline{r'_j}^m \Downarrow h' \mid_{k'},k',v} \; [App]$$

$$\frac{E \vdash h,k,e_1 \Downarrow h',k',v_1 \quad E \cup [x_1 \mapsto v_1] \vdash h',k',e_2 \Downarrow h'',k'',v}{E \vdash h,k,\textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2 \Downarrow h'',k'',v} \; [Let_1]$$

$$\frac{j \leq k \quad fresh(p) \quad E \cup [x_1 \mapsto p] \vdash h \uplus [p \mapsto (j, C\,\overline{v_i}^n)],k,e_2 \Downarrow h',k',v}{E[r \mapsto j, \overline{a_i \mapsto v_i}^n] \vdash h,k,\textbf{let}\ x_1 = C\,\overline{a_i}^n @ r\ \textbf{in}\ e_2 \Downarrow h',k',v} \; [Let_2]$$

$$\frac{C = C_r \quad E \cup [\overline{x_{ri} \mapsto v_i}^{n_r}] \vdash h,k,e_r \Downarrow h',k',v}{E[x \mapsto p] \vdash h[p \mapsto (j, C\,\overline{v_i}^{n_r})],k,\textbf{case}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \rightarrow e_i}^m \Downarrow h',k',v} \; [Case]$$

$$\frac{C = C_r \quad E \cup [\overline{x_{ri} \mapsto v_i}^{n_r}] \vdash h,k,e_r \Downarrow h',k',v}{E[x \mapsto p] \vdash h \uplus [p \mapsto (j, C\,\overline{v_i}^{n_r})],k,\textbf{case!}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \rightarrow e_i}^m \Downarrow h',k',v} \; [Case!]$$

FIGURE 15.2. Operational semantics of *Core-Safe* expressions

values and region variables to actual region identifiers. We adopt the convention that for all E, if c is a constant, $E(c) = c$.

A heap h is a finite mapping from fresh variables p (we call them heap pointers) to construction cells w of the form $(j, C\,\overline{v_i}^n)$, meaning that the cell resides in region j. We say that $region(w) = j$. Actual region identifiers j are just natural numbers. Formal regions appearing in a function body are either region variables r corresponding to formal arguments or the constant *self*. Deviating from other authors, by $h[p \mapsto w]$ we denote a heap h where the binding $[p \mapsto w]$ is highlighted. On the contrary, by $h \uplus [p \mapsto w]$ we denote the disjoint union of heap h with the binding $[p \mapsto w]$. By $h \mid_k$ we denote the heap obtained by deleting from h those bindings living in regions greater than k, and by $dom(h)$, the set $\{p \mid [p \mapsto w] \in h\}$.

The semantics of a program is the semantics of the main expression e in an environment Σ, which is the set containing all the function and data declarations.

Rules *Lit* and Var_1 just say that basic values and heap pointers are normal forms. Rule Var_2 executes a copy expression copying the DS pointed to by p and living in a region j' into a (possibly different) region j. The runtime system function *copy* follows the pointers in recursive positions of the structure starting at p and creates in region j a copy of all recursive cells. We foresee that some restricted type informaton is available in our runtime system so that this function can be implemented. The pointers in non recursive positions of all the copied cells

$$\frac{\begin{array}{c}\text{fresh}(\rho_{self}),\quad \rho_{self} \notin regions(s)\\ \Gamma + \overline{[x_i : t_i]}^{n} + \overline{[r_j : \rho_j]}^{l} + [self : \rho_{self}] + [f : \overline{t_i}^{n} \to \overline{\rho}^{l}_{j} \to s] \vdash e : s\end{array}}{\{\Gamma\}\ \ f\ \overline{x_i}^{n}\ @\ \overline{r}^{l}_{j} = e\ \ \{\Gamma + [f : gen(\overline{t_i}^{n} \to \overline{\rho}^{l}_{j} \to s, \Gamma)]\}}\ \text{[FUNB]}$$

$$\frac{\Sigma(C) = \sigma \qquad \overline{s_i}^{n} \to \rho \to T\ @\overline{\rho}^{m} \trianglelefteq \sigma \qquad \Gamma = \bigoplus_{i=1}^{n}[a_i : s_i] + [r : \rho]}{\Gamma \vdash C\ \overline{a_i}^{n} @ r : T\ @\overline{\rho}^{m}}\ \text{[CONS]}$$

FIGURE 15.3. Rule for function definitions

are kept identical in the new cells. This implies that both DSs may share some sub-structures.

In rule Var_3, the binding $[p \mapsto w]$ in the heap is deleted and a fresh binding $[q \mapsto w]$ to cell w is added. This action may create dangling pointers in the live heap, as some cells may contain free occurrences of p.

Rule *App* shows when a new region is allocated. Notice that the body of the function is executed in a heap with $k+2$ regions. The formal identifier *self* is bound to the newly created region $k+1$ so that the function body may create DSs in this region or pass this region as a parameter to other function calls. Before returning from the function, all cells created in region $k'+1$ are deleted. This action is another source of possible dangling pointers.

Rules Let_1, Let_2, and *Case* are the usual ones for an eager language, while rule *Case*! expresses what happens in a destructive pattern matching: the binding of the discriminant variable disappears from the heap. This action is the last source of possible dangling pointers.

In the following, we will feel free to write the derivable judgements as $E \vdash h,k,e \Downarrow h',k,v$ because of the following:

Proposition 15.1. *If $E \vdash h,k,e \Downarrow h',k',v$ is derivable, then $k = k'$.*

Proof. Straightforward, by induction on the depth of the derivation.

15.4 THE REGION INFERENCE ALGORITHM

As we have shown with examples, the aim of the region inference algorithm is to annotate both the program and the types of the functions with region variables and region type variables respectively. The main correctness requirement is that the annotated type of each function can be assigned to the corresponding annotated function in the type system defined in [6]. The main constraints posed by that system with respect to regions are reflected in the function and constructor typing rules, shown in Figure 15.3.

In rule [FUNB] the fresh (local) program region variable *self* is assigned a fresh type variable ρ_{self} that cannot appear in the function result type. This prevents dangling pointers arising by region deallocation at the end of a function call. The only regions in scope are *self* and the argument regions. Consequently, the

region types appearing in the resulting type s can only be those where the input data structures live, or the given argument regions.

The types of the constructors are given in an initial environment built from the datatype declarations. These types should reflect the fact that the recursive substructures live in the same region. For example, in the case of lists and trees:

$$[\,] : \forall a, \rho.\rho \rightarrow [a]@\rho$$
$$(:) : \forall a, \rho.a \rightarrow [a]@\rho \rightarrow \rho \rightarrow [a]@\rho$$
$$\mathit{Empty} : \forall a, \rho.\rho \rightarrow \mathit{Tree}\ a@\rho$$
$$\mathit{Node} : \forall a, \rho.\mathit{Tree}\ a@\rho \rightarrow a \rightarrow \mathit{Tree}\ a@\rho \rightarrow \rho \rightarrow \mathit{Tree}\ a@\rho$$

As a consequence, rule [CONS] may force some of the actual arguments to live in the same regions.

The inference algorithm proceeds in the following phases:

1. Annotation of the **data** declarations with region variables and inference of the types of the constructors.
2. Type and region inference of each function definition, recording the inferred type in a type environment used to infer subsequent function definitions. For each function, the steps are:
   (a) Generation of fresh type region variables for (known) function and constructor applications, and for copy expressions.
   (b) Generation of equations between types reflecting the unifications needed for the function to be typable according to the type system mentioned before. Some of these equations correspond to the usual Hindley-Milner type inference, e.g. $a = [b] \rightarrow b$. We will concentrate on those concerning region type variables, e.g. $\rho_1 = \rho_2$.
   (c) Resolution of the unification equations and application of the resulting substitution to the function type.
   (d) Region inference and region annotation of the function body. The algorithm decides how many region arguments are necessary and which data structures may live in the local working region *self*.

15.4.1 Region inference of datatype declarations

The algorithm traverses the data declarations and annotates them with region variables. A fresh region variable is generated for each non-recursive nested data type and one more for the type being defined, which is placed as an additional argument for each constructor. Only the recursive occurrences are forced to have the same region arguments. All the region variables are reflected in the type so that all the regions in which the structure has a portion are known. As an example, consider the following declaration:

```
data T a b = C (a,a) [b] | D a (T a b)
```

Region variables ρ_1, ρ_2 and ρ_3 are generated and placed as shown here:

data $T\ a\ b\ @\rho_1\ \rho_2\ \rho_3 = C\ (a,a)@\rho_1\ [b]@\rho_2\ @\rho_3 \mid D\ a\ (T\ a\ b\ @\rho_1\ \rho_2\ \rho_3)\ @\rho_3$

and the constructors types will be the following:

$$C :: (a,a)@\rho 1 \to [b]@\rho_2 \to \rho_3 \to T\ a\ b\ @\rho_1\ \rho_2\ \rho_3$$
$$D :: a \to T\ a\ b\ @\rho_1\ \rho_2\ \rho_3 \to \rho_3 \to T\ a\ b\ @\rho_1\ \rho_2\ \rho_3$$

15.4.2 Region inference of function definitions

For each function the algorithm must decide how many region arguments are required and in which regions the data created by the function body are built. Formalising the previously described steps for a particular function f, the algorithm builds the following sets:

1. The fresh type region variables generated during phase (a) at each construction, known-function application, and copy expression are collected in the set of *explicit fresh* region types, called $Fresh_{expl}$.
2. After phases (b) and (c) a substitution θ is obtained. The algorithm applies it both to the function type and to the set $Fresh_{expl}$. As a result, a substituted type t_f and a set R_{expl} of *explicit* region types are obtained.
3. Now, the set R_{in} of type region variables in the argument part of t_f, and the set R_{out} of those in the result part are built.
4. Let ρ_{self} be an additional fresh type for *self*.

Given these three sets, the region inference problem can be specified as finding two sets R_{arg} and R_{self}, respectively standing for the set of region types needed as additional arguments of f and the set of region types that must be unified with ρ_{self}, subject to the following restrictions:

1. $R_{expl} \subseteq R_{self} \cup R_{arg}$
2. $R_{self} \cap R_{arg} = \emptyset$
3. $R_{self} \cap (R_{in} \cup R_{out}) = \emptyset$
4. $R_{out} \subseteq R_{in} \cup R_{arg}$

The first one expresses that everything built by f must be in regions in scope. The second and third ones tell that region *self* is fresh and hence different from any other region received as an argument or where an input argument lives. These restrictions and the extension of (3) to R_{out} are enforced by the typing rule [FUNB]. The last one expresses the semantics of f and a property of the type system: a function can only return as a result either portions of the input arguments, or portions explicitly built, or a combination of both. Since the region of every part of a data structure is visible in its type, the restriction must hold.

Notice that an algorithm choosing $R_{arg} = R_{expl} \cup R_{out}$ and $R_{self} = \emptyset$ would be correct according to this specification. But this solution would be very poor as, on the one hand no construction would ever be done in the *self* region and, on the other, there might be region arguments never used. We look for an optimal solution in the following two senses:

- We want R_{arg} to be as small as possible, so that only those regions where data are built are given as arguments.
- We want R_{self} to be as big as possible, so that the maximum amount of memory is deallocated at function termination.

Our algorithm computes the following solution:

$$\boxed{R_{self} = R_{expl} - (R_{in} \cup R_{out})} \qquad \boxed{R_{arg} = (R_{out} - R_{in}) \cup (R_{expl} \cap R_{in})}$$

The last step in the inference process is adding region arguments to f according to the cardinality of R_{arg}, and the generation of region arguments in copy expressions, in function application (including the recursive ocurrences of f), and in constructor applications. A fresh program region variable is created for each region type $\rho \in R_{arg}$. Then, the body is traversed again and, if a region type belongs to R_{self}, then the program variable *self* is written as an argument; if it belongs to R_{arg}, then its corresponding region variable is written. Additionaly, a substitution θ mapping all type variables in R_{self} to variable ρ_{self} is applied to all the types.

15.4.3 Correctness, optimality and efficiency

First we prove that the solution satisfies the above specification, so it is correct:

1. $R_{expl} \subseteq (R_{expl} - (R_{in} \cup R_{out})) \cup ((R_{out} - R_{in}) \cup (R_{expl} \cap R_{in}))$
2. $(R_{expl} - (R_{in} \cup R_{out})) \cap ((R_{out} - R_{in}) \cup (R_{expl} \cap R_{in})) = \emptyset$
3. $(R_{expl} - (R_{in} \cup R_{out})) \cap (R_{in} \cup R_{out}) = \emptyset$
4. $R_{out} \subseteq R_{in} \cup ((R_{out} - R_{in}) \cup (R_{expl} \cap R_{in}))$

The four immediately follow by set algebra. We now will show that it is optimal: let us assume a different solution R'_{self}, R'_{arg} satisfying the above restrictions. We show that $R'_{self} \subseteq R_{self}$ and $R_{arg} \subseteq R'_{arg}$:

Let us assume $\rho \in R_{arg}$, By definition of R_{arg}, we can distinguish two cases:

1. $\rho \in R_{out} - R_{in}$. By constraint (4) this implies $\rho \in R'_{arg}$.
2. $\rho \in R_{expl} \cap R_{in}$. By constraint (3) this implies $\rho \notin R'_{self}$. By constraint (1) this implies $\rho \in R'_{arg}$.

Then, R_{arg} is as small as possible. By constraints (2) and (1), then R_{self} will be as big as possible.

Our sets are implemented as balanced trees, and operations such as '$\cup$', '$\cap$', and '$-$' are done with a cost in $\Theta(n+m)$, being n and m the cardinalities of the respective sets, so the inference algorithm is linear with the number or region types occurring in a function body. This raises the question of how is the relation between the number of region types and the text size. Obviously, R_{expl} cannot grow faster than the text size, but R_{in} and R_{out} cardinalities depend on the datatype declarations of the argument and result types. We have found pathological examples of datatype declarations in which the number of region types can be made to grow

exponentially with the size of the declarations, but usually this number is much smaller (one to eight regions cover the huge majority of examples we have tried). As we expect the number of region types in a function to grow usually slower than its text size, the dominant cost of the algorithm is determined in these cases by the two traversals of the function body, which are linear with that size.

15.5 CORRECTNESS OF THE TYPE SYSTEM

In this section a correctness proof of the *Safe*'s type system with respect to the operational semantics is sketched. This type system ensures that dangling pointers are never accessed by a well-typed program. This includes dangling pointers arising from, on the one hand, the destructive pattern matching and, on the other hand, the topmost region deallocation when a function call finishes. In this section we shall deal only with the latter source of dangling pointers. A detailed description of the type system and the full proof of correctness is given in [6]. For the purposes of this section, we can restrict ourselves to the usual Hindley-Milner type system rules, whose types are extended with region variables.

At the end of each function call the topmost region (which is only referenced by the current *self* in scope) is deallocated. The safety of this deallocation is shown by proving that the structure returned by the function call does not reside in *self*. The key idea is to establish a correspondence between type region variables ρ and region numbers j. If a variable admits the algebraic type $T@\overline{\rho_i}^n$ and it is related by E to a pointer p, we have to find out which concrete region of the structure pointed to by p corresponds to every ρ_i. This correspondence is called *region instantiation*. Intuitively, a region instantiation is a function which maps type region variables to dynamic regions (in fact, natural numbers). Two region instantiations θ and θ' are said to be *consistent* if they bind common type region variables to the same region. In this case, their union of their bindings is denoted by $\theta \cup \theta'$. Given a heap, a pointer and a type, the function *build* returns the corresponding region instantiation:

$$
\begin{array}{lcll}
build(h,c,B) & = & \emptyset & \\
build(h,p,T\ \overline{t_i}^n@\overline{\rho_i}^m) & = & \emptyset & \text{if } p \notin dom(h) \\
build(h,p,T\ \overline{t_i}^n@\overline{\rho_i}^m) & = & [\rho_m \rightarrow j] \cup \bigcup_{i=1}^{n_k} build(h,v_i,t_{ki}) & \text{if } p \in dom(h) \\
\multicolumn{4}{l}{\textbf{where} \quad h(p) = (j, C_k\ \overline{v_i}^{n_k})} \\
\multicolumn{4}{l}{\qquad\quad \overline{t_{ki}}^{n_k} \rightarrow \rho_m \rightarrow T\ \overline{t_i}^n@\overline{\rho_i}^m \trianglelefteq \Sigma(C_k)}
\end{array}
$$

Now we define a notion of *consistency* between the variables belonging to a variable environment E. Intuitively, it means that the correspondences between region type variables and concrete regions of each element of $dom(E)$ do not contradict each other, that is, the results of each $build(h,E(x),\Gamma(x))$, where $x \in dom(E)$ are well-defined, and also is their union, which we call the *witness* of this consistency relation (see [6] for a formal definition). The following theorem proves that consistency is preserved by evaluation.

Theorem 15.2. *Let us assume that* $E \vdash h,k,e \Downarrow h',k,v$ *and that* $\Gamma \vdash e : t$. *If* E *and* h *are consistent under* Γ *with witness* θ, *then* $build(h',v,t)$ *is well-defined and consistent with* θ.

Proof. By induction on the depth of the $\Downarrow$ derivation [6].

As a result, if two variables have the same outer region ρ in their type, the cells bound to them at runtime will live in the same actual region. Since the type system (see rule [FUNB] in Fig. 15.3) enforces that the variable ρ_{self} does not occur in the type of the function result, then every data structure returned by the function call does not have cells in *self*. This implies that the deallocation of the $(k+1)$-th region (which always is bound to *self*) at the end of a function call does not generate dangling pointers.

15.6 EXAMPLES

As a first example, consider the previously defined function *partition*. A region variable ρ_1 is created for the input list, so that it has type $[Int]@\rho_1$. In addition seven fresh type region variables are generated, one for each constructor application, let say ρ_2 to ρ_8, and so $R_{expl} = \{\rho_2, \ldots, \rho_8\}$. We show them as annotations in the program just in order to better explain the example:

$$\begin{array}{l} partition\ y\ [\,] = ([\,] :: \rho_2, [\,] :: \rho_3) :: \rho_4 \\ partition\ y\ (x : xs) \\ \qquad \mid x \leq y = (x : ls :: \rho_5, gs) :: \rho_6 \\ \qquad \mid x > y = (ls, x : gs :: \rho_7) :: \rho_8 \\ \qquad \textbf{where}(ls, gs) = partition\ y\ xs \end{array}$$

The type inference rules generate the following equations relative to these type region variables: $\rho_2 = \rho_5$, $\rho_3 = \rho_7$, and $\rho_4 = \rho_6 = \rho_8$, so a possible R_{expl} in this case is $\{\rho_2, \rho_3, \rho_4\}$. After unification, the type of partition is $Int \to [Int]@\rho_1 \to ([Int]@\rho_2, [Int]@\rho_3)@\rho_4$, so $R_{in} = \{\rho_1\}$ and $R_{out} = \{\rho_2, \rho_3, \rho_4\}$.

Then, $R_{self} = R_{expl} - (R_{in} \cup R_{out}) = \emptyset$, and consequently the solution $R_{arg} = R_{expl}$ and $R_{self} = \emptyset$ is the only one satisfying the specification. So the definitive type of *partition* is:

$$partition :: Int \to [Int]@\rho_1 \to \rho_2 \to \rho_3 \to \rho_4 \to ([Int]@\rho_2, [Int]@\rho_3)@\rho_4$$

In the last step, the algorithm is annotated as follows:

$$\begin{array}{l} partition\ y\ [\,]@r_2\ r_3\ r_4 = ([\,]@r_2, [\,]@r_3)@r_4 \\ partition\ y\ (x : xs)@r_2\ r_3\ r_4 \\ \qquad \mid x \leq y = (x : ls@r_2, gs)@r_4 \\ \qquad \mid x > y = (ls, x : gs@r_3)@r_4 \\ \qquad \textbf{where}(ls, gs) = partition\ y\ xs@r_2\ r_3\ r_4 \end{array}$$

Consider now the treesort algorithm before. Let us assume that ρ_1 is the region type of the argument *xs*. Using the types for *inorder* and *makeTree* given above, two fresh region variables are generated, one for each function call, e.g. ρ_3 for *makeTree* and ρ_2 for *inorder*. Then, $R_{expl} = \{\rho_2, \rho_3\}$. There are no equations between them. After the unifications, the type of *treesort* is $[Int]@\rho_1 \to [Int]@\rho_2$,

so $R_{in} = \{\rho_1\}$ and $R_{out} = \{\rho_2\}$. As we have previously said, a correct naive solution would be $R_{arg} = \{\rho_2, \rho_3\}$ and $R_{expl} = \emptyset$. However, our algorithm assigns $R_{arg} = \{\rho_2\}$ and $R_{self} = \{\rho_3\}$, expressing that the temporal tree may live in region *self*, and it will be deallocated upon termination of *treesort*. The definition is then annotated as indicated in Section 15.2.

Our last example implements the dynamic programming approach to the matrix chain multiplication problem: given a sequence $A_1 \cdots A_n$ of matrices we want to decide in what order should they be multiplicated, so as to perform the minimum number of scalar products. We shall assume the existence of a list of dimensions $[d_0, \ldots, d_n]$, so that the i-th matrix has size $d_{i-1} \times d_i$. We denote by $c_{i,j}$ the minimum number of scalar products needed to multiply the subchain $A_i \cdots A_j$. It is defined for each $1 \leq i, j \leq n$ as follows:

$$c_{i,j} = \begin{cases} 0 & \text{if } i \geq j \\ min\ \{c_{i,k} + c_{k+1,j} + d_{i-1} * d_k * d_j \mid i \leq k \leq j-1\} & \text{otherwise} \end{cases}$$

The $c_{i,j}$ already calculated are stored in a table which maps coordinates to values. We assume the following table operations available:

```
emptyTable   :: Table k v
insertTableD :: Table k v! -> k -> v -> Table k v
searchTable  :: Table k v -> k -> Maybe v
```

The function `enumPairsFromTo` returns a list of pairs which establishes the order in which the $c_{i,j}$ are calculated and stored in the table. A function call `enumPairsFromTo 1 n` returns the list: $[(1,1), \ldots, (1,n), \ldots, (n,1), \ldots, (n,n)]$. It is defined as follows:

```
enumPairsFromTo i j = enumPairsFromTo' i i j
enumPairsFromTo' i i' j'| i >  j' = []
                        | True    = concat (putBefore i (enumFromTo i' j'))
                                           (enumPairsFromTo' (i+1) i' j')
```

where the function `enumFromTo`, given two integers i and j, returns the ascending list $[i, i+1, \ldots, j]$. On the other hand, `putBefore` converts a list $[a_1, \ldots, a_n]$ into a list of tuples $[(k, a_1), \ldots, (k, a_n)]$ where k is the value passed as the first parameter. After the region inference, the corresponding types are:

$$\begin{array}{rcl} \mathit{enumFromTo} & :: & \mathit{Int} \to \mathit{Int} \to \rho_1 \to [\mathit{Int}]@\rho_1 \\ \mathit{putBefore} & :: & a \to [b]@\rho_1 \to \rho_2 \to \rho_3 \to [(a,b)@\rho_3]@\rho_2 \\ \mathit{enumPairsFromTo'} & :: & \mathit{Int} \to \mathit{Int} \to \mathit{Int} \to \rho_1 \to \rho_2 \to [(\mathit{Int}, \mathit{Int})@\rho_1]@\rho_2 \\ \mathit{enumPairsFromTo} & :: & \mathit{Int} \to \mathit{Int} \to \rho_1 \to \rho_2 \to [(\mathit{Int}, \mathit{Int})@\rho_1]@\rho_2 \end{array}$$

After the region inference, the function `enumPairsFromTo'` is decorated as follows:

```
enumPairsFromTo' i i' j' @ r1 r2
    | i >  j' = [] @ r2
    | True    = concat (putBefore i (enumFromTo i' j' @ self) @ r1 self)
                       (enumPairsFromTo' (i+1) i' j' @ r1 r2) @ r2
```

Indeed, the list $[i', \ldots, j']$ returned by `enumFromTo` is assigned the working region *self*, since it is not part of the result. Moreover, the function call to `putBefore` creates each tuple in the output region, while the cons-nil spine of the result is created in the working region. This is due to the call to `concat`.

In order to compute a $c_{i,j}$, the set $\{c_{i,k} + c_{k+1,j} + d_{i-1} * d_k * d_j \mid i \leq k \leq j-1\}$, from which the minimum is to be extracted, may be needed. This set (implemented as a list of integers) is calculated by the `createList` function:

```
createList i j d c = createList' i i j d c
createList' k i j d c | k == j = []
                      | k <  j = (c1+(c2+p)):createList' (k+1) i j d c
                      where (Just c1) = searchTable c (i,k)
                            (Just c2) = searchTable c (k+1,j)
                            p = (d !! (i-1)) * ((d !! k) * (d !! j))
```

where `!!` is the list indexing operator. The pairs `(i,k)` and `(k+1,j)` given to the `searchTable` function and the resulting `Just` values are assigned the region *self* after the region inference. The list itself will be constructed in the output region. Each $c_{i,j}$ is calculated by means of `calculateCoord` and `calculateTable`:

```
calculateCoord (i,j) d c | i >= j = 0
                         | i <  j = minimum (createList i j d c)
calculateTable d c [] = c
calculateTable d c (x:xs) = calculateTable d
                            (insertTableD c x (calculateCoord x d c)) xs
```

where the call to `createList` is inferred as `createList i j d c @ self`, as the list returned is not part of the result. Finally, `calculateTable` receives the list `d` containing the d_i used to represent the size of each matrix, the initial map from pairs (i, j) to values $c_{i,j}$ and the list of pairs to be generated and inserted into this map. The algorithm starts with an empty map and proceeds generating each $c_{i,j}$ in the order specified by the call to `enumPairsFromTo 0 n`. Once the whole table is computed, the element $c_{1,n}$ gives us the desired result.

```
chainedMatrix d = searchTable (calculateTable d t
                               (enumPairsFromTo 0 n)) (1,n)
                  where n = length d
                        t = emptyTable
```

The regions corresponding to every generated DS (that is, the pair `(1,n)` and the DSs returned by `emptyTable`, `calculateTable` and `enumPairsFromTo`) are inferred as *self*, so that the memory allocated for these DSs will be recovered when the call to `chainedMatrix` terminates.

15.7 RELATED WORK AND CONCLUSIONS

The pioneer work on region inference is that of M. Tofte, J.-P. Talpin and their colleagues on the ML-Kit compiler [10, 8] (in what follows, TT). They address a much more general problem than we do. Their language is higher-order and they support polymorphic recursion in the region arguments of functional types. This means that recursive calls can be made with possibly different actual region arguments.

The TT algorithm has two phases, respectively called *S* and *R* algorithms. The *S*-algorithm just generates fresh region variables for values and introduces the lexical scope of the regions by using a **letregion** construct. The main idea of a typed expression **letregion** ρ **in** $e : \mu$ is that region ρ does not occur in type μ and so it can be deallocated upon the evaluation of e. Our algorithm has some resemblances with this part of the inference, in the sense that we decide to unify with ρ_{self} all the region variables not occurring in the result type of a function. They do not claim their algorithm to be optimal but in fact they create as many regions as possible, trying to made local *all* the regions not needed in the final value. The consequence is that unneeded regions are deallocated as soon as possible, so optimising the resident space needed by the program.

The *R*-algorithm is responsible for assigning types to recursive functions. It deals with polymorphic recursion and also computes a fixpoint.

Both use a higher-order unification based on directed graphs and on a UNION-FIND data structure. If n is the size of the term being typed, the *S*-algorithm runs in time $O(n^3)$ and the *R*-algorithm in time $O(n^4)$ in the worst case.

The implementation of ML-Kit in 1998 [9] added a third phase called *storage mode analysis* which introduced a region *resetting* action previously to some allocations. In order to take profit of this analysis, the programmer should introduce *copy* expressions in specific parts of the text. A further work [1] added a constraint based and a control flow based analyses after the TT inference. These resulted in modifying the text by delaying as much as possible some region allocations, and by moving forwards some deallocations, without compromising pointer safety. Again, the programmer needs to introduce the *copy* function in appropriate places. Some space leaks could be avoided with respect to using plain TT.

The main difficulty with the TT system is that the region life-times are bound to expression evaluation and then follow a stack discipline, while the object life-times does not usually follow this discipline. A radical deviation from these approaches is [3] which introduces a type system in which region life-times are not necessarily nested. The compiler annotates the program with region variables and supports operations for allocation, releasing, aliasing and renaming. A reference-counting analysis is used in order to decide when a released region should be deallocated. The used language is however first-order. The inference algorithm is described in [5] and it can be defined as a global abstract interpretation of the program by following the control flow of the functions in a backwards direction. Although the authors do not give either asymptotic costs or actual benchmarks, from their explanations it is deduced that this cost could grow more than cuadratically with the program text size in the worst case, as a global fixpoint must be computed and a region variable may disappear at each iteration. This lack of modularity could make the approach unpractical for large programs.

Another approach is [2] in which type-safe primitives are defined for creating, accessing and destroying regions. These are not restricted to have nested lifetimes. Programs are written in a C-like language called *Cyclone*, then translated to a variant of λ-calculus, and then type-checked. So, the price of this flexibility is having explicit region control in the language.

The main virtue of our design is its simplicity. The previous works have no restrictions on the placement of cells belonging to the same data structure. Also, in the case of TT and its derivatives, they support higher-order and polymorphic recursion. As a consequence, the inference algorithms are more complex and costly. Our restrictions make it possible to integrate the region inference as a slight extension of the Hindley-Milner type inference algorithm.

In our language, regions also suffer from the nested lifetimes constraint, since both region allocation and deallocation are bound to function calls. However, the destructive pattern matching facility compensates for this, since it is possible to dispose of a data structure without deallocating the whole region in which it resides. Allocation and destruction of data structures are not necessarily nested, and our type system protects the programmer against missuses of this feature. The price of this flexibility is explicit deallocation of cells. Allocation is implicit. Finally, the inference technique shown in this paper could also be applied to any first-order functional language, since *Safe*'s destruction features are orthogonal to the region inference mechanism.

REFERENCES

[1] A. Aiken, M. Fähndrich, and R. Levien. Better static memory management: improving region-based analysis of higher-order languages. In *Proceedings of the ACM SIGPLAN PLDI'95*, pages 174–185. ACM Press, 1995.

[2] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *ACM PLDI'02, Berlin*, pages 282–293, 2002.

[3] F. Henglein, H. Makholm, and H. Niss. A direct approach to control-flow sensitive region-based memory management. In *PPDP'01, Firenze, Italy*, pages 175–186. ACM Press, 2001.

[4] S. Lucas and R. Peña. Rewriting Techniques for Analysing Termination and Complexity Bounds of SAFE Programs. In *Accepted for presentation in LOPSTR'08, Valencia, Spain*, page (to appear), July 2008.

[5] H. Makholm. A language-independent framework for region inference. Ph.D thesis, Univ. of Copenhagen, Dep. of Computer Science, Denmark, 2003.

[6] M. Montenegro, R. Peña, and C. Segura. A Type System for Safe Memory Management and its Proof of Correctness. In *ACM SIGPLAN PPDP'08, Valencia, Spain*, page (to appear), July 2008.

[7] M. Montenegro, R. Peña, and C. Segura. An Inference Algorithm for Guaranteeing Safe Destruction. In *Accepted for presentation in LOPSTR'08, Valencia, Spain*, page (to appear), July 2008.

[8] M. Tofte and L. Birkedal. A region inference algorithm. *ACM Trans. Program. Lang. Syst.*, 20(4):724–767, 1998.

[9] M. Tofte, L. Birkedal, M. Elsman, N. Hallenberg, T. H. Olesen, P. Sestoft, and P. Bertelsen. Programming with regions in the ML Kit (for version 3). Technical report, Department of Computer Science, 98/25, University of Copenhagen (DIKU), 1998.

[10] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.