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

Manuel Montenegro, Ricardo Peña, Clara Segura

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
montenegro@fdi.ucm.es, {ricardo,csegura}@sip.ucm.es.

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 [7, 8]. In this paper we present an inference algorithm for annotating programs with regions which is both simpler to understand and more efficient than other related algorithms. Programmers are assumed to write programs and to declare datatypes without any reference to regions. The algorithm decides the regions needed by every function. It also allows polymorphic recursion with respect to regions. We show convincing examples of programs before and after region annotation, prove the correctness and optimality of the algorithm, and give its asymptotic cost.

1 Introduction

Safe[1] [7] was introduced as a research platform for investigating the suitability of functional languages for programming small devices and embedded systems with strict memory requirements. The final aim is to be able to infer —at compile time— safe upper bounds on memory consumption for most *Safe* programs. The compiler produces Java bytecode as a target language, so that *Safe* programs can be executed in most mobile devices and web browsers.

In most functional languages memory management is delegated to the runtime system. Fresh heap memory is allocated during program evaluation as long as there is enough free memory available. Garbage collection interrupts program execution in order to copy or mark the live part of the heap so that the rest is considered as free. This does not avoid memory exhaustion if not enough free memory is recovered to continue execution. In that case the program simply

* Work supported by the Ministry of Science grants AP2006-02154, TIN2008-06622-C03-01/TIN (STAMP), and the Madrid Government grant S-0505/TIC/0407 (PROMESAS).

[1] http://dalila.sip.ucm.es/safe

aborts. The main advantage of this approach is that programmers do not have to bother about low level details concerning memory management. Its main disadvantages are:

1. The time delay introduced by garbage collection may prevent the program from providing an answer in a required reaction time.
2. Memory exhaustion may provoke unacceptable personal or economic damage to program users.
3. The programmer cannot easily reason about memory consumption.

These reasons make garbage collectors not very convenient for programming small devices. A possibility is to use heap *regions*, which are disjoint parts of the heap that are dynamically allocated and deallocated. Much work has been done in order to incorporate regions in functional languages. They were introduced by Tofte and Talpin [13, 14] in MLKit by means of a nested **letregion** construct inferred by the compiler. The drawbacks of nested regions are well-known and they have been discussed in many papers (see e.g. [4]). The main problem is that in practice data structures do not always have the nested lifetimes required by the stack-based region discipline.

In order to overcome this limitation several mechanisms have been proposed. An extension of Tofte and Talpin's work [2, 11] allows to *reset* all the data structures in a region, without deallocating the whole region. The AFL system [1] inserts (as a result of an analysis) allocation and deallocation commands separated from the **letregion** construct, which now only brings new regions into scope. In both cases, a deep knowledge about the hidden mechanism is needed in order to optimize the memory usage. In particular, it is required to write copy functions in the program which are difficult to justify without knowing the annotations inferred later by the compiler.

Another more explicit approach is to introduce a language construct to free heap memory. Hofmann and Jost [5] introduce a pattern matching construct which destroys individual constructor cells than can be reused by the memory management system. This allows the programmer to control the memory consumed by the program and to reason about it. However, this approach gives the programmer the whole responsibility for reusing memory, unless garbage collection is used.

In order to overcome the problems related to nested regions, our functional language *Safe* has a semi-explicit approach to memory control: it combines implicit regions with explicit destructive pattern matching, which deallocates individual cells of a data structure. This feature avoid the use of explicit copy functions of other systems. In *Safe*, regions are allocated/deallocated by following a stack discipline associated with function calls and returns. Each function call allocates a local working region, which is deallocated when the function returns. Region management does not add a significant runtime overhead because all its related operations run in constant time (see Sec. 2.3).

Notice that regions and explicit destruction are orthogonal mechanisms: we could have destruction without regions and the other way around. This combination of explicit destruction and implicit regions is novel in the functional

$$
\begin{array}{lll}
prog \rightarrow \overline{data_i}^n ; \overline{dec_j}^m ; e & & \\
data \rightarrow \textbf{data}\ T\ \overline{\alpha_i}^n\ @\ \overline{\rho_j}^m = \overline{C_k\ \overline{t_{ks}}^{n_k}\ @\ \rho_m}^l & & \{\text{recursive, polymorphic data type}\} \\
dec \rightarrow f\ \overline{x_i}^n\ @\ \overline{r_j}^l = e & & \{\text{recursive, polymorphic function}\} \\
e \rightarrow a & & \{\text{atom: literal } c \text{ or variable } x\} \\
\quad \mid f\ \overline{a_i}^n\ @\ \overline{r_j}^l & & \{\text{function application}\} \\
\quad \mid C\ \overline{a_i}^n\ @\ r & & \{\text{constructor application}\} \\
\quad \mid \ldots & & \text{let, case } \ldots
\end{array}
$$

Fig. 1. Simplified *Safe*

programming field. However, destructive pattern matching is not relevant to this paper. More details about it can be found in [7, 8]

Due to the aim of inferring memory consumption upper bounds, at this moment *Safe* is first-order. Its syntax is a (first-order) subset of Haskell extended with destructive pattern matching. Due to this limitation, region inference can be expected to be simpler and more efficient than that of MLKit. Their algorithm runs in time $O(n^4)$ in the worst case, where n is the size of the term, including in it the Hindley-Milner type annotations. The explanation of the algorithm and of its correctness arguments [10] needed around 40 pages of dense writing. So, it is not an easy task to incorporate the MLKit ideas into a new language.

The contribution of this paper is a simple region inference algorithm for *Safe*. It allows polymorphic recursion w.r.t. regions (region-polymorphic recursion, in the following). Hindley-Milner type inference is in general undecidable under polymorphic recursion, but when restricting to region-polymorphic recursion it becomes decidable. Our algorithm runs in $O(n)$ time in the worst case (being n as above) if region-polymorphic recursion is not inferred. If the latter appears, the algorithm needs $O(n^2)$ time in the worst case. Moreover, the first phase of the algorithm can be directly integrated in the usual Hindley-Milner type inference algorithm, just by considering regions as ordinary polymorphic type variables. The second phase involves very simple set operations and the computation of a fixpoint. Unlike [10], termination is always guaranteed without special provisions. There, they had to sacrifice principal types in order to ensure termination. Due to its simplicity, we believe that the algorithm can be easily reused in a different first-order functional language featuring Hindley-Milner types.

The plan of the paper is as follows: In Sec. 2 we summarize the language concepts and part of its big-step operational semantics. In Sec. 3 the region inference algorithm is presented in detail, including its correctness and cost. Section 4 shows some examples of region inference with region polymorphic recursion. Finally, Sec. 5 compares this work with other functional languages with memory management facilities.

2 Language Concepts and Inference Examples

2.1 Operational semantics

In Fig. 1 we show a simplified version of the *Safe* language without the destruction facilities but with explicit region arguments and region types. A program

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

$$\frac{j \leq k \quad fresh(p)}{E[r \mapsto j, \overline{a_i \mapsto v_i}^n] \vdash h, k, C\ \overline{a_i}^n @ r \Downarrow h \uplus [p \mapsto (j, C\ \overline{v_i}^n)], k, p}\ [Cons]$$

Fig. 2. Operational semantics of *Safe* expressions

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 $x_1 \cdots x_n$. 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 $c, c_i \ldots$ etc. Region arguments $r, r_i \ldots$ occur in function definitions and in function and constructor applications. They are containers used at runtime to pass region values around. Region values k are runtime numbers denoting actual regions in the region stack, and region types ρ are static annotations assigned to region variables and occuring in type declarations.

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 **region** is a collection of cells, not necessarily contiguous in memory. Regions are allocated/deallocated by following a stack discipline associated with function calls and returns. Each function call allocates a local working region, which is deallocated when the function returns.

In Fig. 2 we show those rules of the big-step operational semantics which are relevant with respect to regions. 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.

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 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 to construction cells w of the form $(j, C\ \overline{v_i}^n)$, meaning that the cell resides in region j. Actual region identifiers j are just natural numbers denoting the offset of the region from the bottom of the region stack. Formal regions appearing in a function body are either region variables r corresponding to formal arguments or the constant *self*, which represents the local working region. By $h \uplus [p \mapsto w]$ we denote the disjoint

union of heap h with the binding $[p \mapsto w]$. By $h\,|_k$ we denote the heap obtained by deleting from h those bindings living in regions greater than k.

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.

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 data structures 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. In rule *Cons* a fresh construction cell is allocated in the heap.

2.2 Region annotations

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. Before explaining the inference algorithm we show some illustrative examples.

Regions are essentially the parts of the heap where the data structures live. We will consider as a **data structure** (DS) 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 innermost lists. Each one of the latter constitute a DS living in a possibly different region from the outermost's one. However, since all the innermost lists have the same type, they will be forced to reside in the same region. 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 is $\mathit{partition} :: \forall\rho_1\rho_2\rho_3\rho_4.\mathit{Int} \to [\mathit{Int}]@\rho_1 \to \rho_2 \to \rho_3 \to \rho_4 \to ([\mathit{Int}]@\rho_2, [\mathit{Int}]@\rho_3)@\rho_4$. The algorithm splits the output in as many regions as possible. This gives more general types and allows the garbage to be deallocated sooner.

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)
```

void PushRegion ()	-- creates a top empty region
void PopRegion ()	-- removes the topmost region
cell ReserveCell ()	-- returns a fresh cell
void InsertCell (p, j)	-- inserts cell p into region j
void ReleaseCell (p)	-- releases cell p

Fig. 3. The interface of the *Safe* Memory Management System.

where the inferred types are as follows:

$$\begin{aligned} \mathit{makeTree} &:: \forall a\rho_1\rho_2.[a]@\rho_1 \to \rho_2 \to \mathit{Tree}\ a@\rho_2 \\ \mathit{inorder} &:: \forall a\rho_1\rho_2.\mathit{Tree}\ a@\rho_1 \to \rho_2 \to [a]@\rho_2 \\ \mathit{treesort} &:: \forall a\rho_1\rho_2.[a]@\rho_1 \to \rho_2 \to [a]@\rho_2 \end{aligned}$$

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`.

The region inference mechanism will not lead to rejecting programs. It always succeeds although, of course, it will not be able to detect all garbage. Section 3 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.

2.3 Region implementation

As we said above, the heap is implemented as a stack of regions. Each region is pushed initially empty, this action being associated with a *Safe* function invocation. During function execution new cells can be added to, or removed from, any active region as a consequence of constructor applications and destructive pattern matching. Upon function termination the whole topmost region is deallocated. In Fig. 3 we show the main interface between a running *Safe* program and the Memory Management System (MMS). It is written in Java since the code generated by the *Safe* compiler is Java bytecode. The MMS maintains a pool of fresh cells, so that 'allocating' and 'deallocating' a cell respectively mean removing it from, or adding it to the pool.

Notice that access to an arbitrary region is needed in *InsertCell*, whereas *ReleaseCell* is only provided with the cell pointer as an argument. We have implemented all the methods running in constant time by representing the regions and the pool as circular doubly-chained lists. Removing a region amounts to joining two circular lists, which can obviously be done in constant time. The region stack is represented by a static array of dynamic lists, so that constant time access to each region is provided. Fig. 4 shows a picture of the heap.

Tail recursive functions can very easily be detected at compile time so that a special translation for them would not push a new empty region at each invocation, but instead reuse the current topmost region. This translation (not yet

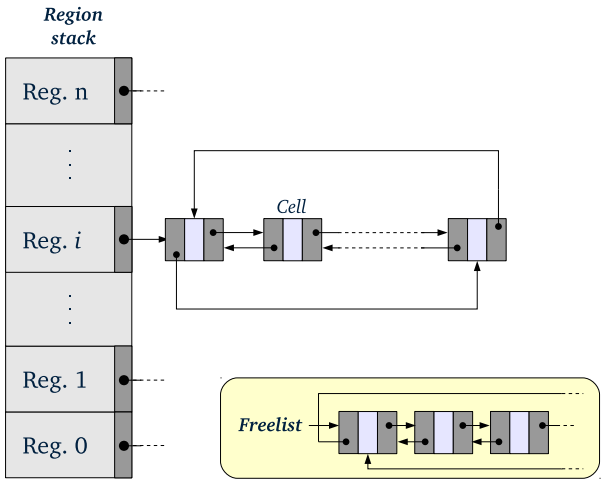

Fig. 4. A picture of the Safe Virtual Machine heap and fresh cells pool

implemented in our compiler) would not avoid consuming new cells at each invocation but at least would consume a constant stack space in the region stack. Consuming constant heap space in tail recursive functions is not feasible in general because any function invocation may freely access regions below the topmost one.

The *Safe* virtual machine has also a conventional stack where local variables are kept. The code generated for function invocation guarantees that tail recursive functions always consume constant stack space. In this respect, no special translation is needed.

3 The Region Inference Algorithm

The main correctness requirement to the region inference algorithm is that the annotated type of each function can be assigned to the corresponding annotated function in the type system defined in [7]. The main constraints posed by that system with respect to regions are reflected in the function and constructor typing rules, shown in Fig. 5.

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 for writing are *self* and the argument regions.

Notice that region-polymorphic recursion is allowed: inside the body e, different applications of f may use different regions. We use $gen(\sigma', \Gamma)$ and $tf \trianglelefteq \sigma$ to respectively denote (standard) generalization of a type with respect to type variables excluding region types, and instantiation of a polymorphic type.

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

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

Fig. 5. Typing rules for function definition and constructor application

```
decorProg :: Assumps -> Prog a -> (Assumps, Prog ExpTipo)
decorProg asInit (datas, defs, exp) =  (as',(datas', concat defs', exp'))
  where (as,datas')  = decorDecsData asInit datas
        groups       = groupBy sameName defs
        (as', defs') = mapAccumL decorAndGenOuterDefs as groups
        exp'         = decorAndGenMainExp as' exp
```

Fig. 6. A high-level view of the Hindley-Milner inference algorithm

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

$$[\,] : \forall a, \rho.\rho \to [a]@\rho$$
$$(:) : \forall a, \rho.a \to [a]@\rho \to \rho \to [a]@\rho$$
$$Empty : \forall a, \rho.\rho \to Tree\ a@\rho$$
$$Node : \forall a, \rho.Tree\ a@\rho \to a \to Tree\ a@\rho \to \rho \to Tree\ a@\rho$$

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

3.1 A high-level view of the algorithm

Figure 6 shows a high-level view of the Hindley-Milner (abbreviated HM in the following) type inference algorithm of the *Safe* compiler, written in Haskell, in which some parts have to do with region inference.

The first phase, `decorDecsData`, annotates the **data** declarations with region variables and infers the types of the data constructors. These are saved in the assumption environment `as`. 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 of 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. In Sec. 2.2 we have shown some examples of the result produced by this phase.

After this, the equations `defs` defining functions are grouped by function name, traversed, and their HM-types and regions inferred for each function (algorithm `decorAndGenOuterDefs`, see below), accumulating the inferred type in the assumption environment `as` in order to infer subsequent function definitions.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{decorAndGenOuterDefs}\ \Gamma\ \mathit{Defs} = (\Gamma \cup [f \mapsto t^+], \mathit{Defs}'')} \\
\quad \textbf{where}\ f & = \mathit{extractFunctionName}\ \mathit{Defs} \\
\quad\quad (\mathit{Defs}', \mathit{Eqs}, \mathit{Fresh}_{expl}, \overline{trec_j}^{\,p}) & = \mathit{decorAndGenEqs}\ \Gamma\ \mathit{Defs} \\
\quad\quad \theta_1 & = \mathit{solveEqs}\ \mathit{Eqs} \\
\quad\quad t & = \theta_1(\mathit{type}\ \mathit{Defs}') \\
\quad\quad (\theta_2, \overline{\varphi_j}^{\,p}) & = \mathit{handleRecCalls}\ t\ (\theta_1(\overline{trec_j}^{\,p})) \\
\quad\quad \theta & = \theta_2 \circ \theta_1 \\
\quad\quad R_{expl} & = \theta(\mathit{Fresh}_{expl}) \\
\quad\quad (\theta_{self}, t^+, \mathit{RegMap}) & = \mathit{inferRegions}\ t\ R_{expl}\ \overline{\varphi_j}^{\,p} \\
\quad\quad \mathit{Defs}'' & = \mathit{annotateDef}\ (\theta_{self} \circ \theta)\ \mathit{RegMap}
\end{array}
$$

Fig. 7. HM-type and region inference for a single function

Finally, the main expression `exp` of the program is inferred, and decorated by `decorAndGenMainExp` (not shown).

3.2 Region inference of function definitions

Figure 7 shows in Haskell-like pseudocode the HM-inference process for a single function consisting of a list *Defs* of equations. Let us call such function f.

We have a decoration phase *decorAndGenEqs* which generates fresh type and region type variables, and equations relating types that have to be unified, but delays all the unifications to a subsequent phase. Some of these equations correspond to the usual HM type inference, e.g. $a = [b] \to b$, but some other unify region type variables, e.g. $\rho_1 = \rho_2$. The decoration phase generates a set Fresh_{expl} of fresh region type variables assigned to the region arguments of constructor applications and (already inferred) function applications. This set will be needed in the second phase of region inference.

Unification equations are solved by *solveEqs* and *handleRecCalls*. The former solves all the equations in the usual HM style except those related to the recursive applications of f, which are solved in a special way by the latter: Hindley-Milner types of recursive applications are unified with the inferring function's type, while region type variables are not unified. This is due to the fact that the type trec_j of every application of f should be a fresh instance of the HM type t of the function with respect to the region types. Each region substitution φ_j reflects this fact by mapping the region type variables in t to those in trec_j. For instance, if the type inferred for a function after *solveEqs* is $[a]@\rho_1 \to b$ and there is a single recursive application with type $[a]@\rho_2 \to [c]@\rho_2$, the resulting substitution of *handleRecCalls* is $\theta = [b \mapsto [c]@\rho_1]$ with a region mapping $\varphi = [\rho_1 \mapsto \rho_2]$.

The next step is the application of the final substitution θ to the set Fresh_{expl} of explicit region types obtained above, obtaining the smaller set R_{expl}. Then, the second and final phase, *inferRegions*, of region inference is done. Its purpose is to detect how many explicit region arguments the (possibly recursive) function f must have, and to infer which region types must be assigned to the local working region *self*. This algorithm is depicted in Fig. 8 and explained in the next section. It delivers a substitution θ_{self} mapping some region type variables to the reserved type variable ρ_{self} assigned to the local region *self*, a map *RegMap*

$$
\begin{array}{l}
\mathit{inferRegions}\ t\ R_{expl}\ \overline{\varphi_j}^p = ([\rho \mapsto \rho_{self} \mid \rho \in R_{self}], \overline{t_i}^n \to \overline{\rho_k}^m \to t', [\overline{\rho_k \mapsto r_j}^m]) \\
\quad \textbf{where}\ \overline{t_i}^n \to t' \quad = t \\
\qquad\qquad R_{out} \quad = \mathit{regions}\ t' \\
\qquad\qquad R_{in} \quad = \mathit{regions}\ \overline{t_i}^n \\
\qquad\qquad R_{arg} \quad = R_{expl} \cap (R_{in} \cup R_{out}) \\
\qquad\qquad (R'_{arg}, R'_{expl}) = \mathit{computeRargFP}\ R_{in}\ R_{out}\ R_{arg}\ R_{expl}\ \overline{\varphi_j}^p \\
\qquad\qquad R_{self} \quad = R'_{expl} - (R_{out} \cup R_{in}) \\
\qquad\qquad \overline{\rho_k}^m \quad = R'_{arg} \\
\\
\mathit{computeRargFP}\ R_{in}\ R_{out}\ R_{arg}\ R_{expl}\ \overline{\varphi_j}^p \\
\qquad\qquad \mid R_{arg} == R'_{arg} = (R'_{arg}, R'_{expl}) \\
\qquad\qquad \mid \textbf{otherwise} \quad = \mathit{computeRargFP}\ R_{in}\ R_{out}\ R'_{arg}\ R'_{expl}\ \overline{\varphi_j}^p \\
\quad \textbf{where}\ R'_{expl} = R_{expl} \cup \bigcup_{j=1}^{p}\ \{\varphi_j(\rho) \mid \rho \in R_{arg}\} \\
\qquad\qquad R'_{arg}\ = R'_{expl} \cap (R_{in} \cup R_{out})
\end{array}
$$

Fig. 8. Second phase of the region inference algorithm

mapping some other region type variables to region arguments, and the extended function type t^+. The last step adds these region arguments to the definition of f. The function's body is traversed again and the above substitutions and mappings are used to incorporate the appropriate region arguments to all the expressions, including the recursive applications of f. Additionally, the final substitution $\theta_{self} \circ \theta$ is applied to all the types.

3.3 Second phase of region inference

Algorithm *inferRegions* of Fig. 8 receives the type t obtained for the function f by the HM inference, the set R_{expl} of initial explicit region types, and the list of substitutions $\overline{\varphi_j}^p$ associated with the recursive applications of f. First, it computes the sets R_{in} and R_{out} of region type variables of respectively the argument and the result parts of t. Let ρ_{self} be an additional fresh type variable for *self*.

Given these three sets, the region inference problem can be specified as finding three sets R'_{expl}, R'_{arg} and R_{self}, respectively standing for the sets of final explicit region types, of region types needed as additional arguments of f, and 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. Every recursive application of f is typeable

The first one expresses that everything built by f's body must be in regions in scope. The second and third ones state 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 can be further formalised by requiring that f's type, extended with the region arguments in R'_{arg}, can produce type instances for typing all the recursive applications of f, each one extended with as many

region arguments as the cardinal of R'_{arg}. So, in order to satisfy restriction (4) one must provide a decoration of each recursive application of f with appropriate region arguments, of region types belonging either to R_{self} or to R'_{arg}, as restriction (1) requires.

In the extended version of this paper [9] we show that any sets R'_{expl}, R'_{arg} and R_{self} satisfying these restrictions produce a version of f which admits a type in the type system. The correctness of the type system with respect to the semantics was established in [7]. There, we proved that dangling pointers arising from region deallocation or destructive pattern matching are never accessed by a well-typed program.

Notice that an algorithm choosing any $R'_{arg} \supseteq R'_{expl}$ 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 two senses. On the one hand, we want R'_{arg} to be as small as possible, so that only those regions where data are built are given as arguments. On the other hand, we want R_{self} to be as big as possible, so that the maximum amount of memory is deallocated at function termination.

3.4 The kernel of the algorithm

Our algorithm initially computes $R_{arg} = R_{expl} \cap (R_{in} \cup R_{out})$, by using the set R_{expl} of initial explicit region types. Then, it starts a fixpoint algorithm *computeRargFP* (see Fig. 8) trying to get the type of f's recursive applications as instances of the type of f extended with the current set R_{arg} of arguments. It may happen that the set of explicit regions R'_{expl} may grow while considering different applications (see the examples in Sec. 4). Adding more explicit variables to one application will influence the type of the applications already inferred. As R'_{arg} depends on R'_{expl}, it may also grow. So, a fixpoint is used in order to obtain the final R'_{arg} and R'_{expl} from the initial ones. Due to our solution above, R'_{arg} cannot grow greater than $R_{in} \cup R_{out}$, so termination of the fixpoint is guaranteed. Once obtained the final R'_{arg} and R'_{expl}, the set R_{self} is computed as $R_{self} = R'_{expl} - (R_{in} \cup R_{out})$. Notice that $R'_{arg} = R'_{expl} \cap (R_{in} \cup R_{out})$ is an invariant of the algorithm.

We show below that these choices maximise the data allocated to the *self* region, which in turn maximises the amount of memory reclaimed at runtime when the corresponding function call finishes. With respect to the remaining DSs not being inferred to live in *self*, they will be allocated to the regions which are parameters to the function being called. It is the *caller* function's responsibility to determine where to put these DSs by passing the suitable arguments. Since the caller function is also inferred by the algorithm, the parameter assignment is done in such a way that the data allocated in the caller's *self* region is also maximised. From a global point of view, every cell not being created in the current topmost region (i.e. the region bound to the *self* identifier) will be created in the highest possible region and hence, will be deallocated at the earliest time allowed by the type system.

3.5 Correctness, optimality and efficiency

First we prove that the proposed solution satisfies the above specification:

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

The three immediately follow by set algebra. We will show now that it is optimal: let us assume a different solution $\hat{R}_{self}, \hat{R}_{expl}, \hat{R}_{arg}$ satisfying the above restrictions. Notice that $R_{expl} \subseteq R'_{expl}$ by construction. Without loss of generality we can rename those variables in $\hat{R}_{expl}$ which decorate copy expressions, constructor applications and function calls differerent from f, so that such decorations coincide with those in R'_{expl}. After such renaming $R_{expl} \subseteq \hat{R}_{expl}$. We can also rename the argument regions in recursive calls to f that also appear in R'_{expl}. For example, assume there is a recursive call decorated by R'_{expl} as $f :: \overline{t_i}^n \to \rho'_1 \to \rho'_2 \to t$. If that recursive call was decorated by $\hat{R}_{expl}$ as $f :: \overline{t'_i}^n \to \hat{\rho}_1 \to \hat{\rho}_2 \to \hat{\rho}_3 \to t'$, then $\hat{\rho}_1$ would be renamed as ρ'_1 and $\hat{\rho}_2$ as ρ'_2.

We must show that $\hat{R}_{self} \subseteq R_{self}$ and $R'_{arg} \subseteq \hat{R}_{arg}$. Let us assume $\rho \in R'_{arg}$. By definition of R'_{arg}, $\rho \in R'_{expl}$ and $\rho \in R_{in} \cup R_{out}$. By (3), $\rho \in R_{in} \cup R_{out}$ implies that $\rho \notin \hat{R}_{self}$. Now we distinguish two cases:

$\rho \in R_{expl}$ As $R_{expl} \subseteq \hat{R}_{expl}$, then $\rho \in \hat{R}_{expl}$. By (1) $\rho \in \hat{R}_{arg}$.

$\rho \in R'_{expl} - R_{expl}$ If $\rho \in \hat{R}_{expl}$, then by $\rho \in \hat{R}_{arg}$. Otherwise, R'_{expl} contains more explicit variables which are also arguments of f than $\hat{R}_{expl}$. This case is not possible because R'_{expl} is the least fixpoint of function *computeRargFP* by construction. By (4), $\hat{R}_{expl}$ is also a fixpoint of *computeRargFP*; otherwise, the recursive calls would not be typeable.

Consequently, $\rho \in R'_{arg}$. So, R_{arg} is as small as possible. By constraints (2) and (1), then R_{self} is as big as possible. Regarding regions, there are no principal types in our system, since other correct types bigger than our minimal type could not be obtained as an instance of it.

Our sets are implemented as balanced trees, and operations such as $\cup$, $\cap$, and '$-$' are done in a time in $\Theta(n + m)$, being n and m the cardinalities of the respective sets, so each iteration of the fixpoint algorithm is linear with the number of region type variables occurring in a function body. As it is done in [10], considering as the term size n the sum of the sizes of the abstract syntax tree and of the HM type annotations, each iteration needs time linear with this size. If several iterations are needed, these cannot be more than the number of region type variables in $R_{in} \cup R_{out}$. This gives us $O(n^2)$ cost in the worst case.

4 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 $Fresh_{expl} = \{\rho_2, \ldots, \rho_8\}$. We show them as annotations in the program just in order to better explain the example:

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

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 the initial 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, $R_{arg} = \{\rho_2, \rho_3, \rho_4\}$. Now we shall compare the type of the definition (augmented with the variables of R_{arg}) and the type used in the recursive call, where the tuple (ls, gs) is assumed to live in the region ρ_9.

$$\begin{array}{rl} \text{Definition:} & Int \to [Int]@\rho_1 \to \rho_2 \to \rho_3 \to \rho_4 \to ([Int]@\rho_2, [Int]@\rho_3)@\rho_4 \\ \text{Rec. call:} & Int \to [Int]@\rho_1 \to \rho_2 \to \rho_3 \to \rho_9 \to ([Int]@\rho_2, [Int]@\rho_3)@\rho_9 \end{array}$$

We obtain the region substitution $\varphi = [\rho_1 \mapsto \rho_1, \rho_2 \mapsto \rho_2, \rho_3 \mapsto \rho_3, \rho_4 \mapsto \rho_9]$. As a consequence, the variable ρ_9 is made explicit, so $R_{expl} = \{\rho_2, \rho_3, \rho_4, \rho_9\}$. The set R_{arg} does not change and hence the fixpoint has been computed. We get $R_{self} = \{\rho_9\}$ and the program is annotated as follows:

$$\begin{array}{ll} \mathit{partition} :: Int \to [Int]@\rho_1 \to \rho_2 \to \rho_3 \to \rho_4 \to ([Int]@\rho_2, [Int]@\rho_3)@\rho_4 & \\ \mathit{partition}\ y\ [\,]\ @\ r_2\ r_3\ r_4 = ([\,]@r_2, [\,]@r_3)@r_4 & \\ \mathit{partition}\ y\ (x : xs)\ @\ r_2\ r_3\ r_4 & \mid x \leq y = ((x : ls)@r_2, gs)@r_4 \\ & \mid x > y = (ls, (x : gs)@r_3)@r_4 \\ & \textbf{where}\ (ls, gs) = \mathit{partition}\ y\ xs\ @\ r_2\ r_3\ \mathit{self} \end{array}$$

Notice that the tuple resulting from the recursive call to *partition* is located in the working region. Without region-polymorphic recursion this tuple would have to be stored in the output region r_4, requiring $O(n)$ space in a caller region.

Another example is the dynamic programming approach to computing binomial coefficients by using the Pascal's triangle. We start from the unit list [1], which corresponds to the 0-th row of the triangle. If $[x_0, x_1, \ldots, x_{i-1}, x_i]$ are the elements located on the i-th row, then the elements of the $i+1$-th row are given by the list $[x_0 + x_1, x_1 + x_2, \ldots, x_{i-1} + x_i, x_i]$. The binomial coefficient $\binom{n}{m}$ can be obtained from the m-th element in the n-th row of the Pascal's triangle. Function *sumList*, computes the $i+1$-th row of the triangle from its i-th row:

$$\begin{array}{ll} \mathit{sumList}\ (x : [\,]) = (x : [\,] :: \rho_2) :: \rho_3 & \\ \mathit{sumList}\ (x : xs) = (x + y : \mathit{sumList}\ xs) :: \rho_4 & \textbf{where}\ (y : \_) = xs \end{array}$$

In the definition above we just show those region variables belonging to $Fresh_{expl}$. Let us assume that after unification the input list has type $[Int]@\rho_1$. In addition, the variables ρ_2, ρ_3 and ρ_4 are unified, so $R_{expl} = \{\rho_2\}$ and the inferred type (without region parameters) for *sumList* is $[Int]@\rho_1 \to [Int]@\rho_2$. Hence we get $R_{in} = \{\rho_1\}$, $R_{out} = \{\rho_2\}$ and $R_{arg} = \{\rho_2\}$. We extend the signature of *sumList* to $[Int]@\rho_1 \to \rho_2 \to [Int]@\rho_2$.

Next we analyse the recursive call. Since the inferred type for xs is $[Int]@\rho_1$ and the type of the recursive call's result is $[Int]@\rho_2$, the type for *sumList* in this call is $[Int]@\rho_1 \to [Int]@\rho_2$. By pairing with the type of *sumList* in the definition, we get that the additional argument needed in the recursive call also has type ρ_2. Therefore, ρ_2 is added to the set of explicit variables R_{expl}. Since it was already in this set, R_{expl} stays the same as the one calculated previously and hence R_{arg} also does, so the fixpoint has been reached. Finally we obtain $R_{self} = \emptyset$ and the function is annotated as follows:

$$
\begin{array}{l}
\mathit{sumList} \quad :: [Int]@\rho_1 \to \rho_2 \to [Int]@\rho_2 \\
\mathit{sumList}\ (x : [\,])\ @\ r = (x : ([\,]\ @\ r))@\ r \\
\mathit{sumList}\ (x : xs)\ @\ r = (x + y : \mathit{sumList}\ xs\ @\ r)\ @\ r \quad \textbf{where}\ (y : \_) = xs
\end{array}
$$

Function *pascal* iterates over the initial list in order to get the desired row. Below we show the region variables generated in constructor applications and in non-recursive function applications, just *after* type unification:

$$
\begin{array}{l}
\mathit{pascal}\ 0 = (1 : [\,] :: \rho_1) :: \rho_1 \\
\mathit{pascal}\ n = (1 : \mathit{sumList}\ (\mathit{pascal}\ (n-1))) :: \rho_1
\end{array}
$$

The type inferred for *pascal* is $Int \to [Int]@\rho_1$. Hence $R_{in} = \emptyset$, $R_{out} = \{\rho_1\}$ and $R_{expl} = \{\rho_1\}$, which gives us an initial $R_{arg} = \{\rho_1\}$. The type signature for *pascal* changes accordingly to $Int \to \rho_1 \to [Int]@\rho_1$. Let us assume that the result of the recursive call to *pascal* has type $[Int]@\rho_2$. Therefore, the type of this function in the recursive call is $Int \to \rho_2 \to [Int]@\rho_2$. Since ρ_2 is now made explicit, it is added to R_{expl}, which now contains the region variables $\{\rho_1, \rho_2\}$. However, R_{arg} stays the same and hence the fixpoint has been reached. Finally, we get $R_{self} = \{\rho_2\}$ and the program is annotated as follows:

$$
\begin{array}{l}
\mathit{pascal} \quad :: Int \to \rho_1 \to [Int]@\rho_1 \\
\mathit{pascal}\ 0\ @\ r = (1 : [\,]\ @\ r)@\ r \\
\mathit{pascal}\ n\ @\ r = (1 : \mathit{sumList}\ (\mathit{pascal}\ (n-1)\ @\ \mathit{self})\ @\ r)\ @\ r
\end{array}
$$

The resulting list from the recursive call to *pascal* will be destroyed once the calling function finishes. Hence a function call *pascal* n has a cost of $O(n)$ in space. Without region-polymorphic recursion the result of every recursive call would be built in the output region r, which would imply $O(n^2)$ heap cost.

5 Related Work and Conclusions

The pioneer work on region inference is that of M. Tofte, J.-P. Talpin and their colleagues on the MLKit compiler [14, 10] (in what follows, TT). Their language is higher-order and they also support polymorphic recursion in region arguments. The TT algorithm has two phases, respectively called S and R. The S-algorithm just generates fresh region variables for values and introduces the lexical scope of the regions by using a **letregion** construct. The R-algorithm is responsible for assigning types to recursive functions. It deals with region-polymorphic recursion and also computes a fixpoint. The total cost is in $O(n^4)$. The meaning of a typed expression **letregion** ρ **in** $e : \mu$ is that region ρ does not occur free in type

μ, 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 make local *all* the regions not needed in the final value. One problem reported in [12] is that most of the regions inferred in the first versions of the algorithm contained a single value so that region management produced a big overhead at runtime. Later, they added a new analysis to collapse all these regions into a single one local to the invocation (allocated in the stack). So, having a single local region *self* per function invocation does not seem to us to be a big drawback if function bodies are small enough. We believe that region-polymorphic recursion has a much bigger impact in avoiding memory leaks than multiplicity of local regions. So, we claim that the results of our algorithm are comparable to those of TT for first-order programs.

A radical deviation from these approaches is [4] 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 language is first-order. The inference algorithm [6] 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, it can be deduced that this cost could grow more than quadratically 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 [3] in which type-safe primitives are defined for creating, accessing and destroying regions. These are not restricted to have nested lifetimes. Programs are written and manually typed 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 functions. As a consequence, the inference algorithms are more complex and costly. 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 where it lives. Allocation and destruction are not necessarily nested, and our type system protects the programmer against misuses of this feature. Since allocation is implicit, the price of this flexibility is the explicit deallocation of cells.

In the near future we plan to extend *Safe* to support higher-order functions and mutually recursive data structures. We expect high difficulties in other aspects of the language such as extending dangling pointers safety analyses or

memory bounds inference, but not so many to extend the region inference algorithm presented here. It is still open whether we could achieve a cost better than the $O(n^4)$ got by Tofte and Talpin.

References

1. A. Aiken, M. Fähndrich, and R. Levien. Better static memory management: improving region-based analysis of higher-order languages. In *ACM SIGPLAN Conference on Programming Languages Design and Implementation, PLDI'95*, pages 174–185. ACM Press, 1995.
2. L. Birkedal, M. Tofte, and M. Vejlstrup. From region inference to von Neumann machines via region representation inference. In *23rd ACM Symposium on Principles of Programming Languages, POPL'96*, pages 171–183. ACM Press, 1996.
3. M. Fluet, G. Morrisett, and A. Ahmed. Linear regions are all you need. In *15th European Symposium on Programming, ESOP 2006. LNCS, vol. 3924*, pages 7–21. Springer, 2006.
4. F. Henglein, H. Makholm, and H. Niss. A direct approach to control-flow sensitive region-based memory management. In *3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, PPDP'01*, pages 175–186. ACM Press, 2001.
5. M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *30th ACM Symposium on Principles of Programming Languages, POPL'03*, pages 185–197. ACM Press, 2003.
6. H. Makholm. A language-independent framework for region inference. Ph.D thesis, Univ. of Copenhagen, Dep. of Computer Science, Denmark, 2003.
7. M. Montenegro, R. Peña, and C. Segura. A type system for safe memory management and its proof of correctness. In *10th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming, PPDP'08*, pages 152–162, 2008.
8. M. Montenegro, R. Peña, and C. Segura. An inference algorithm for guaranteeing safe destruction. In *18th International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR'08, Lecture Notes in Computer Science 5438*, pages 135–151, 2009.
9. M. Montenegro, R. Peña, and C. Segura. A simple region inference algorithm for a first-order functional language (extended version). Technical report, SIC-5-09. Dpto. de Sist. Informáticos y Computación. UCM, 2009. Available at http://federwin.sip.ucm.es/sic/investigacion/publicaciones/informes-tecnicos/.
10. M. Tofte and L. Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):724–767, 1998.
11. M. Tofte, L. Birkedal, M. Elsman, N. Hallenberg, T. H. Olesen, and P. Sestoft. Programming with regions in the MLKit (revised for version 4.3.0). Technical report, IT University of Copenhagen, Denmark, 2006.
12. M. Tofte and N. Hallenberg. Region-Based Memory Management in Perspective. In *Invited talk Space 2001 Work., London*, pages 1–8. Imperial College, Jan. 2001.
13. M. Tofte and J.-P. Talpin. Implementing the call-by-value lambda-calculus using a stack of regions. In *21st ACM Symposium on Principles of Programming Languages, POPL'94*, pages 188–201, Jan. 1994.
14. M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.