A Type System for Safe Memory Management and its Proof of Correctness [*]

Manuel Montenegro   Ricardo Peña   Clara Segura

Dpto. de Sistemas Informáticos y Computación
Univ. Complutense de Madrid
C/ Prof. José García Santesmases s/n
28040, Madrid, Spain
montenegro@fdi.ucm.es, {ricardo,csegura}@sip.ucm.es

Abstract

We present a destruction-aware type system for the functional language *Safe*, which 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 equipped with several analyses and inference algorithms so that regions, sharing information and types are automatically inferred by the compiler. Here, we concentrate on the correctness of the type system with respect to the operational semantics of the language. In particular, we prove that, in spite of sharing and of the use of implicit and explicit memory deallocation operations, all well-typed programs will be free of dangling pointers at runtime.

The paper ends up with some examples of well-typed programs.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Applicative (functional) languages; D.3.3 [*Language constructs and features*]: Dynamic storage management; D.3.4 [*Processors*]: Memory management (garbage collection), Compilers; F.3.2 [*Semantics of Programming Languages*]: Program analysis, Operational Semantics

General Terms Languages, Security.

Keywords Type systems, safe memory deallocation.

1. Introduction

Most functional languages abstract the programmer from the memory management done by programs at run time. The runtime support system usually allocates fresh heap memory while program expressions are being evaluated as long as there is enough free memory available. Should the memory be exhausted, the garbage collector will copy/mark the live part of the heap and will consider the rest as free. This normally implies the suspension of program execution for some time. Occasionally, not enough free memory has been recovered and the program simply aborts. This model is acceptable in most situations, being its main advantage that programmers are not bothered, and programs are not obscured, with low level details about memory management. But, in some other contexts, this scheme may not be acceptable:

1. The time delay introduced by garbage collection prevents the program from providing an answer in a required reaction time.
2. Memory exhaustion abortion may provoke unacceptable personal or economic damage to program users.
3. The programmer wishes to reason about memory consumption.

On the other hand, many imperative languages offer low level mechanisms to allocate and free heap memory. These mechanisms give programmers a complete control over memory usage but are very error prone. Well known problems are dangling references, undesired sharing with complex side effects, and polluting memory with garbage.

In our functional language *Safe*, we have chosen a semi-explicit approach to memory control in which programmers may cooperate with the memory management system by providing some information about the intended use of data structures (in what follows, abbreviated as DS). For instance, they may indicate that some particular DS will not be needed in the future and that it should be destroyed by the runtime system and its memory recovered. Programmers may also launch copies of a DS and control the degree of sharing between DSs. In order to use these facilities in safe way, we have developed a type system which guarantees that dangling pointers will never arise at runtime in the living heap. The *Safe* language and a sharing analysis for it were published in [18]. In [10] a type inference algorithm is presented and proved correct with respect to the type system explained in this paper. A region inference algorithm for *Safe* is described in [12].

The proposed approach overcomes the above mentioned shortcomings: (1) A garbage collector is not needed because the heap is structured into disjoint *regions* which are dynamically allocated and deallocated; (2) as we will see below, we will be able to reason about memory consumption. It will even be possible to show that an algorithm runs in constant heap space, independently of input size; and (3), as an ultimate goal regions will allow us to statically infer sizes for them and eventually an upper bound to the memory consumed by the program.

The language is targeted to mobile code applications with limited resources in a Proof Carrying Code framework [14, 15]. The

[*] Work supported by the projects TIN2004-07943-C04, S-0505/TIC/0407 (PROMESAS) and the MEC FPU grant AP2006-02154.

final aim is to endow programs with formal certificates proving the above properties. This aspect, as well as region size inference, are however beyond the scope of the current paper.

The plan of the paper is as follows; In Section 2 we informally introduce and motivate the language features. Section 3 formally defines its operational semantics. The kernel of the paper are sections 4 and 5 where respectively the destruction-aware type system is presented and proved correct. By lack of space, the detailed proofs are included in a separate technical report [13]. In Section 6 examples of successful type derivations are shown. In Section 7 we present related work with respect both to region-based memory management and to heap cells reuse. Finally, Section 8 concludes.

2. Summary of *Safe*

Safe is a first-order polymorphic functional language similar to (first-order) Haskell or ML with some facilities to manage memory. The memory model is based on heap regions where data structures are built. However, in *Full-Safe* in which programs are written, regions are implicit. These are inferred when *Full-Safe* is desugared into *Core-Safe*. As all the analyses mentioned in this paper [18] happen at *Core-Safe* level, later in this section we will describe it in detail.

The allocation and deallocation of regions is bound to function calls: a *working region* (called *self*) is allocated when entering the call and deallocated when exiting it. Inside the function, data structures may be built but they can also be destroyed by using a destructive pattern matching denoted by ! or a **case!** expression, which deallocates the cell corresponding to the outermost constructor. Using recursion the recursive spine of the whole data structure may be deallocated. We say that it is *condemned*. As an example, we show an append function destroying the first list's spine, while keeping its elements in order to build the result:

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

As a consequence, the concatenation needs constant heap space, while the usual version needs linear heap space. The fact that the first list is lost is reflected in the type of the function:
`concatD :: [a]! -> [a] -> [a]`.

The data structures which are not part of the function's result are built in the local working region, which we call *self*, and they die when the function terminates. As an example we show a destructive version of the treesort algorithm:

```
treesortD :: [Int]! -> [Int]
treesortD xs = inorder (mkTreeD xs)
```

First, the original list `xs` is used to build a search tree by applying function `mkTreeD` (defined below). This tree is then traversed in inorder to produce the sorted list. The tree is not part of the result of the function, so it will be built in the working region and will die when the `treesortD` function returns (in *Core-Safe* where regions are explicit this will be apparent). The original list is destroyed and the destructive appending function is used in the traversal so that constant heap space is consumed.

Function `mkTreeD` inserts each element of the list in the binary search tree.

```
mkTreeD :: [Int]! -> BSTree Int
mkTreeD []!      = Empty
mkTreeD (x:xs)!  = insertD x (mkTreeD xs)
```

The function `insertD` is the destructive version of insertion in a binary search tree. Then `mkTreeD` exactly consumes in the heap the space occupied by the list. The nondestructive version of this function would consume in the worst case quadratic heap space.

$$
\begin{array}{lcll}
\mathit{prog} & \rightarrow & \mathit{dec}_1; \ldots; \mathit{dec}_n; e & \\
\mathit{dec} & \rightarrow & f\ \overline{x_i}^n\ @\ \overline{r_j}^l = e & \{\text{recursive, polymorphic function}\} \\
e & \rightarrow & a & \{\text{atom: literal } c \text{ or variable } x\} \\
 & & \mid x@r & \{\text{copy}\} \\
 & & \mid x! & \{\text{reuse}\} \\
 & & \mid f\ \overline{a_i}^n\ @\ \overline{r_j}^l & \{\text{function application}\} \\
 & & \mid \textbf{let}\ x_1 = \mathit{be}\ \textbf{in}\ e & \{\text{non-recursive, monomorphic}\} \\
 & & \mid \textbf{case}\ x\ \textbf{of}\ \overline{\mathit{alt}_i}^n & \{\text{read-only case}\} \\
 & & \mid \textbf{case!}\ x\ \textbf{of}\ \overline{\mathit{alt}_i}^n & \{\text{destructive case}\} \\
\mathit{alt} & \rightarrow & C\ \overline{x_i}^n \rightarrow e & \\
\mathit{be} & \rightarrow & C\ \overline{a_i}^n\ @\ r & \{\text{constructor application}\} \\
 & & \mid e &
\end{array}
$$

Figure 1. *Core-Safe* language definition

```
insertD :: Int -> BSTree Int! -> BSTree Int
insertD x Empty! = Node Empty x Empty
insertD x (Node lt y rt)!
            | x == y = Node lt! y  rt!
            | x > y  = Node lt! y (insertD x rt)
            | x < y  = Node (insertD x lt) y rt!
```

Notice in the first guard, that the cell just destroyed must be built again. When a data structure is condemned its recursive children may subsequently be destroyed or they may be reused as part of the result of the function. We denote the latter with a !, as shown in this function `insertD`. This is due to safety reasons: a condemned data structure cannot be returned as the result of a function, as it potentially may contain dangling pointers. Reusing turns a condemned data structure into a safe one. The original reference is not accessible any more. The type system shown in this paper copes with all these features to avoid dangling pointers. So, in the example `lt` and `rt` are condemned and they must be reused in order to be part of the result.

Data structures may also be copied using @ notation. Only the recursive spine of the structure is copied, while the elements are shared with the old one. This is useful when we want non-destructive versions of functions based on the destructive ones. For example, we can define `treesort xs = (treesortD xs@)`.

In Fig. 1 we show the syntax of *Core-Safe*. A program $prog$ is a sequence of possibly recursive polymorphic function definitions followed by a main expression e, calling them, whose value is the program result. The abbreviation $\overline{x_i}^n$ stands for $x_1 \cdots x_n$. Destructive pattern matching is desugared into **case!** expressions. Constructions are only allowed in **let** bindings, and atoms are used in function applications, **case/case!** discriminant, copy and reuse. Regions are explicit in constructor application and the copy expression. Function definitions have additional parameters $\overline{r_j}^l$ where data structures may be built. In the right hand side expression only the r_j and its working region *self* may be used. functional types include region parameter types.

Polymorphic algebraic data types are defined through **data** declarations. Algebraic types declarations have additional parameters indicating the regions where the constructed values of that type are allocated. Region inference adds region arguments to constructors forcing the restriction that recursive substructures must live in the same region as its parent. For example, trees are represented as follows:

```
data BSTree a @ rho
        = Empty@rho
        | Node (BSTree a@rho) a (BSTree a@rho) @ rho
```

The recursive occurrences of the type being defined in a **data** declaration must be identical to the left-hand side (polymorphic recursion is not allowed for the moment).

There may be several region parameters when nested types are used: different components of the data structure may live in different regions. In that case the last region variable is the *outermost region* where the constructed values of this type are allocated. In the following example

```
data T a b @ rho1 rho2
        = C1 ([a] @ rho1) @ rho2
        | C2 b @ rho2
```

`rho2` is where the constructed values of type T are allocated, while `rho1` is where the list of a `C1` value is allocated.

Function `splitD` is an example with several output regions. In order to save space we show here a semi-desugared version with explicit regions. Notice that the resulting tuple and its components may live in different regions:

```
splitD :: Int -> [a]!@rho2 -> rho1 -> rho2
       -> rho3 -> ([a]@rho1, [a]@rho2)@rho3

splitD 0 zs!      @ r1 r2 r3 = ([]@r1, zs!)@r3
splitD n []!      @ r1 r2 r3 = ([]@r1, []@r2)@r3
splitD n (y:ys)! @ r1 r2 r3 = ((y:ys1)@r1, ys2)@r3
      where (ys1, ys2) = splitD (n-1) ys @r1 r2 r3
```

3. Operational Semantics

In Fig. 2 we show the big-step operational semantics of the core language expressions. We use $v, v_i, \ldots$ to denote either heap pointers or basic constants, and $p, p_i, q, \ldots$ to denote heap pointers, i.e. the syntax of normal form values is:

$$v \rightarrow c \mid p \qquad \text{literal } c,\;\; p \in dom\ h$$

We use $a, a_i, \ldots$ to denote either program variables or basic constants (atoms). 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 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. 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 $h\ |_k$ we denote the heap obtained by deleting from h those bindings living in regions greater than k.

The semantics of a program $d_1; \ldots; d_n; e$ is the semantics of the main expression e in an environment Σ containing all the functions declarations $d_1, \ldots, d_n$.

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. Some restricted type information is available in our runtime system so that this function can be implemented in a generic way. The pointers in non recursive positions of all the copied cells are kept identical in the new cells. This implies that both DSs may share some sub-structures.

In the rule Var_3 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 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.

By $fv(e)$ we denote the set of free variables of expression e, excluding function names and region variables, and by $dom(h)$ the set $\{p \mid [p \mapsto w] \in h\}$.

In [11] we present a virtual machine and a translation of *Safe* expressions implementing this semantics, and show that all memory management actions (allocation and deallocation of both cells and regions) are done in constant time. A region is created empty in rule App and it may grow and shrink during its lifetime due to the evaluation of constructions and of **case**! expressions. The support function $copy$ runs in time linear with the size of the DS being copied. We remind that a DS in *Safe* is defined to consist only of the cells reached by following the recursive arguments of the constructors. A remarkable feature of our compilation and virtual machine is that tail recursion is executed in constant stack space. See [11] for more details.

4. Safe Type System

In this section we describe a polymorphic type system with algebraic data types for programming in a safe way when using the destruction facilities offered by the language. The syntax of type expressions is shown in Fig. 3. As the language is first-order, we distinguish between functional, tf, and non-functional types, t, r. Non-functional algebraic types may be safe types s, condemned types d or in-danger types r. In-danger and condemned types are respectively distinguished by a # or ! annotation. In-danger types arise as an intermediate step during typing and are useful to control the side-effects of the destructions. But notice that the types of functions only include either safe or condemned types. The intended semantics of these types is the following:

- ***Safe* types (s):** A DS of this type can be read, copied or used to build other DSs. They cannot be destroyed or reused by using the symbol !. The predicate $safe?$ tells us whether a type is safe.
- ***Condemned* types (d):** It is a DS directly involved in a **case**! action. Its recursive descendants will inherit the same condemned type. They cannot be used to build other DSs, but they can be read or copied before being destroyed. They can also be reused once. The predicate $cmd?$ is true for these types.
- ***In-danger* types (r):** This is a DS sharing a recursive desdendant of a condemned DS, so potentially it can contain dangling pointers. The predicate $danger?$ is true for these types. The predicate $unsafe?$ is true for condemned and in-danger types. Function $danger(s)$ denotes the in-danger version of s.

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

$$E[x \mapsto v] \vdash h, k, x \Downarrow h, k, v \quad [\mathit{Var}_1]$$

$$\frac{j \leq k \quad (h', p') = \mathit{copy}(h, p, j)}{E[x \mapsto p, r \mapsto j] \vdash h, k, x@r \Downarrow h', k, p'} \; [\mathit{Var}_2]$$

$$\frac{\mathit{fresh}(q)}{E[x \mapsto p] \vdash h \uplus [p \mapsto w], k, x! \Downarrow h \uplus [q \mapsto w], k, q} \; [\mathit{Var}_3]$$

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

$$\frac{E \vdash h, k, e_1 \Downarrow h', k', v_1 \quad E \cup [x_1 \mapsto v_1] \vdash h', k', e_2 \Downarrow h'', k'', v}{E \vdash h, k, \mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2 \Downarrow h'', k'', v} \; [\mathit{Let}_1]$$

$$\frac{j \leq k \quad \mathit{fresh}(p) \quad E \cup [x_1 \mapsto p] \vdash h \uplus [p \mapsto (j, C\ \overline{v_i}^n)], k, e_2 \Downarrow h', k', v}{E[r \mapsto j, \overline{a_i \mapsto v_i}^n] \vdash h, k, \mathbf{let}\ x_1 = C\ \overline{a_i}^n @r\ \mathbf{in}\ e_2 \Downarrow h', k', v} \; [\mathit{Let}_2]$$

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

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

Figure 2. Operational semantics of *Safe* expressions

$$\begin{array}{lcll}
\tau & \rightarrow & t & \{\text{external}\} \\
 & & \mid r & \{\text{in-danger}\} \\
 & & \mid \sigma & \{\text{polymorphic function}\} \\
 & & \mid \rho & \{\text{region}\} \\
t & \rightarrow & s & \{\text{safe}\} \\
 & & \mid d & \{\text{condemned}\} \\
s & \rightarrow & T\ \overline{s}@\overline{\rho}^m & \\
 & & \mid b & \\
d & \rightarrow & T\ \overline{t}!@\overline{\rho}^m &
\end{array}
\qquad
\begin{array}{lcll}
r & \rightarrow & T\ \overline{s}\#@\overline{\rho}^m & \\
b & \rightarrow & a & \{\text{variable}\} \\
 & & \mid B & \{\text{basic}\} \\
\mathit{tf} & \rightarrow & \overline{t_i}^n \rightarrow \overline{\rho}^l \rightarrow T\ \overline{s}@\overline{\rho}^m & \{\text{function}\} \\
 & & \mid \overline{t_i}^n \rightarrow b & \\
 & & \mid \overline{s_i}^n \rightarrow \rho \rightarrow T\ \overline{s}@\overline{\rho}^m & \{\text{constructor}\} \\
\sigma & \rightarrow & \forall a.\sigma & \\
 & & \mid \forall \rho.\sigma & \\
 & & \mid \mathit{tf} &
\end{array}$$

Figure 3. Type expressions

In order to illustrate the meanings of these types, let us consider the following *Core-Safe* definition:

$$\begin{array}{lcl}
f\ xs\ @\ r & = & \mathbf{let}\ ys = (\mathbf{case}\ xs\ \mathbf{of}\ (x : xx) \rightarrow xx)\ \mathbf{in} \\
 & & \mathbf{let}\ zs = (1 : ys)@r\ \mathbf{in\ case!}\ xs\ \mathbf{of}\ \ldots
\end{array}$$

The variable ys refers to the tail of the list pointed to by xs, which is used in the definition of zs to build another list. After that, the first cons cell of the list pointed to by xs is disposed by means of a **case**!. In order to type the **case**! $xs\ \ldots$ expression under an environment Γ, the variable xs must appear in this environment with a condemned type $[\mathit{Int}]!@\rho$. Moreover, since both ys and zs are sharing a recursive descendant of xs, they must occur in Γ with an in-danger type $[\mathit{Int}]\#@\rho$. Note that, regarding the typing environment of the non-destructive **case** in the definition of ys, the variable xs may appear in this environment with a safe type $[\mathit{Int}]@\rho$, since it is not being destroyed there.

The motivation for not allowing in-danger types in function signatures is the following: when a parameter is condemned we know clearly that the recursive substructure of the DS is condemned. When the function is applied to an argument we know that the recursive substructure of such argument may be partially or totally destroyed. However when a parameter is in-danger the only thing we know is that some part (recursive or not) of the whole DS may be dangling but we do not know which. This is a very imprecise information to put in the type of a function.
We will write $T@\overline{\rho}^m$ instead of $T\ \overline{s}@\overline{\rho}^m$ to abbreviate whenever the $\overline{s}$ are not relevant. We shall even use $T@\rho$ to highlight only the outermost region. A partial order between types is defined: $\tau \geq \tau$, $T!@\overline{\rho}^m \geq T@\overline{\rho}^m$, and $T\#@\overline{\rho}^m \geq T@\overline{\rho}^m$. This partial order is extended below to type environments in the context of the expression being typed.

Predicates $\mathit{region?}(\tau)$ and $\mathit{function?}(\tau)$ respectively indicate that τ is a region type or a functional type.

Constructor types have one region argument ρ which coincides with the outermost region variable of the resulting algebraic type $T\ \overline{s}@\overline{\rho}^m$. As recursive sharing of DSs may happen only inside the same region, the constructors are given types indicating that the recursive substructure and the structure itself must live in the same region. For example, in the case of lists and trees:

$$\begin{array}{l}
[\,] : \forall a, \rho.\rho \rightarrow [a]@\rho \\
(:) : \forall a, \rho.a \rightarrow [a]@\rho \rightarrow \rho \rightarrow [a]@\rho \\
\mathit{Empty} : \forall a, \rho.\rho \rightarrow \mathit{BSTree}\ a@\rho \\
\mathit{Node} : \forall a, \rho.\mathit{BSTree}\ a@\rho \rightarrow a \rightarrow \mathit{BSTree}\ a@\rho \rightarrow \rho \rightarrow \mathit{BSTree}\ a@\rho
\end{array}$$

We assume that the types of the constructors are collected in an environment Σ, easily built from the **data** type declarations.

In functional types returning a DS, where there may be several region arguments $\overline{\rho}^l$, these are a subset of the result's regions $\overline{\rho}^m$. The reason is that our region inference algorithm generates as region arguments only those that are actually needed to build the result. A function like `f x @ r = x` of type `f :: a -> rho -> a`, cannot be obtained from the desugaring of a *Full-Safe* program, but we can have

```
data T a @ rho1 rho2 = (C [a]@rho1)@rho2

g :: [a]@rho1 -> rho2 -> T a @ rho1 rho2
g xs @ r = C xs @ r
```

where `rho1` is not an argument as the function does not build anything there.

In the type environments, Γ, we can find region type assignments $r : \rho$, variable type assignments $x : t$, and polymorphic scheme assignments to functions $f : \sigma$. In the rules we will also use $gen(tf, \Gamma)$ and $tf \trianglelefteq \sigma$ to respectively denote (standard) generalization of a monomorphic type and restricted instantiation of a polymorphic type with safe types.

The operators on type environments used in the typing rules are shown in Fig. 4. The usual operator $+$ demands disjoint domains. Operators $\otimes$ and $\oplus$ are defined only if common variables have the same type, which must be safe in the case of $\oplus$. If one of these operators is not defined in a rule, we assume that the rule cannot be applied. Operator $\rhd^L$ is explained below. The predicate $utype?(t, t')$ is true when the underlying Hindley-Milner types of t and t' are the same.

We now explain in detail the typing rules. In Fig. 5 we present the rule [FUNB] for function definitions. Notice that the only regions in scope are the region parameters $\overline{r_j}^l$ and $self$, which gets a fresh region type ρ_{self}. The latter cannot appear in the type of the result as $self$ dies when the function returns its value ($\rho_{self} \notin regions(s)$). To type a complete program the types of the functions are accumulated in a growing environment and then the main expression is typed.

In Figure 6, the rules for typing expressions are shown. Function $sharerec(x, e)$ gives an upper approximation to the set of variables in scope in e which share a recursive descendant of the DS starting at x. This set is computed by the abstract interpretation based sharing analysis defined in [18].

One of the key points to prove the correctness of the type system with respect to the semantics is an invariant of the type system (see Lemma 2) telling that if a variable appears as condemned in the typing environment, then those variables sharing a recursive substructure appear also in the environment with unsafe types. This is necessary in order to propagate information about the possibly damaged pointers.

There are rules for typing literals ([LIT]), and variables of several kinds ([VAR], [REGION] and [FUNCTION]). Notice that these are given a type under the smallest typing environment.

Rules [EXTS] and [EXTD] allow to extend the typing environments in a controlled way. The addition of variables with safe types, in-danger types, region types or functional types is allowed. If a variable with a condemned type is added, all those variables sharing its recursive substructure but itself must be also added to the environment with its corresponding in-danger type in order to preserve the invariant mentioned above. Notation $type(y)$ represents the Hindley-Milner type inferred for variable y[1].

[1] The implementation of the inference algorithm proceeds by first inferring Hindley-Milner types and then the destruction annotations.

$$\begin{array}{l}
inh(s, s, \tau) \leftrightarrow safe?(\tau) \vee dgr?(\tau) \vee (\neg utype?(s, \tau) \wedge cmd?(\tau)) \\
inh(danger(s), s, \tau) \leftrightarrow dgr?(\tau) \vee (utype?(s, \tau) \wedge cmd?(\tau)) \\
\\
inh!(s, s, d) \leftrightarrow \neg utype?(s, d) \\
inh!(d, s, d) \leftrightarrow utype?(s, d)
\end{array}$$

Figure 7. Definitions of inheritance compatibility

Rule [COPY] allows any variable to be copied. This is expressed by extending the previously defined partial order between types to environments:

$$\begin{array}{rcl}
\Gamma_1 \geq_e \Gamma_2 & \equiv & dom(\Gamma_2) \subseteq dom(\Gamma_1) \\
& & \wedge\ \forall x \in dom(\Gamma_2).\Gamma_1(x) \geq \Gamma_2(x) \\
& & \wedge\ \forall x \in dom(\Gamma_1).\ cmd?(\Gamma_1(x)) \rightarrow \\
& & \quad \forall z \in sharerec(x, e).z \in dom(\Gamma_1) \\
& & \qquad\qquad \wedge\ unsafe?(\Gamma_1(z))
\end{array}$$

The third conjunction of this definition enforces variables pointing to a recursive substructure of a condemned variable in Γ_1 to appear in this environment with an unsafe type, so that the invariant of the type system still holds.

Rule [LET] controls the intermediate data structures, that may be safe, condemned or in-danger in the main expression (τ covers the three cases). Operator $\rhd^L$, defined in Figure 4, guarantees that:

1. Each variable y condemned or in-danger in e_1 may not be referenced in e_2 (i.e. $y \notin fv(e_2)$), as it could be a dangling reference.
2. Those variables marked as unsafe either in Γ_1 or in Γ_2 will keep those types in the combined environment.

Rule [REUSE] establishes that in order to reuse a variable, it must have a condemned type in the environment. Those variables sharing its recursive descendants are given in-danger types in the environment.

Rule [APP] deals with function application. The use of the operator $\oplus$ avoids a variable to be used in two or more different positions unless they are all safe parameters. Otherwise undesired side-effects could happen. The set R collects all the variables sharing a recursive substructure of a condemned parameter, which are marked as in-danger in environment Γ_R.

Rule [CONS] is more restrictive as only safe variables can be used to construct a DS.

Rule [CASE] allows its discriminant variable to be safe, in-danger, or condemned as it only reads the variable. Relation inh, defined in Figure 7, determines which types are acceptable for pattern variables according to the previously explained semantics. Apart from the fact that the underlying types are correct from the Hindley-Milner point of view: if the discriminant is safe, so must be all the pattern variables; if it is in-danger, the pattern variables may be safe or in-danger; if it is condemned, recursive pattern variables are in-danger while non-recursive ones are safe.

In rule [CASE!] the discriminant is destroyed and consequently the text should not try to reference it in the alternatives. The same happens to those variables sharing a recursive substructure of x, as they may be corrupted. All those variables are added to the set R. Relation $inh!$, defined in Fig. 7, determines the types inherited by pattern variables: recursive ones are condemned while non-recursive ones must be safe.

As recursive pattern variables inherit condemned types, the type environments for the alternatives contain all the variables sharing their recursive substructures as in-danger. In particular x may appear with an in-danger type. In order to type the whole expression we must change it to condemned.

Operator ($\bullet$)	**$\Gamma_1 \bullet \Gamma_2$ defined if**	**Result of $(\Gamma_1 \bullet \Gamma_2)(x)$**
$+$	$dom(\Gamma_1) \cap dom(\Gamma_2) = \emptyset$	$\Gamma_1(x)$ if $x \in dom(\Gamma_1)$ $\Gamma_2(x)$ otherwise
$\otimes$	$\forall x \in dom(\Gamma_1) \cap dom(\Gamma_2) \,.\, \Gamma_1(x) = \Gamma_2(x)$	$\Gamma_1(x)$ if $x \in dom(\Gamma_1)$ $\Gamma_2(x)$ otherwise
$\oplus$	$\forall x \in dom(\Gamma_1) \cap dom(\Gamma_2) \,.\ \Gamma_1(x) = \Gamma_2(x)$ $\wedge\ safe?(\Gamma_1(x))$	$\Gamma_1(x)$ if $x \in dom(\Gamma_1)$ $\Gamma_2(x)$ otherwise
$\rhd^L$	$(\forall x \in dom(\Gamma_1) \cap dom(\Gamma_2).\ utype?(\Gamma_1(x), \Gamma_2(x)))$ $\wedge\ (\forall x \in dom(\Gamma_1).\ unsafe?(\Gamma_1(x)) \rightarrow x \notin L)$	$\Gamma_2(x)$ if $x \notin dom(\Gamma_1) \vee$ $(x \in dom(\Gamma_1) \cap dom(\Gamma_2)$ $\wedge\ safe?(\Gamma_1(x)))$ $\Gamma_1(x)$ otherwise

Figure 4. Operators on type environments

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

Figure 5. Rule for function definitions

LEMMA 2. *If* $\Gamma \vdash e : s$ *and* $\Gamma(x) = d$ *then:*

$$\forall y \in sharerec(x, e) - \{x\} : y \in dom(\Gamma) \wedge unsafe?(\Gamma(y))$$

Proof: By induction on the depth of the type derivation. □

An inference algorithm for this type system has been developed. A detailed description can be found in [10].

5. Correctness of the Type System

The proof proceeds in two steps: first we prove absence of dangling pointers due to destructive pattern matching and then the safety of the region deallocation mechanism.

5.1 Absence of Dangling Pointers due to Cell Destruction

The intuitive idea of a variable x being typed with a safe type s is that all the cells in h reachable from $E(x)$ are also safe and they should be disjoint of unsafe cells. The idea behind a condemned variable x is that all variables (including itself) and all live cells sharing any of its recursive descendants are unsafe. Firstly, formal definitions of reachability and sharing are given:

DEFINITION 3. *Given a heap h, we define the child ($\rightarrow_h$) and recursive child ($\twoheadrightarrow_h$) relations on heap pointers as follows:*

$$\begin{array}{lcl} p \rightarrow_h q & \stackrel{def}{=} & h(p) = (j, C\ \overline{v_i}^n)\ \wedge\ q \in \overline{v_i}^n \\ p \twoheadrightarrow_h q & \stackrel{def}{=} & h(p) = (j, C\ \overline{v_i}^n)\ \wedge\ q = v_i \\ & & \textit{for some } i \in recPos(C) \end{array}$$

where $recPos(C)$ is the set of recursive argument positions of constructor C.

The reflexive and transitive closure of these relations are respectively denoted by $\rightarrow_h^*$ and $\twoheadrightarrow_h^*$. In addition, we will use the following terminology:

$closure(E, X, h) \stackrel{\text{def}}{=} \{q \mid E(x) \rightarrow_h^* q\ \wedge\ x \in X\}$

Set of locations reachable in h by $\{E(x) \mid x \in X\}$.

$closure(p, h) \stackrel{\text{def}}{=} \{q \mid p \rightarrow_h^* q\}$

Set of locations reachable in h by location p.

$live(E, L, h) \stackrel{\text{def}}{=} closure(E, L, h)$

Live part of h, i.e. $closure(E, L, h)$.

$recReach(E, x, h) \stackrel{\text{def}}{=} \{q \mid E(x) \twoheadrightarrow_h^* q\}$

Set of recursive descendants of $E(x)$ including itself.

$closed(E, L, h) \stackrel{\text{def}}{=} live(E, L, h) \subseteq dom(h)$

If there are no dangling pointers in $live(E, L, h)$.

$p \rightarrow_h^* V \stackrel{\text{def}}{=} \exists q \in V.\, p \rightarrow_h^* q$

There is a pointer path from p to a $q \in V$.

By abuse of notation, we will write $closure(E, x, h)$ instead of $closure(E, \{x\}, h)$, and also $closed(v, h)$ to indicate that there are no dangling pointers in $closure(v, h)$.

The correctness of the sharing analysis mentioned in Section 4 has been proved elsewhere and it is not the subject of this paper, but we need it in order to prove the correctness of the whole type system. We will assume then the following property:

$$\begin{array}{l}\forall x, y \in scope(e).\\ \quad closure(E, x, h) \cap recReach(E, y, h) \neq \emptyset \rightarrow x \in sharerec(y, e)\end{array}$$

If expression e reduces to v, i.e. $E \vdash h, k, e \Downarrow h', k, v$, and $\Gamma \vdash e : s$, and $L = fv(e)$, we will call *initial configuration* to the tuple (Γ, E, h, L, s) combining static information about variables and types of expression e and dynamic information such as the runtime environment E and the initial heap h. Likewise, we will call *final configuration* to the tuple (s, v, h') including the final value and heap together with the static type s of the original expression (hence, s is also the type of the value).

In the following, we will use the notations $\Gamma[x] = t$ and $\Gamma \vdash e : t$, with $t \in \{s, d, r\}$, to indicate that the type of x and e are respectively a safe, condemned or in-danger type. Now, we define the following two sets of heap locations as functions of an initial configuration (Γ, E, h, L, s):

$$S \stackrel{\text{def}}{=} \bigcup_{x \in L, \Gamma[x]=s} \{closure(E, x, h)\}$$
$$R \stackrel{\text{def}}{=} \bigcup_{x \in L, \Gamma[x]=d} \{p \in live(E, L, h) \mid p \rightarrow_h^* recReach(E, x, h)\}$$

DEFINITION 4. *An initial configuration (Γ, E, h, L, s) is said to be* good *whenever:*

1. $E \vdash h, k, e \Downarrow h', k, v,\ L = fv(e),\ \Gamma \vdash e : s$*, and*
2. $S \cap R = \emptyset$*, and*

$$\frac{\begin{array}{c}\Gamma \vdash e : s \quad x \notin \mathit{dom}(\Gamma)\\ \mathit{safe?}(\tau) \vee \mathit{danger?}(\tau) \vee \mathit{region?}(\tau) \vee \mathit{function?}(\tau)\end{array}}{\Gamma + [x : \tau] \vdash e : s}\ \text{[EXTS]} \qquad \frac{\begin{array}{c}\Gamma \vdash e : s \quad x \notin \mathit{dom}(\Gamma)\\ R = \mathit{sharerec}(x, e) - \{x\}\\ \Gamma_R = \{y : \mathit{danger}(\mathit{type}(y)) |\ y \in R\}\end{array}}{\Gamma \otimes \Gamma_R + [x : d] \vdash e : s}\ \text{[EXTD]}$$

$$\frac{}{\emptyset \vdash c : B}\ \text{[LIT]} \qquad \frac{}{[x : s] \vdash x : s}\ \text{[VAR]} \qquad \frac{}{[r : \rho] \vdash r : \rho}\ \text{[REGION]} \qquad \frac{\mathit{tf} \trianglelefteq \sigma}{[f : \sigma] \vdash f : \mathit{tf}}\ \text{[FUNCTION]}$$

$$\frac{\begin{array}{c}R = \mathit{sharerec}(x, x!) - \{x\}\\ \Gamma_R = \{y : \mathit{danger}(\mathit{type}(y)) |\ y \in R\}\end{array}}{\Gamma_R + [x : T!@\rho] \vdash x! : T@\rho}\ \text{[REUSE]} \qquad \frac{\Gamma_1 \geq_{x@r} [x : T@\rho', r : \rho]}{\Gamma_1 \vdash x@r : T\ @\rho}\ \text{[COPY]}$$

$$\frac{\Gamma_1 \vdash e_1 : s_1 \qquad \Gamma_2 + [x_1 : \tau_1] \vdash e_2 : s \qquad \mathit{utype?}(\tau_1, s_1)}{\Gamma_1 \rhd^{fv(e_2)} \Gamma_2 \vdash \mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2 : s}\ \text{[LET]}$$

$$\frac{\begin{array}{c}\overline{t_i}^n \to \overline{\rho_j}^l \to T\ @\overline{\rho}^m \trianglelefteq \sigma \qquad \Gamma = [f : \sigma] + \bigoplus_{j=1}^{l}[r_j : \rho_j] + \bigoplus_{i=1}^{n}[a_i : t_i]\\ R = \bigcup_{i=1}^{n}\{\mathit{sharerec}(a_i, f\ \overline{a_i}^n @\overline{r_j}^l) - \{a_i\} \mid \mathit{cmd?}(t_i)\} \quad \Gamma_R = \{y : \mathit{danger}(\mathit{type}(y)) |\ y \in R\}\end{array}}{\Gamma_R + \Gamma \vdash f\ \overline{a_i}^n @\ \overline{r_j}^l : T\ @\overline{\rho}^m}\ \text{[APP]}$$

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

$$\frac{\begin{array}{cc}\forall i \in \{1..n\}.\Sigma(C_i) = \sigma_i & \forall i \in \{1..n\}.\overline{s_i}^{n_i} \to \rho \to T\ @\rho \trianglelefteq \sigma_i\\ \Gamma \geq_{\mathbf{case}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n} [x : T@\rho] & \forall i \in \{1..n\}.\forall j \in \{1..n_i\}.\mathit{inh}(\tau_{ij}, s_{ij}, \Gamma(x))\\ \multicolumn{2}{c}{\forall i \in \{1..n\}.\Gamma + \overline{[x_{ij} : \tau_{ij}]}^{n_i} \vdash e_i : s}\end{array}}{\Gamma \vdash \mathbf{case}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n : s}\ \text{[CASE]}$$

$$\frac{\begin{array}{ll}(\forall i \in \{1..n\}).\ \Sigma(C_i) = \sigma_i & \forall i \in \{1..n\}.\ \overline{s_i}^{n_i} \to \rho \to T\ @\rho \trianglelefteq \sigma_i\\ R = \mathit{sharerec}(x, \mathbf{case!}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n) - \{x\} & \forall i \in \{1..n\}.\ \forall j \in \{1..n_i\}.\mathit{inh!}(t_{ij}, s_{ij}, T\ !@\rho)\\ \forall z \in R \cup \{x\}, i \in \{1..n\}.z \notin \mathit{fv}(e_i) & \forall i \in \{1..n\}.\ \Gamma + [x : T\ \#@\rho] + \overline{[x_{ij} : t_{ij}]}^{n_i} \vdash e_i : s\\ \multicolumn{2}{c}{\Gamma_R = \{y : \mathit{danger}(\mathit{type}(y)) \mid y \in R\}}\end{array}}{\Gamma_R \otimes \Gamma + [x : T\ !@\rho] \vdash \mathbf{case!}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n : s}\ \text{[CASE!]}$$

Figure 6. Type rules for expressions

3. *$closed(E, L, h)$.*

By analogy, a final configuration (s, v, h') is good *whenever $closed(v, h')$ holds.*

We claim that the property $closed(E, L, h)$ is invariant along the execution of any well-typed *Safe* program. This will prove that dangling pointers never arise at runtime. Previously, we need the following lemma expressing that safe pointers in the heap are preserved by evaluation.

LEMMA 5. *Let (Γ, E, h, L, s) be an initial good configuration. Then, for all $x \in L$ such that $\Gamma[x] = s$ we have $closure(E, x, h) = closure(E, x, h')$.*

Proof: By induction on the size of the $\Downarrow$ derivation. □

THEOREM 6. *Let e be a Core-Safe expression. Let us assume that (Γ, E, h, L, s) is good. Then, (s, v, h') is good, and all the intermediate configurations in the derivation tree of $\Downarrow$ are good.*

Proof: By induction on the size of the $\Downarrow$ derivation. □

Hence, if the initial configuration for a expression e is good, during the evaluation of e it never arises a dangling pointer in the heap. As, when executing a *Safe* program, the heap is initially empty (so, closed), and there are no free variables, (so, $S = R = \emptyset$), the initial configuration is good. We conclude then that all well-typed *Safe* program never produce dangling pointers at runtime.

5.2 Correctness of Region Deallocation

At the end of each function call the topmost region is deallocated, which could be a source of dangling pointers. This section proves that the structure returned by the function call does not reside in *self*. First we shall show that the topmost region is only referenced by the current *self*:

LEMMA 7. *Let e_0 be the main expression of a Core-Safe program and let us assume that $[self \mapsto 0] \vdash \emptyset, 0, e_0 \Downarrow h_f, 0, v_f$ can be derived. Then in every judgment $E \vdash h, k, e \Downarrow h', k, v$ belonging to this derivation it holds that:*

1. *$self \in dom(E)\ \wedge\ E(self) = k$.*
2. *For every region variable $r \in dom(E)$, if $r \neq self$ then $E(r) < k$.*

Proof: By induction on the depth of $\Downarrow$ derivation. □

This lemma allows us to leave out the condition $j \leq k$ in rules $[Let_2]$ and $[Var_2]$ of Fig. 2. The rest of the correctness proof is to establish a correspondence between type region variables ρ and region numbers j. If a variable admits the algebraic type $T@\overline{\rho_i}^n$

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

Figure 8. Definition of *build* function.

$$
\begin{array}{l}
copy(h_0[p \mapsto (k, C\ \overline{v_i}^n)], p, j) \quad = \quad (h_n \cup [p' \mapsto (j, C\ \overline{v_i'}^n)], p') \\
\quad \textbf{where} \quad fresh(p') \\
\qquad \forall i \in \{1..n\}.(h_i, v_i') = \left\{ \begin{array}{ll} (h_{i-1}, v_i) & \text{if } v_i = c \vee i \notin RecPos(C) \\ copy(h_{i-1}, v_i, j) & \text{otherwise} \end{array} \right.
\end{array}
$$

Figure 9. Definition of *copy*.

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* and is defined as follows:

DEFINITION 8. *A **region instantiation** θ is a function from type region variables to natural numbers (interpreted as regions). It can also be defined as a set of **bindings** $[\rho \to n]$, where no variable ρ occurs twice in the left-hand side of a binding unless it is bound to the same region number.*

*Two region instantiations θ and θ' are said to be **consistent** if they bind common type region variables to the same number, that is: $\forall \rho \in dom(\theta) \cap dom(\theta')$. $\theta(\rho) = \theta'(\rho)$.*

*The **union** of two region instantiations θ and θ' (denoted by $\theta \cup \theta'$) is defined only if θ and θ' are consistent and returns another region instantiation over $dom(\theta) \cup dom(\theta')$ defined as follows:*

$$
(\theta \cup \theta')(\rho) = \left\{ \begin{array}{ll} \theta(\rho) & \text{if } \rho \in dom(\theta) \\ \theta'(\rho) & \text{otherwise} \end{array} \right.
$$

Given a pointer and a type, the function *build*, defined in Fig. 8, returns the corresponding region instantiation.

If p is a dangling pointer, its corresponding *build* is well-defined. However, dangling pointers are never accessed by a program (Sec 5.1). 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.

DEFINITION 9. *Let E be a variable environment, h a heap and Γ a type environment. We say that E is consistent with h under type environment Γ iff:*

1. *For all non-region variables $x \in dom(E)$ the result of $build(h, E(x), \Gamma(x))$ is well-defined.*
2. *For each pair of non-region variables $x, y \in dom(E)$: $build(h, E(x), \Gamma(x))$ and $build(h, E(y), \Gamma(y))$ are consistent. In other words, if we define:*

$$
\theta_X = \bigcup_{z \in dom(E)} build(h, E(z), \Gamma(z))
$$

   then θ_X is well-defined.
3. *If θ_R is defined as follows:*

$$
\begin{array}{rcl}
\theta_R & = & \{[\Gamma(r) \to E(r)] \mid \\
& & \quad r \text{ is a region variable and } r \in dom(E)\}
\end{array}
$$

   Then θ_X and θ_R are consistent.

*The result of $\theta_X \cup \theta_R$ is called the **witness** of this consistency relation.*

The notation $x@$, which allows to copy the recursive spine of a DS, is introduced in Section 2. As much as we copy a DS, the result of the *build* function applied to the fresh pointer created is well-defined if the result of the *build* corresponding to the original DS is also well-defined:

DEFINITION 10. *The function copy is defined as shown in Fig. 9.*

LEMMA 11. *If $\theta = build(h, p, T@\rho)$ is well-defined and $(h', p') = copy(h, j, p)$, then for all ρ' such that $[\rho' \to j]$ is consistent with θ, $build(h', p', T@\rho')$ is well-defined and consistent with θ.*

Proof: By induction on the size of the structure pointed to by p. □

The following theorem proves that consistency is preserved by evaluation.

THEOREM 12. *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. □

So far we have set up a correspondence between the actual regions where a data structure resides and the corresponding region types assigned by the type system: 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. 5) 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*, as Lemma 7 states) at the end of a function call does not generate dangling pointers.

6. Examples

Now we shall consider the *concatD*, *treesort* and *treesortD* functions defined in Sec. 2. The desugared versions of their definitions are shown in Fig. 10. The first column is the result of the region inference phase, which inserts the $@r$ annotations into the code. Temporary structures are assigned the working region *self*. The second column shows the translation to *Core-Safe*.

Function *concatD* has type $[a]!@\rho_1 \to [a]@\rho \to \rho \to [a]@\rho$. Rule [FUNB] establishes that its body must be typed with *zs* being condemned and *ys* being safe. The typing derivation is shown in Fig. 11. The typing rule [CASE!] is applied in (1). The branch guarded by [ ] can be typed by means of the [VAR] and [EXTS]

***Full-Safe* with regions**	***Core-Safe***
$\mathit{concatD}\ \ [\,]!\ \ \mathit{ys}\ @\ r = \mathit{ys}$ $\mathit{concatD}\ \ (x : \mathit{xs})!\ \ \mathit{ys}\ @\ r = (x : \mathit{concatD}\ \mathit{xs}\ \mathit{ys}\ @\ r)@\ r$	$\mathit{concatD}\ \mathit{zs}\ \mathit{ys}\ @\ r =$ $\quad \mathbf{case!}\ \mathit{zs}\ \mathbf{of}$ $\qquad [\,] \to \mathit{ys}$ $\qquad (x : \mathit{xs}) \to \mathbf{let}\ x_1 = \mathit{concatD}\ \mathit{xs}\ \mathit{ys}\ @\ r$ $\qquad\qquad \mathbf{in}\ (x : x_1)@\ r$
$\mathit{treesortD}\ \mathit{xs}\ @\ r = \mathit{inorder}\ (\mathit{mkTreeD}\ \mathit{xs}\ @\ \mathit{self})\ @\ r$	$\mathit{treesortD}\ \mathit{xs}\ @\ r =$ $\quad \mathbf{let}\ x_1 = \mathit{mkTreeD}\ \mathit{xs}\ @\ \mathit{self}$ $\quad \mathbf{in}\ \mathit{inorder}\ x_1\ @\ r$
$\mathit{treesort}\ \mathit{xs}\ @\ r = \mathit{treesortD}\ (\mathit{xs}@\mathit{self})\ @\ r$	$\mathit{treesort}\ \mathit{xs}\ @\ r = \mathbf{let}\ \mathit{xs}' = \mathit{xs}@\mathit{self}$ $\qquad \mathbf{in}\ \mathit{treesortD}\ \mathit{xs}'\ @\ r$

Figure 10. Desugared versions of *concatD*, *treesortD* and *treesort*

$$\dfrac{\dfrac{\cdots}{\Gamma_1 \vdash \mathit{ys} : [a]@\rho}\ (2) \qquad \dfrac{\dfrac{\cdots}{\Gamma_3 \vdash \mathit{concatD}\ \mathit{xs}\ \mathit{ys}\ @r : [a]@\rho}\ (4) \quad \dfrac{\cdots}{\Gamma_4 + [x_1 : [a]@\rho] \vdash (x : x_1)@r : [a]@\rho}\ (5)}{\Gamma_2 \vdash \mathbf{let}\ x_1 = \ldots\ \mathbf{in}\ \ldots : [a]@\rho}\ (3)}{\Gamma \vdash \mathbf{case!}\ \mathit{zs}\ \mathbf{of} \ldots : [a]@\rho}\ (1)$$

$$\begin{array}{lcl}
\Gamma & = & \Gamma' + [\mathit{zs} : [a]!@\rho_1] \\
\Gamma' & = & [\mathit{ys} : [a]@\rho, r : \rho, \mathit{self} : \rho_{\mathit{self}}, \mathit{concatD} : \sigma] \\
\Gamma_1 & = & \Gamma' + [\mathit{zs} : [a]\#@\rho_1] \\
\Gamma_2 & = & \Gamma' + [\mathit{zs} : [a]\#@\rho_1, x : a, \mathit{xs} : [a]!@\rho]
\end{array}
\qquad
\begin{array}{lcl}
\Gamma_3 & = & [\mathit{xs} : [a]!@\rho_1, \mathit{zs} : [a]\#@\rho_1, \mathit{ys} : [a]@\rho, r : \rho, \mathit{concatD} : \sigma] \\
\Gamma_4 & = & [x : a, r : \rho, \mathit{self} : \rho_{\mathit{self}}] \\
\sigma & = & [a]!@\rho_1 \to [a]@\rho \to \rho \to [a]@\rho
\end{array}$$

Figure 11. Simplified typing derivation for *concatD*

$$\dfrac{\dfrac{\cdots}{\Gamma_1 \vdash \mathit{mkTreeD}\ \mathit{xs}\ @\ \mathit{self} : \mathit{BSTree}\ \mathit{Int}@\rho_{\mathit{self}}}\ (2) \quad \dfrac{\cdots}{\Gamma_2 + [x_1 : \mathit{BSTree}\ \mathit{Int}@\rho_{\mathit{self}}] \vdash \mathit{inorder}\ x_1\ @\ r : [\mathit{Int}]@\rho}\ (3)}{\Gamma \vdash \mathbf{let}\ x_1 = \mathit{mkTreeD}\ \mathit{xs}\ @\ \mathit{self}\ \mathbf{in}\ \mathit{inorder}\ x_1\ @\ r : [\mathit{Int}]@\rho}\ (1)$$

$$\Gamma = [\mathit{xs} : [\mathit{Int}]!@\rho_1, r : \rho, \mathit{self} : \rho_{\mathit{self}}, \mathit{mkTreeD} : \sigma_1, \mathit{inorder} : \sigma_2, \mathit{treesortD} : \sigma]$$

$$\begin{array}{l}
\Gamma_1 = [\mathit{xs} : [\mathit{Int}]!@\rho_1, \mathit{self} : \rho_{\mathit{self}}, \mathit{mkTreeD} : \sigma_1] \\
\Gamma_2 = [r : \rho, \mathit{inorder} : \sigma_2, \mathit{treesortD} : \sigma]
\end{array}
\qquad
\begin{array}{l}
\sigma_1 = \forall \rho_1, \rho_2 . [\mathit{Int}]!@\rho_1 \to \rho_2 \to \mathit{BSTree}\ \mathit{Int}@\rho_2 \\
\sigma_2 = \forall a, \rho_1, \rho_2 . \mathit{BSTree}\ a@\rho_1 \to \rho_2 \to [a]@\rho_2 \\
\sigma = \forall \rho_1, \rho . [\mathit{Int}]!@\rho_1 \to \rho \to [\mathit{Int}]@\rho
\end{array}$$

Figure 12. Simplified typing derivation for *treesortD*

rules (2). With respect to the second branch, the definition of $\mathit{inh}!$ specifies that xs must have a condemned type in Γ, since it is a recursive child of zs (i.e. has the same underlying type). In (3) the rule [LET] is applied, where x_1 is not used destructively in the main expression of the **let** binding. We have $\Gamma_2 = \Gamma_3 \rhd^{\{x, x_1\}} \Gamma_4$, which is well-defined since the unsafe variables in $\mathit{dom}(\Gamma_2)$ (i.e. xs and zs) do not occur free in the expression $(x : x_1)@r$. The bound expression of $\mathbf{let}\ x_1 = \ldots$ is typed via the [APP] rule (4) and in its main expression the rule [CONS] is applied (5).

The typing derivation for *treesortD* is shown in Fig. 12. We assume that *mkTreeD* and *inorder* have been already typed, obtaining σ_1 and σ_2, respectively. The rule [LET] is applied in (1), where x_1 is not destroyed in the call to *inorder*. In addition, variable xs does not occur free there, so the environment $\Gamma = \Gamma_1 \rhd^{\{x_1\}} \Gamma_2$ is well-defined. In (2) the rule [APP] is applied, while in (3) first we apply [EXTS] in order to exclude the binding $[\mathit{treesortD} : \sigma]$ of Γ_2 and then [APP]. With respect to *treesort*, we get the following type scheme: $\forall \rho_1, \rho . [\mathit{Int}]@\rho_1 \to \rho \to [\mathit{Int}]@\rho$. To type its body, rule [LET] is now applied, where xs' is destroyed in the *treesortD* call.

The types of the remaining *Safe* functions presented in Section 2 could be derived in the same way. Including regions, the derived types are the following:

$$\begin{array}{l}
\mathit{mkTreeD} :: \forall \rho_1, \rho_2\ .\ [\mathit{Int}]!@\rho_1 \to \rho_2 \to \mathit{BSTree}\ \mathit{Int}@\rho_2 \\
\mathit{insertD} :: \forall \rho\ .\ \mathit{Int} \to \mathit{BSTree}\ \mathit{Int}!@\rho \to \rho \to \mathit{BSTree}\ \mathit{Int}@\rho \\
\mathit{splitD} :: \forall a, \rho_1, \rho_2, \rho_3\ .\ \mathit{Int} \to [a]!@\rho_2 \to \rho_1 \to \rho_2 \to \rho_3 \\
\qquad \to ([a]@\rho_1, [a]@\rho_2)@\rho_3
\end{array}$$

7. Related work

The use of regions in functional languages to avoid garbage collection is not new. Tofte and Talpin [20] introduced in ML-Kit —a variant of ML— the use of nested regions by means of a **letregion** construct. A lot of work has been done on this system [1, 4, 19]. Their main contribution is a *region inference* algorithm adding region annotations at the intermediate language level.

Hughes and Pareto [7] incorporate regions in Embedded-ML. This language uses a sized-types system in which the programmer annotates heap and stack sizes and these annotations can be type-checked. So, regions can be proved to be bounded.

A small difference with these approaches is that, in *Safe*, region allocation and deallocation are synchronized with function calls instead of being introduced by a special language construct. A more relevant difference is that *Safe* has an additional mechanism allowing the programmer to selectively destroy data structures inside a region.

A difficulty with the original Tofte and Talpin's system is the fact that regions have nested lifetimes. There exist a few programs that result in memory leaks due to this restriction. In [5] this problem is alleviated by defining a variant of λ-calculus with type-safe primitives for creating, accessing and destroying regions, which are not restricted to have nested lifetimes. Programs are written in a C-like language called *Cyclone* having explicit memory management primitives, then it is translated into this variant of λ-calculus, and then type checked. So, the price of this flexibility is explicit region control.

In our language *Safe*, regions also suffer from the nested lifetimes constraint, since both region allocation and deallocation are bound to function calls, which are necessarily nested. 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 distinct data structures are not necessarily nested, and the type system presented here protects the programmer against missuses of this feature. Again, the price of this flexibility is explicit deallocation of cells. Allocation is implicit in constructions and the target region of the allocation is inferred by the compiler. It is arguable whether it is better to explicitly manage regions or cells.

More recently, Hofmann and Jost [6] have developed a type system to infer heap consumption. Theirs is also a first-order eager functional language with a construct $match'$ that destroys constructor cells. Its operational behaviour is similar to that of *Safe*'s **case**!. The main difference is that they lack a compile time analysis guaranteeing the safe use of this dangerous feature. Also, their language does not use regions. In [18] a more detailed comparison with all these works can be found.

Our safety type system has some characteristics of linear types (see [21] as a basic reference). A number of variants of linear types have been developed for years for coping with the related problems of achieving safe updates in place in functional languages [17] or detecting program sites where values could be safely deallocated [8]. The work closest to our system is [2], which proposes a type system for a language explicitly reusing heap cells. They prove that well-typed programs can be safely translated into an imperative language with an explicit deallocation/reusing mechanism. We summarise here the differences and similarities with our work.

There are non-essential differences such as: (1) they only admit algorithms running in constant heap space, i.e. for each allocation there must exist a previous deallocation; (2) they use at the source level an explicit parameter d representing a pointer to the cell being reused; and (3) they distinguish two different cartesian products depending on whether there is sharing or not between the tuple components. But, in our view, the following more essential differences makes our type-system more powerful than theirs:

1. Their uses 2 and 3 (read-only and shared, or just read-only) could be roughly assimilated to our use s (read-only), and their use 1 (destructive), to our use d (condemned), both defined in Section 4. We add a third use r (in-danger) arising from a sharing analysis based on abstract interpretation [18]. This use allows us to know more precisely which variables are in danger when some other one is destroyed.
2. Their uses form a total order $1 < 2 < 3$. A type assumption can always be worsened without destroying the well-typedness. Our marks s, r, d do not form a total order. Only in some expressions (**case** and $x@r$) we allow the partial order $s \leq r$ and $s \leq d$. It is not clear whether that order gives or not more power to the system. In principle it will allow diferent uses of a variable in different branches of a conditional being the use of the whole conditional the worst one. For the moment our system does not allow this.
3. Their system forbids non-linear applications such as $f(x, x)$. We allow them for s-type arguments.
4. Our typing rule for $\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2$ allow more use combinations than theirs. Let $i \in \{1, 2, 3\}$ the use assigned to x_1, j the use of a variable z in e_1, and k the use of the variable z in e_2. We allow the following combinations (i, j, k) that they forbid: $(1, 2, 2)$, $(1, 2, 3)$, $(2, 2, 2)$, $(2, 2, 3)$. The deep reason is our more precise sharing information and the new in-danger type. In a more recent version of this sytem [3] combination $(2, 2, 3)$ is allowed.
5. They need explicit declaration of uses while we infer them [10].

An example of Safe program using the combination $(1, 2, 3)$ is the following:

$$\mathbf{let}\ x = z : [\,]\ \mathbf{in\ case!}\ x\ \mathbf{of} \ldots \mathbf{case}\ z\ \mathbf{of} \ldots$$

Variable x is destroyed, but a sharing variable z can be read both in the auxiliary and in the main expression. An example of Safe program using the combination $(1, 2, 2)$ is the following:

$$\mathbf{let}\ x = z : [\,]\ \mathbf{in\ case!}\ x\ \mathbf{of} \ldots z$$

Here, the result z shares the destroyed variable x. Both programs take profit from the fact that the sharing variable z is not a recursive descendant of x. Our type system assigns an s-type to these variables.

8. Conclusions and Future Work

We have presented a destruction-aware type system for a functional language with regions and explicit destruction and proved it correct, in the sense that the live heap will never contain dangling pointers. The compiler's front-end, including all the analyses mentioned in this paper —region inference, sharing analysis, and safe types inference— is fully implemented[2] and, by using it, we have successfully typed a significant number of small examples. We are currently working on the space consumption analysis. Preliminary work on a previously needed termination analysis has been reported in [9].

We are also working in the code generation and certification phases, trying to express the correctness proofs of our analyses as certificates which could be mechanically proof-checked by the proof assistant Isabelle [16].

Longer term work include the extension of the language and of the analyses to higher-order. We have also found examples where the inclusion of polymorphic recursion in regions is useful, even under the constructors constraint on recursive children. Finally, we plan to study how the type system scales to larger programs.

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 1995 conference on Programming language design and implementation, PLDI'95*, pages 174–185. ACM Press, 1995.

[2] D. Aspinall and M. Hofmann. Another Type System for in-place Updating. In *ESOP'02, LNCS 2305*, pages 36–52. Springer-Verlag, 2002.

[3] D. Aspinall, M. Hofmann, and M. Konečný. A type system with usage aspects. *Journal of Functional Programming*, 18(2):141–178, 2008.

[4] L. Birkedal, M. Tofte, and M. Vejlstrup. From region inference to von neumann machines via region representation inference. In

[2] The front-end is now about 5 000 Haskell lines long.

Conference Record of POPL '96: The $23^{\rm rd}$ *ACM SIGPLAN-SIGACT*, pages 171–183, 1996.

[5] M. Fluet, G. Morrisett, and A. J. Ahmed. Linear regions are all you need. In *Europeam Symposium On Programming, ESOP'06*, pages 7–21, 2006.

[6] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proc. 30th ACM Symp. on Principles of Programming Languages, POPL'03*, pages 185–197. ACM Press, 2003.

[7] R. J. M. Hughes and L. Pareto. Recursion and Dynamic Data-Structures in Bounded Space; Towards Embedded ML Programming. In *Proc. 4th ACM Int. Conf. on Functional Programming, ICFP'99*, ACM Sigplan Notices, pages 70–81, Paris, France, September 1999. ACM Press.

[8] N. Kobayashi. Quasi-linear Types. In *POPL'99*, pages 29–42. ACM, 1999.

[9] S. Lucas and R. Peña. Termination and Complexity Bounds for SAFE Programs. In *Proc. 19th Int. Symp. on Implementation and Application of Functional Languages, IFL'07, Freiburg, Sept. 2007*, pages 8–23, 2007.

[10] M. Montenegro, R. Peña, and C. Segura. An Inference Algorithm for Guaranteeing Safe Destruction. In *Proc. 8th Symp. on Trends in Functional Programming, TFP'07. New York, April 2007*, pages XIV–1–16, 2007.

[11] M. Montenegro, R. Peña, and C. Segura. A Resource-Aware Semantics and Abstract Machine for a Functional Language with Explicit Deallocation, 2008. Submitted to 17th Int'l Workshop on Functional and (Constraint) Logic Programming, Siena, Italy July, 2008.

[12] M. Montenegro, R. Peña, and C. Segura. A Simple Region Inference Algorithm for a First-Order Functional Language. In *Ninth Symposium on Trends in Functional Programming, TFP'08, Nijmegen, The Netherlands, May. 2008*, pages XV–1–15, 2008.

[13] M. Montenegro, R. Peña, and C. Segura. A type system for safe memory management and its proof of correctness. Technical report, Dpto. de Sistemas Informticos y Computación. Universidad Complutense de Madrid, 2008. Technical Report SIC-5-08.

[14] G. C. Necula. Proof-Carrying Code. In *Conference Record of POPL'97: The 24TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119. ACM SIGACT and SIGPLAN, ACM Press, 1997.

[15] G. C. Necula and P. Lee. The Design and Implementation of a Certifying Compiler. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, pages 333–344, 1998.

[16] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. Number 2283 in LNCS. Springer, 2002.

[17] M. Odersky. Observers for Linear Types. In *ESOP'92, LNCS 582*, pages 390–407. Springer-Verlag, 1992.

[18] R. Peña, C. Segura, and M. Montenegro. A Sharing Analysis for SAFE. In *Selected Papers of the 7th Symp. on Trends in Functional Programming, TFP'06.*, pages 109–128. Intellect, 2007.

[19] 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.

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

[21] P. Wadler. Linear types can change the world! In *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 561–581. North Holland, 1990.