Certified Absence of Dangling Pointers in a Language with Explicit Deallocation *

Javier de Dios, Manuel Montenegro, and Ricardo Peña

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid
jdcastro@aventia.com, montenegro@fdi.ucm.es, ricardo@sip.ucm.es

Abstract. *Safe* is a first-order eager functional language with facilities for programmer controlled destruction of data structures. It provides also *regions*, i.e. disjoint parts of the heap, where the program allocates data structures, so that the runtime system does not need a garbage collector. A region is a collection of cells, each one big enough to allocate a data constructor. Deallocating cells or regions may create dangling pointers. The language is aimed at inferring and certifying memory safety properties in a Proof Carrying Code like environment. Some of its analyses have been presented elsewhere. The one relevant to this paper is a type system and a type inference algorithm guaranteeing that well-typed programs will be free of dangling pointers at runtime.
Here we present how to generate formal certificates about the absence of dangling pointers property inferred by the compiler. The certificates are Isabelle/HOL proof scripts which can be proof-checked by this tool when loaded with a database of previously proved theorems. The key idea is proving an Isabelle/HOL theorem for each syntactic construction of the language, relating the static types inferred by the compiler to the dynamic properties about the heap that will be satisfied at runtime.
keywords: Memory management, type-based analysis, formal certificates, proof assistants.

1 Introduction

Certifying program properties consists of providing mathematical evidence about them. In a Proof Carrying Code (PCC) environment [14], these proofs should be checked by an appropriate tool. The certified properties may be obtained either manually, interactively, or automatically, but whatever is the effort needed for generating them, the PCC paradigm insists on their checking to be fully automatic.

In our setting, the certified property (absence of dangling pointers) is automatically inferred as the product of several static analyses, so that the certificate can be generated by the compiler without any human intervention. Certifying the inferred property is needed in our case to convince a potential consumer that

* Work supported by the projects TIN2008-06622-C03-01/TIN (STAMP), S2009/TIC-1465 (PROMETIDOS), and MEC FPU grant AP2006-02154.

the static analyses are sound and that they have been correctly implemented in the compiler.

Our functional language *Safe*, described below, is equipped with type-based analyses for inferring *regions* where data structures are located [13], and for detecting when a program with explicit deallocation actions is free of dangling pointers [12]. One may wonder why a functional language with explicit deallocation may be useful and why not using a more conventional one such as e.g. C. Explicit deallocation is a low-level facility which, when used without restrictions, may create complex heap structures and programs difficult or impossible to analyse for pointer safety. On the contrary, functional languages have more structure and the explicit deallocation can be confined to a small part of it (in our case, to pattern matching), resulting in heap-safe programs most of the time and, more importantly, amenable to a safety analysis in an automatic way.

Region inference was proved optimal in [13]: assigning data to regions minimises their lifetimes, subject to allocating/deallocating regions in a stack-based way. On the other hand, explicit deallocation eliminates garbage before the region mechanism does, so memory leaks are not a major concern here.

The above analyses have been manually proved correct in [11], but we embarked on the certification task by several reasons:

- The proof in [11] was very much involved. There were some subtleties that we wanted to have formally verified in a proof assistant.
- The implementation was also very involved. Generating and checking certificates is also a way of increasing our trust in the implementation.
- A certificate is a different matter than proving analyses correct, since the proof it contains must be related to every specific compiled program.

In this paper we describe how to create a certificate from the type annotations inferred by the analyses. The key idea is creating a database of theorems, proved once forever, relating these static annotations to the dynamic properties the compiled programs are expected to satisfy. There is a *proof rule* for each syntactic construction of the language and a theorem proving its soundness. These proof-rules generate *proof obligations*, which the generated certificate must discharge. We have chosen the proof assistant Isabelle/HOL [16] both for constructing and checking proofs. To the best of our knowledge, this is the first system certifying absence of dangling pointers in an automatic way.

The certificates are produced at the intermediate language level called *Core-Safe*, at which also the analyses are carried out. This deviates a bit from the standard PCC paradigm where certificates are at the bytecode/assembly language level, i.e. they certify properties satisfied by the executable code. We chose instead to formally verify the compiler's back-end: *Core-Safe* is translated in two steps to the bytecode language of the Java Virtual Machine, and these steps have been verified in Isabelle/HOL [7, 6], so that the certified property is preserved across compilation. This has saved us the (huge) effort of translating *Core-Safe* certificates to the JVM level, while nothing essential is lost: a scenario can be imagined where the *Core-Safe* code and its certificate are sent from the producer

to a consumer and, once validated, the consumer uses the certified back-end for generating the executable code. On the other hand, our certificates are smaller than the ones which could be obtained at the executable code level.

In the next section we describe the relevant aspects of *Safe*. In Sec. 3 a first set of proof rules related to explicit deallocation is presented, while a second set related to implicit region deallocation is explained in Sec. 4. Section 5 is devoted to certificate generation and Sec. 6 presents related work and concludes.

2 The language

Safe is a first-order eager language with a syntax similar to Haskell's. Its runtime system uses *regions*, i.e. disjoint parts of the heap where the program allocates data structures. The smallest memory unit is the *cell*, a contiguous memory space big enough to hold a 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. Each cell is allocated at constructor application time. A *region* is a collection of cells. It is created empty and it may grow and shrink while it is active. Region deallocation frees all its cells. The allocation and deallocation of regions is bound to function calls. A *working region*, denoted by *self*, is allocated when entering the call and deallocated when exiting it. Inside the function, data structures not belonging to the output may be built there. The region arguments are explicit in the intermediate code but not in the source, since they are inferred by the compiler [13]. The following list sorting function builds an intermediate tree not needed in the output:

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

After region inference, the code is annotated with region arguments (those ocurring after the `@`):

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

so that the tree is created in `treesort`'s *self* region and deallocated upon termination of `treesort`.

Besides regions, destruction facilities are associated with pattern matching. For instance, we show here a destructive function splitting a list into two:

```
unshuffle []!     = ([],[])
unshuffle (x:xs)! = (x:xs2,xs1) where (xs1,xs2) = unshuffle xs
```

The ! mark is the way programmers indicate that the matched cell must be deleted. The space consumption is reduced with respect to a conventional version because, at each recursive call, a cell is deleted by the pattern matching. At termination, the whole input list has been returned to the runtime system.

The *Safe* front-end desugars *Full-Safe* and produces a bare-bones functional language called *Core-Safe*. The transformation starts with region inference and continues with Hindley-Milner type inference, pattern matching desugaring, and some other simplifications. In Fig. 1 we show the syntax of *Core-Safe*. A program

$$
\begin{array}{lll}
prog \to \overline{data_i}^{\,n}\,;\,\overline{dec_j}^{\,m}\,;\,e & & \{\textit{Core-Safe}\text{ program}\}\\
data \to \textbf{data}\ T\ \overline{\alpha_i}^{\,n}\ @\ \overline{\rho_j}^{\,m} = \overline{C_k\ \overline{t_{ks}}^{\,n_k}\ @\ \rho_m}^{\,l} & & \{\text{recursive, polymorphic data type}\}\\
dec \to f\ \overline{x_i}^{\,n}\ @\ \overline{r_j}^{\,l} = e & & \{\text{recursive, polymorphic function}\}\\
e \to a & & \{\text{atom: literal } c \text{ or variable } x\}\\
\quad \mid x\ @\ r & & \{\text{copy data structure } x \text{ into region } r\}\\
\quad \mid x! & & \{\text{reuse data structure } x\}\\
\quad \mid a_1 \oplus a_2 & & \{\text{primitive operator application}\}\\
\quad \mid f\ \overline{a_i}^{\,n}\ @\ \overline{r_j}^{\,l} & & \{\text{function application}\}\\
\quad \mid \textbf{let}\ x_1 = be\ \textbf{in}\ e & & \{\text{non-recursive, monomorphic}\}\\
\quad \mid \textbf{case}\ x\ \textbf{of}\ \overline{alt_i}^{\,n} & & \{\text{read-only case}\}\\
\quad \mid \textbf{case!}\ x\ \textbf{of}\ \overline{alt_i}^{\,n} & & \{\text{destructive case}\}\\
alt \to C\ \overline{x_i}^{\,n} \to e & & \{\text{case alternative}\}\\
be \to C\ \overline{a_i}^{\,n}\ @\ r & & \{\text{constructor application}\}\\
\quad \mid e & &
\end{array}
$$

Fig. 1. *Core-Safe* syntax

is a sequence of possibly recursive polymorphic data and function definitions followed by a main expression e whose value is the program result. The over-line abbreviation $\overline{x_i}^{\,n}$ stands for $x_1 \cdots x_n$. **case!** expressions implement destructive pattern matching, constructions are only allowed in **let** bindings, and atoms —or just variables— are used in function applications, **case**/**case!** discriminant, copy and reuse. Region arguments are explicit in constructor and function applications and in copy expressions. As an example, we show the *Core-Safe* version of the `unshuffle` function above:

```
unshuffle x34 @ r1 r2 r3 = case! x34 of
          x49 : x50 -> let x40 = unshuffle x50 @ r2 r1 self in
                       let x15 = case x40 of (x45,x46) -> x45 in
                       let x16 = case x40 of (x47,x48) -> x48 in
                       let x38 = x49 : x16 @ r1 in
                       let x39 = (x38,x15) @ r3 in x39
          []        -> let x36 = [] @ r1 in
                       let x35 = [] @ r2 in
                       let x37 = (x36,x35) @ r3 in x37
```

2.1 Operational Semantics

In Figure 2 we show the big-step operational semantics rules of the most relevant core language expressions. We use $v, v_i, \ldots$ to denote either heap pointers or basic constants, $p, p_i, q, \ldots$ to denote heap pointers, and $a, a_i, \ldots$ to denote either program variables or basic constants (atoms). The former are named $x, x_i, \ldots$ and the latter c, c_i etc. Finally, we use $r, r_i, \ldots$ to denote region arguments.

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

$$E \vdash (h,k), c \Downarrow (h,k), c \;\; [Lit] \qquad E[x \mapsto v] \vdash (h,k), x \Downarrow (h,k), v \;\; [Var_1]$$

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

$$\frac{E \vdash (h,k), e_1 \Downarrow (h',k), v_1 \qquad E \cup [x_1 \mapsto v_1] \vdash (h',k), e_2 \Downarrow (h'',k), v}{E \vdash (h,k), \mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2 \Downarrow (h'',k), v}\ [Let]$$

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

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

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

Fig. 2. Operational semantics of *Safe* expressions

values and region variables to actual region numbers in the range $\{0 \ldots k\}$. We adopt the convention that for all E, if c is a constant, $E(c) = c$.

In a heap (h, k), h is a finite mapping from pointers p to construction cells w of the form $(j, C\ \overline{v_i}^n)$, where $j \le k$, meaning that the cell resides in a region j in scope. By $h[p \mapsto w]$ we denote a heap mapping h where the binding $[p \mapsto w]$ exists and is highlighted, $h \uplus [p \mapsto w]$ denotes the disjoint union of h and $[p \mapsto w]$, and $(h\ |_k, k)$ is the heap obtained by deleting from (h, k') the bindings living in regions greater than k.

The semantics of a program $dec_1; \ldots; dec_n; e$ is the semantics of the main expression e in an environment Σ containing all the function declarations. We only comment the rules related to allocation/deallocation, which may create dangling pointers in the heap. The rest are the usual ones for an eager language.

Rule *App* shows when a new region is allocated. The formal identifier *self* is bound to the newly created region $k + 1$ so that the function body may create bindings in this region. Before returning, all cells created in region $k + 1$ are deleted. This action is a source of possible dangling pointers. Rule *Case*! expresses what happens in a destructive pattern matching: the binding of the discriminant variable disappears from the heap. This action is another source of possible dangling pointers.

2.2 Safe Type System

We distinguish between functional and non-functional types. Non-functional algebraic types may be safe types (internally marked as s), condemned types (marked as d), or in-danger types (marked as r). In-danger types arise as an intermediate step during typing and are useful to control the side-effects of destructions, but function arguments can only receive either safe or condemned types. The intended semantics of these types is the following:

- ***Safe* types (s):** Data structures (DS) of this type can be read, copied or used to build other DSs. They cannot be destroyed.
- ***Condemned* types (d):** A DS directly involved in a **case**! action. Its recursive descendants inherit a condemned type. They cannot be used to build other DSs, but they can be read/copied before being destroyed.
- ***In-danger* types (r):** A DS sharing a recursive descendant of a condemned DS, so it can potentially contain dangling pointers.

Functional types can be polymorphic both in the Hindley-Milner sense and in the region sense: they may contain polymorphic type variables (denoted $\rho, \rho' \ldots$) representing regions. If a region type variable occurs several times in a type, then the actual runtime regions of the corresponding arguments should be the same. Constructor applications have one region argument $r : \rho$ whose type occurs as the outermost region in the resulting algebraic type $T\ \overline{s}\ @\ \overline{\rho}^m$ (i.e. $\rho_m = \rho$). Constructors are given types forcing its recursive substructures and the whole structure to live in the same region. For example, for lists and trees:

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

Function types may have zero or more region arguments. For instance, the type inferred for `unshuffle` is:

$$\forall a\ \rho_1\ \rho_2\ \rho_3\ \rho_4\,.\,[a]! \ @\ \rho_4 \to \rho_1 \to \rho_2 \to \rho_3 \to ([a]\ @\ \rho_1, [a]\ @\ \rho_2)\ @\ \rho_3$$

where ! is the external mark of a condemned type (internal mark d). Types without external marks are assumed to be safe.

The constructor types are collected in an environment Σ, easily built from the **data** type declarations. In typing environments Γ we can find region type assumptions $r : \rho$, variable type assumptions $x : t$, and polymorphic scheme assumptions for function symbols $f : \forall\overline{a}\forall\overline{\rho}.t$. The operators between typing environments used in the typing rules are shown in Fig. 3. 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 $\oplus$. Operator $\rhd^L$ is an asymmetric composition used to type **let** expressions. Predicate $\mathit{utype}?(t, t')$ tells whether the underlying types (i.e. without marks) of t and t' are the same, while $\mathit{unsafe}?$ is true for types with a mark r or d.

In Fig. 4 we show the most relevant rules of the type system, which illustrate the use of the above environment operators. Function $\mathit{sharerec}(x, e)$ is the result of a sharing analysis and returns the set of free variables in the scope of expression e which at runtime may share a recursive descendant of variable x. An important consequence of having a sharing analysis is the unusual feature of our type system that the typing environment may contain type assumptions for variables which are not free in the typed expression.

Predicates inh and $\mathit{inh}!$ restrict the types of the **case**/**case**! patterns according to the type of the discriminant. The most important restriction is that

Operator	$\Gamma_1 \bullet \Gamma_2$ **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

Fig. 3. Operators on type environments

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

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

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

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

Fig. 4. Some *Safe* typing rules for expressions

the recursive patterns of a condemned discriminant must also be condemned. Predicate *danger*? is true for r-marked types, while function $danger(t)$ attaches a mark r to a safe type t. For a complete description, see [11]. An inference algorithm for this type system has been developed in [12].

3 Cell deallocation by destructive pattern matching

The idea of the certificate is to ask the compiler to deliver some static information inferred during the type inference phase, and then to use a database of previously proved lemmas relating this information with the dynamic properties the program is expected to satisfy at runtime. In this case, the static information consists of a mark $m \in \{s, r, d\}$ —respectively meaning *safe, in-danger*, and *condemned* type— for every variable, and the dynamic property the certificate must prove is that the heap remains closed during evaluation.

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 $\{p \mid [p \mapsto w] \in h\}$. A

static assertion has the form $[\![L, \Gamma]\!]$, where $L \subseteq dom(\Gamma)$ is a set of program variables and Γ a mark environment assigning a mark to each variable in L and possibly to some other variables. We will write $\Gamma[x] = m$ to indicate that x has mark $m \in \{s, r, d\}$ in Γ. We say that a *Core-Safe* expression e satisfies a static assertion, denoted $e : [\![L, \Gamma]\!]$, if $fv(e) \subseteq L$ and some semantic conditions below hold. Our certificate for a given program consists of proving a static assertion $[\![L, \Gamma]\!]$ for each *Core-Safe* expression e resulting from compiling the program.

If E is the runtime environment, the intuitive idea of a variable x being typed with a safe mark s is that all the cells in the heap h reached at runtime by $E(x)$ do not contain dangling pointers and they are disjoint from unsafe cells. The idea behind a condemned variable x is that the cell pointed to by $E(x)$ will be removed from the heap and all live cells reaching any of $E(x)$'s recursive descendants by following a pointer chain are in danger. We use the following definitions, formally specified in Isabelle/HOL:

$closure\ (E, X, h)$	Set of locations reachable in heap h by $\{E(x) \mid x \in X\}$
$closure\ (v, h)$	Set of locations reachable in h by location v
$recReach\ (E, x, h)$	Set of recursive descendants of $E(x)$ including itself
$recReach\ (v, h)$	Set of recursive descendants of v in h including itself
$closed\ (E, L, h)$	There are no dangling pointers in h, i.e. $closure\ (E, L, h) \subseteq dom(h)$
$p \rightarrow_h^* V$	There is a pointer path in h from p to a $q \in V$

By abuse of notation, we will write $closure(E, x, h)$ and also $closed(v, h)$. Now, we define the following two sets, respectively denoting the safe and unsafe locations of the live heap, as functions of L, Γ, E, and h:

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

Definition 1. *Given the following properties*

$P1 \equiv E \vdash (h, k), e \Downarrow (h', k), v$
$P2 \equiv dom(\Gamma) \subseteq dom(E)$
$P3 \equiv L \subseteq dom(\Gamma)$
$P4 \equiv fv(e) \subseteq L$
$P5 \equiv \forall x \in dom(E).\ \forall z \in L\ .$
$\quad \Gamma[z] = d\ \wedge\ recReach(E, z, h) \cap closure(E, x, h) \neq \emptyset \rightarrow x \in dom(\Gamma) \wedge \Gamma[x] \neq s$
$P6 \equiv \forall x \in dom(E)\ .\ closure\ (E, x, h) \not\equiv closure\ (E, x, h') \rightarrow x \in dom(\Gamma) \wedge \Gamma[x] \neq s$
$P7 \equiv S_{L,\Gamma,E,h} \cap R_{L,\Gamma,E,h} = \emptyset$
$P8 \equiv closed(E, L, h)$
$P9 \equiv closed(v, h')$

we say that expression e satisfies the static assertion $[\![L, \Gamma]\!]$*, denoted* $e : [\![L, \Gamma]\!]$*, if* $P3 \wedge P4 \wedge (\forall E\ h\ k\ h'\ v\ .\ P1 \wedge P2 \rightarrow P5 \wedge P6 \wedge (P7 \wedge P8 \rightarrow P9))$.

A notion of satisfaction relative to the validity of a function environment Σ_M, denoted $e, \Sigma_M : [\![L, \Gamma]\!]$, is also defined.

Property $P1$ defines any runtime evaluation of e. Properties $P2$ to $P4$ just guarantee that each free variable has a type and a value. Properties $P5$ to $P7$

$$c, \Sigma_M \vdash (\emptyset, \emptyset) \quad LIT \qquad x, \Sigma_M \vdash (\{x\}, \Gamma + [x : s]) \quad VAR1$$

$$\frac{e_1 \neq C\ \overline{a_i}^n \quad e_1, \Sigma_M \vdash (L_1, \Gamma_1) \quad x_1 \notin L_1 \quad e_2, \Sigma_M \vdash (L_2, \Gamma_2' + [x_1 : s]) \quad def(\Gamma_1 \triangleright^{L_2} \Gamma_2')}{\textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2, \Sigma_M \vdash (L_1 \cup (L_2 - \{x_1\}), \Gamma_1 \triangleright^{L_2} \Gamma_2')}\ LET1$$

$$\frac{e_1 \neq C\ \overline{a_i}^n \quad e_1, \Sigma_M \vdash (L_1, \Gamma_1) \quad x_1 \notin L_1 \quad e_2, \Sigma_M \vdash (L_2, \Gamma_2' + [x_1 : d]) \quad def(\Gamma_1 \triangleright^{L_2} \Gamma_2')}{\textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2, \Sigma_M \vdash (L_1 \cup (L_2 - \{x_1\}), \Gamma_1 \triangleright^{L_2} \Gamma_2')}\ LET2$$

$$\frac{L_1 = \{\overline{a_i}^n\} \quad \Gamma_1 = [\overline{a_i \mapsto s}^n] \quad x_1 \notin L_1 \quad e_2, \Sigma_M \vdash (L_2, \Gamma_2' + [x_1 : s]) \quad def(\Gamma_1 \triangleright^{L_2} \Gamma_2')}{\textbf{let}\ x_1 = C\ \overline{a_i}^n @ r\ \textbf{in}\ e_2, \Sigma_M \vdash (L_1 \cup (L_2 - \{x_1\}), \Gamma_1 \triangleright^{L_2} \Gamma_2')}\ LET1_C$$

$$\frac{L_1 = \{\overline{a_i}^n\} \quad \Gamma_1 = [\overline{a_i \mapsto s}^n] \quad x_1 \notin L_1 \quad e_2, \Sigma_M \vdash (L_2, \Gamma_2 + [x_1 : d]) \quad def(\Gamma_1 \triangleright^{L_2} \Gamma_2')}{\textbf{let}\ x_1 = C\ \overline{a_i}^n @ r\ \textbf{in}\ e_2, \Sigma_M \vdash (L_1 \cup (L_2 - \{x_1\}), \Gamma_1 \triangleright^{L_2} \Gamma_2')}\ LET2_C$$

$$\frac{\forall i\ .\ (e_i, \Sigma_M \vdash (L_i, \Gamma_i)\ \forall j . \Gamma_i[x_{ij}] \neq d)\ \Gamma \supseteq \bigotimes_i (\Gamma_i \backslash \{\overline{x_{ij}}\})\ x \in dom(\Gamma)\ L = \{x\} \cup (\bigcup_i (L_i - \{\overline{x_{ij}}\}))}{\textbf{case}\ x\ \textbf{of}\ \overline{C_i \overline{x_{ij}} \to e_i}, \Sigma_M \vdash (L, \Gamma)}\ CASE$$

$$\frac{\begin{array}{c}\forall i\ .\ (e_i, \Sigma_M \vdash (L_i, \Gamma_i) \quad \forall j\ .\ \Gamma_i[x_{ij}] = d \rightarrow j \in RecPos(C_i)) \\ L' = \bigcup_i (L_i - \{\overline{x_{ij}}\})\ \Gamma \supseteq (\bigotimes_i \Gamma_i \backslash (\{\overline{x_{ij}}\} \cup \{x\})) + [x : d]\ \forall z \in dom(\Gamma)\ .\ \Gamma[z] \neq s \rightarrow (\forall i\ .\ z \notin L_i)\end{array}}{\textbf{case!}\ x\ \textbf{of}\ \overline{C_i \overline{x_{ij}} \to e_i}, \Sigma_M \vdash (L' \cup \{x\}, \Gamma)}\ CASE!$$

$$\frac{\Sigma_M(g) = \overline{m_i}^n \qquad L = \{\overline{a_i}^n\} \quad \Gamma_0 = \bigoplus_{i=1}^n [a_i : m_i]\ \text{defined} \quad \Gamma \supseteq \Gamma_0}{g\ \overline{a_i}^n @\ \overline{r_j}^m, \Sigma_M \vdash (L, \Gamma)}\ APP$$

$$\frac{f\ \overline{x_i}^n @\ \overline{r_j}^m = e_f \quad L_f = \{\overline{x_i}^n\} \quad \Gamma_f = [\overline{x_i \mapsto m_i}^n] \quad e_f, \Sigma_M \uplus [f \mapsto \overline{m_i}^n] \vdash (L_f, \Gamma_f)}{e_f, \Sigma_M \vdash (L_f, \Gamma_f)}\ REC$$

Fig. 5. Proof rules for explicit deallocation

formalise the dynamic meaning of safe and condemned types: if some variable can share a recursive descendant of a condemned one, or its closure changes during evaluation, it should occur as unsafe in the environment.

The key properties are $P8$ and $P9$. If they were proved for all the judgements of any e's derivation, they would guarantee that the live part of the heap would remain closed, hence there would not be dangling pointers. We have proved that $P8$ is an 'upwards' invariant in any derivation, while $P9$ is a 'downwards' one. Formally:

Theorem 1 (closedness). *Consider a derivation $E \vdash (h, k), e \Downarrow (h', k), v$. If $e : [\![L, \Gamma]\!]$, $P2(\Gamma, E)$, $P7(L, \Gamma, E, h)$ and $P8(E, L, h)$ hold, then $P8(E_i, L_i, h_i)$ and $P9(v_i, h_i')$ hold for all judgements $E_i \vdash (h_i, k_i), e_i \Downarrow (h_i', k_i), v_i$ belonging to that derivation.*

But $P2$, $P7$ and $P8$ trivially hold for the empty heap, empty environment Γ, and empty set L of free variables, which are the ones corresponding to the initial expression, so $P8$ and $P9$ hold across the whole derivation of the program.

In Fig. 5 we show the proof rules related to this property. The following soundness theorem (a lemma for each expression) has been interactively proved by induction on the derivations obtained with these rules.

Theorem 2 (soundness). *If $e, \Sigma_M \vdash (L, \Gamma)$ then $e, \Sigma_M : [\![L, \Gamma]\!]$.*

When proving the soundness of the APP rule, the closedness of the heap before returning from g does not in principle guarantee that the heap will remain closed

after deallocating the heap topmost region. Proving this, needs a separate collection of theorems showing that the value returned by g does not contain cells in that region. This part of the problem is explained in Sec. 4.

For each expression e, the compiler generates a pair (L, Γ). According to e's syntax, the certificate chooses the appropriate proof rule, checks that its premises are satisfied, and applies it in order to get the conclusion $e, \Sigma_M \vdash (L, \Gamma)$. For example, in an application expression $g\ \overline{a_i}^n @\ \overline{r_j}^m$, the certificate access to Σ_M and checks that the given environment Γ contains the actual arguments a_i with these marks m_i assigned. It also checks that operator $\bigoplus$, requiring any duplicated actual argument to be safe (see Fig. 3), is well-defined, and then it applies the proof rule *APP*.

4 Region deallocation

We present here the proof rules certifying that region deallocation does not create dangling pointers. As before, the compiler delivers static information about the region types used by the program variables and expressions, and a soundness theorem relates this information to the runtime properties of the actual regions.

In an algebraic type $T\ \overline{t_i}^m @\ \overline{\rho_j}^l$, the last region type variable ρ_l of the list is always the most external one, i.e. the region where the cell of the most external constructor is allocated. By *regions* (t) we denote the set of region type variables occurring in the type t. There is a reserved identifier ρ^f_{self} for every defined function f, denoting the region type variable assigned to the working region *self* of function f. We will assume that the expression e being certified belongs to the body of a context function f or to the *main* expression.

By $\theta, \theta_i, \ldots$ we denote *typing environments*, i.e. mappings from program variables and region arguments to types. For region arguments, $\theta(r) = \rho$ means that ρ is the type variable the compiler assigns to argument r.

In function or constructor applications, the set of generic region types used in the signature of an applied function g (of a constructor C) must be related to the actual region types used in the application. Also, some ordinary polymorphic type variables of the signature may become instantiated by algebraic types introducing additional regions. Let us denote by μ the *type instantiation mapping* used by the compiler. This mapping should correctly map the types of the formal arguments to the types of the corresponding actual arguments.

Definition 2. *Given the instantiated types $\overline{t_i}^n$, the instantiated region types $\overline{\rho_j}^m$, the arguments of the application $\overline{a_i}^n$, $\overline{r_j}^m$, and the typing mapping θ, we say that the application is* argument preserving*, denoted argP $(\overline{t_i}^n, \overline{\rho_j}^m, \overline{a_i}^n, \overline{r_j}^m, \theta)$, if:* $\forall i \in \{1..n\}\ .\ t_i = \theta(a_i) \wedge \forall j \in \{1..m\}\ .\ \rho_j = \theta(r_j)$.

For functions, the certificate incrementally constructs a global environment Σ_T keeping the most general types of the functions already certified. For constructors, the compiler provides a global environment Γ_T giving its polymorphic most general type. If $\Gamma_T(C) = \overline{t_i}^n \to \rho \to T\ \overline{t_j}^l @\ \overline{\rho_i}^m$, the following property, satisfied by the type system, is needed for proving the proof rules below:

Definition 3. *Predicate* $wellT(\overline{t_i}^n, \rho, T\ \overline{t_j}^l @\ \overline{\rho_i}^m)$*, read* well-typed*, is defined as* $\rho_m = \rho\ \wedge\ \bigcup_{i=1}^n regions\ (t_i) \subseteq regions\ (T\ \overline{t_j}^l @\ \overline{\rho_i}^m)$

So far for the static concepts. We move now to the dynamic or runtime ones. By $\eta,\ \eta_i, \ldots$ we denote *region instantiation mappings* from region type variables to runtime regions identifiers in scope. Region identifiers $k,\ k_i, \ldots$ are just natural numbers denoting offsets of the actual regions from the bottom of the region stack. If k if the topmost region in scope, then for all ρ, $0 \leq \eta(\rho) \leq k$ holds. The intended meaning of $k' = \eta(\rho)$ is that, in a particular execution of the program, the region type ρ has been instantiated to the actual region k'. Admissible region instantiation mappings should map ρ^f_{self} to the topmost region, and other region types to lower regions.

Definition 4. *Assuming that* k *denotes the topmost region of a given heap, we say that the mapping* η *is* admissible*, denoted admissible* (η, k)*, if:*

$$\rho^f_{self} \in dom(\eta) \wedge \eta(\rho^f_{self}) = k \wedge \forall \rho \in dom(\eta) - \{\rho^f_{self}\}\ .\ \eta(\rho) < k$$

The important notion is *consistency* between the static information θ and the dynamic one $E,\ \eta,\ h,\ h'$. Essentially, it tells us that the static region types, its runtime instantiation to actual regions, and the actual regions where the data structures are stored in the heap, do not contradict each other.

Definition 5. *We say that the mappings* $\theta,\ \eta$*, the runtime environment* E*, and the heap* h *are* consistent*, denoted consistent* (θ, η, E, h)*, if:*

1. $\forall x \in dom(E)\ .\ consistent\ (\theta(x), \eta, E(x), h)$ *where:*

$$\begin{array}{lll}
consistent\ (B, \eta, c, h) & = true & \text{-- } B \text{ denotes a basic type} \\
consistent\ (a, \eta, v, h) & = true & \text{-- } a \text{ denotes a type variable} \\
consistent\ (T\ \overline{t'_i}^m @\ \overline{\rho_j}^l, \eta, p, h) & = \exists j\ C\ \overline{v_k}^n\ \mu\ \overline{t_{kC}}^n\ \overline{\rho_{jC}}^l\ .\ h(p) = (j, C\ \overline{v_k}^n) & \\
 & \wedge\ \rho_l \in dom(\eta)\ \wedge\ \eta(\rho_l) = j & \\
 & \wedge\ \Gamma_T(C) = \overline{t_{kC}}^n \rightarrow \rho_{lC} \rightarrow T\ \overline{t'_{iC}}^m @\ \overline{\rho_{jC}}^l & \\
 & \wedge\ \mu(T\ \overline{t'_{iC}}^m @\ \overline{\rho_{jC}}^l) = T\ \overline{t'_i}^m @\ \overline{\rho_j}^l & \\
 & \wedge\ \forall k \in \{1..n\}\ .\ consistent\ (\mu(t_{kC}), \eta, v_k, h)) &
\end{array}$$

2. $\forall r \in dom(E)\ .\ \theta(r) \in dom(\eta)\ \wedge\ E(r) = (\eta \cdot \theta)(r)$
3. $self \in dom(E) \wedge \theta(self) = \rho^f_{self}$

We are ready to define the satisfaction of a *static assertion* relating the static and dynamic properties referred to regions: A judgement of the form $e : [\![\theta, t]\!]$ defines that, if expression e is evaluated with an environment E, a heap (h, k), and an admissible mapping η consistent with θ, then η, the final heap h', and the final value v are consistent with t. Formally:

Definition 6. *An expression* e *satisfies the pair* (θ, t)*, denoted* $e : [\![\theta, t]\!]$ *if*

$$\begin{array}{lll}
\forall E\ h\ k\ h'\ v\ \eta\ .\ E \vdash (h, k), e \Downarrow (h', k), v & & \text{-- } P1 \\
\wedge\ dom(E) \subseteq dom(\theta) & & \text{-- } P2 \\
\wedge\ admissible\ (\eta, k) & & \text{-- } P3 \\
\wedge\ consistent\ (\theta, \eta, E, h) & & \text{-- } P4 \\
\rightarrow consistent\ (t, \eta, v, h') & & \text{-- } P5
\end{array}$$

$$\frac{}{c, \Sigma_T \vdash \theta \rightsquigarrow B}\ LIT \qquad \frac{}{x, \Sigma_T \vdash \theta \rightsquigarrow \theta(x)}\ VAR1 \qquad \frac{e_1, \Sigma_T \vdash \theta \rightsquigarrow t_1 \quad e_2, \Sigma_T \vdash \theta \uplus [x_1 \mapsto t_1] \rightsquigarrow t_2}{\textbf{let } x_1 = e_1 \textbf{ in } e_2, \Sigma_T \vdash \theta \rightsquigarrow t_2}\ LET$$

$$\frac{\Gamma_T(C) = \overline{t_i}^n \to \rho \to t \ \ wellT(\overline{t_i}^n, \rho, t) \ \ e_2, \Sigma_T \vdash \theta \uplus [x_1 \mapsto \mu(t)] \rightsquigarrow t_2 \ \ argP(\overline{\mu(t_i)}^n, \mu(\rho), \overline{a_i}^n, r, \theta)}{\textbf{let } x_1 = C\ \overline{a_i}^n\ @\ r \textbf{ in } e_2, \Sigma_T \vdash \theta \rightsquigarrow t_2}\ LET_C$$

$$\frac{\begin{array}{c}\forall i.\ (\Gamma_T(C_i) = \overline{t_{ij}}^{n_i} \to \rho \to t \quad wellT(\overline{t_{ij}}^{n_i}, \rho, t)) \\ \forall i.\ e_i, \Sigma_T \vdash \theta \uplus [x_{ij} \to \mu(t_{ij})^{n_i}] \rightsquigarrow t' \quad \theta(x) = \mu(t)\end{array}}{\textbf{case } x \textbf{ of } \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n, \Sigma_T \vdash \theta \rightsquigarrow t'}\ CASE$$

$$\frac{\begin{array}{c}\forall i.\ (\Gamma_T(C_i) = \overline{t_{ij}}^{n_i} \to \rho \to t \quad wellT(\overline{t_{ij}}^{n_i}, \rho, t)) \\ \forall i.\ e_i, \Sigma_T \vdash \theta \uplus [x_{ij} \to \mu(t_{ij})^{n_i}] \rightsquigarrow t' \quad \theta(x) = \mu(t)\end{array}}{\textbf{case! } x \textbf{ of } \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n, \Sigma_T \vdash \theta \rightsquigarrow t'}\ CASE!$$

$$\frac{\Sigma(g) = \overline{t_i}^n \to \overline{\rho_j}^m \to t_g \quad \rho^g_{self} \notin regions\ (t_g) \quad argP\ (\overline{\mu(t_i)}^n, \overline{\mu(\rho_j)}^m, \overline{a_i}^n, \overline{r_j}^m, \theta) \quad t = \mu(t_g)}{g\ \overline{a_i}^n @\ \overline{r_j}^m, \Sigma_T \vdash \theta \rightsquigarrow t}\ APP$$

$$\frac{\begin{array}{l}f\ \overline{x_i}^n @\ \overline{r_j}^m = e_f \\ \theta_f = [x_i \mapsto \overline{t_i}^n, \overline{r_j \mapsto \rho_j}^m, self \mapsto \rho_{self}] \quad e_f, \Sigma_T \cup \{f \mapsto \overline{t_i}^n \to \overline{\rho_j}^m \to t_f\} \vdash \theta_f \rightsquigarrow t_f\end{array}}{e_f, \Sigma_T \vdash \theta_f \rightsquigarrow t_f}\ REC$$

Fig. 6. Proof rules for region deallocation

Theorem 3 (consistency). *If* $e : [\![\theta, t]\!]$, $E \vdash (h, k), e \Downarrow (h', k), v$, $P2(E, \theta)$, $P3(\eta, k)$, $P4(\theta, \eta, E, h)$ *hold, then* $P3(\eta_i, k_i)$, $P4(\theta_i, \eta_i, E_i, h_i)$, $P5(t_i, \eta_i, v_i, h'_i)$ *hold for all judgements* $E_i \vdash (h_i, k_i), e_i \Downarrow (h'_i, k_i), v_i$ *belonging to that derivation.*

But $P2$, $P3$ and $P4$ trivially hold for the empty heap h_0, $dom(E_0) = dom(\theta_0) = \{self\}$, $\theta_0(self) = \rho^{main}_{self}$, $k_0 = 0$, and $\eta_0(\rho^{main}_{self}) = E_0(self) = 0$, which are the ones corresponding to the initial expression, so $P3$, $P4$ and $P5$ hold across the whole program derivation. In Fig. 6 we show the proof rules related to regions.

Theorem 4 (soundness). *If* $e, \Sigma_T \vdash \theta \rightsquigarrow t$ *then* $e, \Sigma_T : [\![\theta, t]\!]$.

To prove it, there is a separate Isabelle/HOL theorem for each syntactic form. As we have said, region allocation/deallocation takes place at function call/return. The premise $\rho^g_{self} \notin regions\ (t_g)$ in the APP rule, together with properties $P3$ and $P5$ guarantee that the data structure returned by the function has no cells in the deallocated region corresponding to $\eta(\rho^g_{self})$. So, deallocating this region cannot cause dangling pointers.

For each expression e, the compiler generates a pair (θ, t) (and a μ when needed). According to e's syntax, the certificate applies the corresponding proof rule by previously discharging its premises, then deriving $e \vdash \theta \rightsquigarrow t$. For example, in an application expression, we assume that the most general type of the called function g is kept in the global environment Σ_T. The certificate receives the (θ, t, μ) for this particular application, gets g's signature from Σ_T, checks $t = \mu(t_g)$ and the rest of premises of the APP proof rule, and then applies it.

5 Certificate generation

Given the above sets of already proved theorems, certificate generation for a given program is a rather straightforward task. It consists of traversing the program's abstract syntax tree and producing the following information:

		Expression	L	Γ
e_1	$\stackrel{\text{def}}{=}$	*unshuffle* x_{50} @ r_2 r_1 *self*	$\{x_{50}\}$	$[x_{50} : d,\ x_{34} : r]$
e_2	$\stackrel{\text{def}}{=}$	x_{45}	$\{x_{45}\}$	$[x_{45} : s,\ x_{34} : r]$
e_3	$\stackrel{\text{def}}{=}$	**case** x_{40} **of** $(x_{45}, x_{46}) \rightarrow e_2$	$\{x_{40}\}$	$[x_{40} : s,\ x_{34} : r]$
e_4	$\stackrel{\text{def}}{=}$	x_{48}	$\{x_{48}\}$	$[x_{48} : s,\ x_{34} : r]$
e_5	$\stackrel{\text{def}}{=}$	**case** x_{40} **of** $(x_{47}, x_{48}) \rightarrow e_4$	$\{x_{40}\}$	$[x_{40} : s,\ x_{34} : r]$
e_6	$\stackrel{\text{def}}{=}$	x_{39}	$\{x_{39}\}$	$[x_{39} : s,\ x_{34} : r]$
e_7	$\stackrel{\text{def}}{=}$	**let** $x_{39} = (x_{38}, x_{15})$ @ r_3 **in** e_6	$\{x_{15}, x_{38}\}$	$[x_{15} : s,\ x_{38} : s,\ x_{34} : r]$
e_8	$\stackrel{\text{def}}{=}$	**let** $x_{38} = x_{49} : x_{16}$ @ r_1 **in** e_7	$\{x_{15}, x_{16}, x_{49}\}$	$[x_{15} : s,\ x_{16} : s,\ x_{49} : s,\ x_{34} : r]$
e_9	$\stackrel{\text{def}}{=}$	**let** $x_{16} = e_5$ **in** e_4	$\{x_{15}, x_{40}, x_{49}\}$	$[x_{15} : s,\ x_{40} : s,\ x_{49} : s,\ x_{34} : r]$
e_{10}	$\stackrel{\text{def}}{=}$	**let** $x_{15} = e_3$ **in** e_2	$\{x_{40}, x_{49}\}$	$[x_{40} : s,\ x_{49} : s,\ x_{34} : r]$
e_{11}	$\stackrel{\text{def}}{=}$	**let** $x_{40} = e_1$ **in** e_{10}	$\{x_{49}, x_{50}\}$	$[x_{49} : s,\ x_{50} : d,\ x_{34} : r]$
e_{12}	$\stackrel{\text{def}}{=}$	x_{37}	$\{x_{37}\}$	$[x_{37} : s,\ x_{34} : r]$
e_{13}	$\stackrel{\text{def}}{=}$	**let** $x_{37} = (x_{36}, x_{35})$ @ r_3 **in** e_{12}	$\{x_{35}, x_{36}\}$	$[x_{35} : s,\ x_{36} : s,\ x_{34} : r]$
e_{14}	$\stackrel{\text{def}}{=}$	**let** $x_{35} = [\,]$ @ r_2 **in** e_{13}	$\{x_{36}\}$	$[x_{36} : s,\ x_{34} : r]$
e_{15}	$\stackrel{\text{def}}{=}$	**let** $x_{36} = [\,]$ @ r_1 **in** e_{14}	$\{\,\}$	$[x_{34} : r]$
e_{16}	$\stackrel{\text{def}}{=}$	**case!** x_{34} **of** $\{x_{49} : x_{50} \rightarrow e_{11}; [\,] \rightarrow e_{15}\}$	$\{x_{34}\}$	$[x_{34} : d]$

Fig. 7. Isabelle/HOL definitions of *Core-Safe* expressions, free variables, and mark environments for `unshuffle`

- A definition in Isabelle/HOL of the abstract syntax tree.
- A set of Isabelle/HOL definitions for the static objects inferred by the analyses: sets of free variables, mark environments, typing environments, type instantiation mappings, etc.
- A set of Isabelle/HOL proof scripts proving a lemma for each expression, consisting of first checking the premises of the proof-rule associated to the syntactic form of the expression, and then applying the proof rule.

This strategy results in small certificates and short checking times as the total amount of work is linear with program size. The heaviest part of the proof —the database of proved proof rules— has been done in advance and is reused by each certified program.

In Fig. 7 we show the Isabelle/HOL definitions for the elementary *Core-Safe* expressions of the `unshuffle` function defined in Sec. 2, together with the components L and Γ of the static assertions proving the absence of dangling pointers for cell deallocation. They are arranged bottom-up, from simple to compound expressions, because this is the order required by Isabelle/HOL for applying the proof rules. In Fig. 8 we show (this time top-down for a better understanding) the components θ, t, and μ of the static assertions for region deallocation for the expressions e_{14}, e_{15}, and e_{16} of Fig. 7. We show also the most general types of some constructors given by the global environment Γ_T.

The *Core-Safe* text for `unshuffle` consists of about 50 lines, while the certificate for it is about 1000 lines long, 300 of which are devoted to definitions. This expansion factor of 20 is approximately the same for all the examples we have certified so far, so confirming that certificate size grows linearly with program

$$\theta_{16} \stackrel{\text{def}}{=} [x_{34} : [a]@\rho_4,\ r_1 : \rho_1,\ r_2 : \rho_2,\ r_3 : \rho_3, self : \rho_{self}] \qquad t_{16} \stackrel{\text{def}}{=} ([a]@\rho_1, [a]@\rho_2)@\rho_3$$
$$\theta_{15} \stackrel{\text{def}}{=} \theta_{16} \qquad t_{15} \stackrel{\text{def}}{=} ([a]@\rho_1, [a]@\rho_2)@\rho_3$$
$$\theta_{14} \stackrel{\text{def}}{=} \theta_{15} + [x_{36} : [a]@\rho_1] \qquad t_{14} \stackrel{\text{def}}{=} ([a]@\rho_1, [a]@\rho_2)@\rho_3$$

$$\mu_{16} \stackrel{\text{def}}{=} \{a \mapsto a,\ \rho_1 \mapsto \rho_4\} \qquad \Gamma_T([\,]) = \rho_1 \to [a]@\rho_1$$
$$\mu_{15} \stackrel{\text{def}}{=} \{a \mapsto a,\ \rho_1 \mapsto \rho_1\} \qquad \Gamma_T(:) = a \to [a]@\rho_1 \to \rho_1 \to [a]@\rho_1$$
$$\mu_{14} \stackrel{\text{def}}{=} \{a \mapsto a,\ \rho_1 \mapsto \rho_2\}$$

Fig. 8. Isabelle/HOL definitions of typing mappings, and types for `unshuffle`

size. There is room for optimisation by defining an Isabelle/HOL tactic for each proof rule. This reduces both the size and the checking time of the certificate. We have implemented this idea in the region deallocation part.

The Isabelle/HOL proof scripts for the cell deallocation proof-rules reach 8 000 lines, while the ones devoted to region deallocation tally up to 4 000 lines more. Together they represent about 1.5 person-year effort. All the theories are available at `http://dalila.sip.ucm.es/safe/certifdangling`. There is also an online version of the *Safe* compiler at `http://dalila.sip.ucm.es/~safe` where users may remotely submit source files and browse all the generated intermediate files, including certificates. An extended version of this paper with proof schemes available can be found at `http://dalila.sip.ucm.es/safe`.

6 Related work and conclusion

Introducing pointers in a Hoare-style assertion logic and using a proof assistant for proving pointer programs goes back to the late seventies [9], where the Stanford Pascal Program Verifier was used. A more recent reference is [5], using the Jape proof editor. A formalisation of Bornat's ideas in Isabelle/HOL was done by Mehta and Nipkow in [10], where they add a complete soundness proof.

A type system allowing safe heap destruction was studied in [1] and [2]. In [11] we made a detailed comparison with those works showing that our system accepts as safe some programs that their system rejects. Another difference is that we have developed a type inference algorithm [12] which they lack.

Connecting the results of a static analysis with the generation of certificates was done from the beginning of the PCC paradigm (see for instance [15]). A more recent work is [3].

Our work is more closely related to [4], where a resource consumption property obtained by a special type system developed in [8] is transformed into a certificate. The compiler is able to infer a linear upper bound on heap consumption and to certify this property by emitting an Isabelle/HOL script proving it. Our static assertions have been inspired by their *derived assertions*, used also there to connect static with dynamic properties. However, their heap is simpler to deal with than ours since it essentially consists of a free list of cells, and the only data type available is the list. We must also deal with regions and with any user-defined data type. This results in our complex notion of consistency.

Apart from the proofs themselves, our contribution has been defining the appropriate functions, predicates and relations such as *closure, recReach, closed, consistent,...* relating the static and the runtime information in such a way that the proof-rules could be proved correct.

References

1. D. Aspinall and M. Hofmann. Another Type System for In-Place Update. In *Proceedings of the 11th European Sysmposium on Programming, ESOP'02*, volume 2305 of *LNCS*, pages 36–52, 2002.
2. D. Aspinall, M. Hofmann, and M. Konečný. A Type System with Usage Aspects. *Journal of Functional Programming*, 18(2):141–178, 2008.
3. G. Barthe, B. Grégoire, C. Kunz, and T. Rezk. Certificate Translation for Optimizing Compilers. In *SAS'06, LNCS 4134*, pages 301–317. Springer, 2006.
4. L. Beringer, M. Hofmann, A. Momigliano, and O. Shkaravska. Automatic Certification of Heap Consumption. In *LPAR'04, LNAI 3452*, pages 347–362, 2005.
5. R. Bornat. Proving Pointer Programs in Hoare Logic. In *Proc. 5th Int. Conf. Mathematics of Program Construction, MPC'00*, pages 102–126. Springer, 2000.
6. J. de Dios and R. Peña. A Certified Implementation on top of the Java Virtual Machine. In *Formal Method in Industrial Critical Systems, FMICS'09, Eindhoven (The Netherlands)*, pages 181–196. LNCS 5825, Springer, November 2009.
7. J. de Dios and R. Peña. Formal Certification of a Resource-Aware Language Implementation. In *Int. Conf. on Theorem Proving in Higher Order Logics, TPHOL'09, Munich (Germany)*, pages 196–211. LNCS 5674, Springer, August 2009.
8. 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.
9. D. C. Luckham and N. Suzuki. Verification of array, record and pointer operations in Pascal. *ACM Trans. on Prog. Lang. and Systems*, 1(2):226–244, Oct. 1979.
10. F. Mehta and T. Nipkow. Proving Pointer Programs in Higher-Order Logic. In *Automated Deduction CADE-19*, pages 121–135. LNCS 2741, Springer, 2003.
11. M. Montenegro, R. Peña, and C. Segura. A Type System for Safe Memory Management and its Proof of Correctness. In *ACM Principles and Practice of Declarative Programming, PPDP'08, Valencia, Spain, July. 2008*, pages 152–162, 2008.
12. M. Montenegro, R. Peña, and C. Segura. An Inference Algorithm for Guaranteeing Safe Destruction. In *Selected papers of Logic-Based Program Synthesis and Transformation, LOPSTR'08, LNCS 5438, Springer*, pages 135–151, 2009.
13. M. Montenegro, R. Peña, and C. Segura. A simple region inference algorithm for a first-order functional language. In S. Escobar, editor, *Work. on Functional and Logic Programming, WFLP 2009, Brasilia*, pages 145–161. LNCS 5979, 2009.
14. G. C. Necula. Proof-Carrying Code. In *ACM SIGPLAN-SIGACT Principles of Programming Languages, POPL'97*, pages 106–119. ACM Press, 1997.
15. G. C. Necula and P. Lee. Safe Kernel Extensions Without Run-Time Checking. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 229–243, Seattle, Washington, oct 1996.
16. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic.* LNCS 2283. Springer, 2002.