Certified Absence of Dangling Pointers in a Language with Explicit Deallocation

Javier de Dios[1,2] Ricardo Peña[1,3] Manuel Montenegro[1,4]

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

Abstract

Safe is a first-order eager functional 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. A region is a collection of cells, each one is big enough to allocate a data constructor. The runtime system does not need a garbage collector and all allocation/deallocation actions are done in constant time. Deallocating cells or regions may create dangling pointers. The language is aimed at inferring and certifying memory safety properties in a Proof Carrying Code 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.
In this paper we present how to generate formal certificates of 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 [7], these proofs should be automatically checked by an appropriate tool. In our language *Safe*, we have chosen the proof assistant Isabelle/HOL [9] both for constructing and checking proofs.

Safe, described below, is equipped with analyses for inferring *regions* where data structures are located [4], and for detecting when a program with explicit deallocation actions is free of dangling pointers [6]. These analyses have been manually proved correct, but a certificate is a different matter than proving analyses correct:

- The proof it contains must be related to a specific program.

[1] Work supported by the projects TIN2008-06622-C03-01/TIN (STAMP), S-0505/TIC/0407 (PROMESAS)
[2] Email: jdcastro@aventia.com.
[3] Email: ricardo@sip.ucm.es
[4] Email: montenegro@fdi.ucm.es. Work supported by the MEC FPU grant AP2006-02154.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

- The proof must be automatically validated by a proof checker.

In this paper we describe how to create a certificate from the properties inferred by the analyses. The key idea is creating a database of theorems, proved once forever, relating these static properties to the dynamic ones the compiled programs are expected to satisfy. There is one such theorem for each syntactic construction of the language. Then, these theorems are considered as *proof obligations* which the generated certificate must discharge.

In the rest of this section we describe the relevant aspects of *Safe*. In Sec. 2 a first set of proof obligations related to explicit deallocation is presented, while a second set related to implicit region deallocation is explained in Sec. 3. Sec. 4 is devoted to certificate generation and Sec. 5 concludes.

1.1 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 [4]. 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:

```
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 to pattern matching. For instance, we show here a constant space function appending two lists:

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

The ! mark is the way programmers indicate that the matched cell must be deleted. The constant space consumption is due to that, at each recursive call, a cell is deleted by the pattern matching while a new one is allocated by the (:).

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 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

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

Fig. 1. *Core-Safe* syntax

$\overline{x_i}^n$ 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 above `append` function:

$$
\begin{array}{lll}
append\ xs\ ys\ @\ r = \textbf{case!}\ xs\ \textbf{of} & & \\
\qquad\qquad [\,] & \to & ys \\
\qquad\qquad x : xx & \to & \textbf{let}\ yy = append\ xx\ ys\ @\ r\ \textbf{in} \\
\qquad\qquad & & \textbf{let}\ zz = x : yy\ @\ r\ \textbf{in}\ zz
\end{array}
$$

1.2 Operational Semantics

In Figure 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, $p, p_i, q, \ldots$ 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 variables.

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 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, $E(c) = c$.

A heap h is a finite mapping from pointers p to construction cells w of the form $(j, C\ \overline{v_i}^n)$, meaning that the cell resides in region j. 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 the heap h and the binding $[p \mapsto w]$, while $h\ |_k$ is the heap obtained by deleting from h the 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 function declarations. We only comment the rules related to allocation/deallocation actions, some of which may create dangling pointers in the heap. The rest are the usual ones for an eager language.

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

$$\frac{j \leq k \quad (h', p') = copy(h, p, j)}{E[x \mapsto p, r \mapsto j] \vdash h, k, x@r \Downarrow h', k, p'} \; [Var_2] \qquad \frac{fresh(q)}{E[x \mapsto p] \vdash h \uplus [p \mapsto w], k, x! \Downarrow h \uplus [q \mapsto w], k, q} \; [Var_3]$$

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

$$\frac{op_{\oplus}\ v_1\ v_2 = v}{E[a_1 \mapsto v_1, a_2 \mapsto v_2] \vdash h, k, a_1 \oplus a_2 \Downarrow h, k, v} \; [Primop]$$

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

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

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

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

Fig. 2. Operational semantics of *Safe* expressions

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

Fig. 3. Type expressions

Rule Var_2 executes a copy expression copying the data structure pointed to by p and living in 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. The non-recursive cells are shared with the old structure. In rule Var_3, the binding $[p \mapsto w]$ is deleted and a fresh binding $[q \mapsto w]$ to cell w is added. This action may create dangling pointers, as some cells may contain free occurrences of p. 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 another 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 the last source of possible dangling pointers.

1.3 Safe Type System

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.

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. 4. Operators on type environments

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 function arguments only include either safe or condemned types. The intended semantics of these types is the following:

- ***Safe* types (s):** A data structure (DS) of this type can be read, copied or used to build other DSs. They cannot be destroyed or reused by using the symbol !.
- ***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.
- ***In-danger* types (r):** This is a DS sharing a recursive descendant of a condemned DS, so potentially it can contain dangling pointers.

Data constructors have one region argument $r :: \rho$ reflected as the outermost region variable of the resulting algebraic type $T\ \overline{s}@\overline{\rho}^m$. 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:

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

We assume that the types of the constructors are collected in an environment Σ, easily built from the **data** type declarations. In type environments, Γ, we can find region type assignments $r : \rho$, variable type assignments $x : t$, and polymorphic scheme assignments to functions $f : \sigma$. 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 $\oplus$. Operator $\rhd^L$ is an asymmetric composition used to type **let** expressions. The predicate $utype?(t, t')$ is true when the underlying Hindley-Milner types of t and t' are the same.

In Fig. 5 we show two rules of the type system to illustrate the use of the above environment operators. For a complete description, see [5]. An inference algorithm for this type system has been developed in [6].

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

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

Fig. 5. Two *Safe* typing rules for expressions

2 Static Assertions

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 is a set of program variables and Γ a typing environment. A Safe expression e satisfies a static assertion, denoted $e : [\![L, \Gamma]\!]$, if $L = fv(e)$, there exists a type t such that $\Gamma \vdash e : t$, and some semantic conditions below hold. Our certificate for a given program consists of proving a static assertion $[\![L, \Gamma]\!]$ for each Safe expression e resulting from compiling the program. We will write $\Gamma[x] = m$ to indicate that x has mark $m \in \{s, r, d\}$ in Γ.

The intuitive idea of a variable x being typed with a safe mark s is that all the cells in h reachable at runtime from $E(x)$ do not contain dangling pointers and are disjoint of unsafe cells. The idea behind a condemned variable x is that this cell will be removed from the heap and all live cells reaching any of x's recursive descendants by following a pointer chain are in danger.

We use the following definitions, which have been formally specified in Isabelle:

$\mathit{closure}\ (E, X, h)$	Set of locations reachable in heap h by $\{E(x) \mid x \in X\}$
$\mathit{closure}\ (v, h)$	Set of locations reachable in h by location v
$\mathit{live}\ (E, L, h)$	Live part of h, i.e. $\mathit{closure}\ (E, L, h)$
$\mathit{scope}\ (E, h)$	The part of h reachable from all variables in scope
$\mathit{recReach}\ (E, x, h)$	Set of recursive descendants of $E(x)$ including itself
$\mathit{recReach}\ (v, h)$	Set of recursive descendants of v in h including itself
$\mathit{closed}\ (E, L, h)$	If there are no dangling pointers in $\mathit{live}\ (E, L, h)$
$p \to_h^* V$	There is a pointer path in h from p to a $q \in V$

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

$$\begin{array}{rcl} S_{L,\Gamma,E,h} & \stackrel{\text{def}}{=} & \bigcup_{x \in L, \Gamma[x]=s}\{\mathit{closure}(E, x, h)\} \\ R_{L,\Gamma,E,h} & \stackrel{\text{def}}{=} & \bigcup_{x \in L, \Gamma[x]=d}\{p \in \mathit{live}(E, L, h) \mid p \to_h^* \mathit{recReach}(E, x, h)\} \end{array}$$

We say that two closures are *identical*, denoted $\mathit{closure}(E, x, h) \equiv \mathit{closure}(E, x, h')$, if $\mathit{closure}(E, x, h) = \mathit{closure}(E, x, h')$ and $\forall p \in \mathit{closure}(E, x, h)\ .\ h(p) = h'(p)$.

Definition 2.1 Let us give names to the following properties:

$P1$	$E \vdash h, k, e \Downarrow h', k, v$
$P2$	$dom(\Gamma) \subseteq dom(E)$
$P3$	$L \subseteq dom(\Gamma)$
$P4$	$fv(e) \subseteq L$
$P5$	$\forall x \in dom(E).\ \forall z \in L\ .\ \Gamma[z] = d\ \wedge\ \mathit{recReach}(E, z, h) \cap \mathit{closure}(E, x, h) \neq \emptyset \to x \in dom(\Gamma) \wedge \Gamma[x] \neq s$

$$c : [\![\emptyset, \emptyset]\!] \quad LIT \qquad x : [\![\{x\}, [x : s]]\!] \quad VAR1$$

$$\frac{x \in dom\ \Gamma}{x@r : [\![\{x\}, \Gamma]\!]}\ VAR2 \qquad \frac{\Gamma[x] = d \quad \Gamma \text{ well formed}}{x! : [\![\{x\}, \Gamma]\!]}\ VAR3 \qquad \frac{L = \{\overline{a_i}^2\} \quad \Gamma = \bigoplus_{i=1}^{2}[a_i : s] \text{ defined}}{a_1 \oplus a_2 : [\![L, \Gamma]\!]}\ PRIMOP$$

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

$$\frac{\begin{array}{c}\Sigma(f) = (f\ \overline{x_i}^n@\ \overline{r_j}^m = e_f, \overline{m_i}^n) \qquad L = \{\overline{a_i}^n\} \\ \Gamma_0 = \bigoplus_{i=1}^{n}[a_i : m_i] \text{ defined} \qquad \Gamma \supseteq \Gamma_0 \text{ and well-formed}\end{array}}{f\ \overline{a_i}^n@\ \overline{r_j}^m : [\![L, \Gamma]\!]}\ APP$$

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

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

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

$$\frac{\forall i\ .\ (e_i : [\![L_i, \Gamma_i]\!] \quad \Gamma_i[\overline{x_{ij}}] \neq d) \quad \Gamma \supseteq \bigotimes_i(\Gamma_i/\{\overline{x_{ij}}\}) \quad x \in dom(\Gamma) \quad L = \{x\} \cup (\bigcup_i(L_i - \{\overline{x_{ij}}\}))}{\textbf{case}\ x\ \textbf{of}\ \overline{C_i\overline{x_{ij}} \rightarrow e_i} : [\![L, \Gamma]\!]}\ CASE$$

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

Fig. 6. Proof obligations for explicit deallocation (each one is an Isabelle/HOL theorem)

$P6$ $\quad \forall x \in dom(E)\ .\ closure\ (E, x, h) \not\equiv closure\ (E, x, h') \rightarrow x \in dom(\Gamma) \wedge \Gamma[x] \neq s$
$P7$ $\quad S_{L,\Gamma,E,h} \cap R_{L,\Gamma,E,h} = \emptyset$
$P8$ $\quad closed(E, L, h)$
$P9$ $\quad closed(v, h')$

We say that the expression e satisfies the static assertion $[\![L, \Gamma]\!]$, denoted $e : [\![L, \Gamma]\!]$, if $P1 \wedge P2 \rightarrow P3 \wedge P4 \wedge P5 \wedge P6 \wedge (P7 \wedge P8 \rightarrow P9)$.

The key properties are $P8$ and $P9$. They guarantee that across the whole derivation the live part of the head remains closed, hence there will not be dangling pointers. In Fig. 6 we show the proof obligations related to this property, which must be discharged by the certificate. Each one is a separate theorem interactively proved by Isabelle/HOL, which is kept in its database of proved theorems.

3 Region deallocation

In this section we define and prove correct a number of syntax-driven proof obligations used to create certificates establishing that region deallocation does not create dangling pointers in the heap. As before, the compiler delivers static information about the region types used by program being compiled, and then the proof obligations relate this static information to runtime properties about actual regions.

By $S, S_i, \ldots$ we denote finite sets $\{\rho_1, \ldots, \rho_r\}$ of region type variables. There is a reserved identifier ρ^f_{self} for every defined function f, denoting the type variable assigned to the working region *self* of function f. By $R, R_i, \ldots$ we denote pairs (S, ρ), where ρ is a highlighted region denoting the most external region type of a

given expression, and S denote the rest of its region types. By abuse of notation, sometimes we consider a pair $R = (S, \rho)$ as the set $S \cup \{\rho\}$. By $\theta, \theta_i, \ldots$ we denote *typing mappings* from program variables and region variables to pairs (S, ρ) of region type variables. The intended meaning of $\theta(x) = (S, \rho),\ \rho' \in S \cup \{\rho\}$ is that ρ' is one of the region types occurring in the type the compiler assigns to x. If $\rho' = \rho$, then it is the most external region type. For region variables, the pair has the form $\theta(r) = (\{\rho\}, \rho)$, and we mean that ρ is the type the compiler assigns to r. We will call *range* of a typing mapping θ to the set:

$$range\ (\theta) \stackrel{\text{def}}{=} \bigcup_{x \in dom(\theta),\ (S,\ \rho) = \theta(x)} (S \cup \{\rho\})$$

By $\eta,\ \eta_i, \ldots$ we denote *instantiation mappings* from region type variables to actual region 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 $\rho,\ 0 \leq \eta(\rho) \leq k$ 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'. We will apply a mapping η to a pair $R = (S, \rho)$ of region types, obtaining the set $\eta(R) \stackrel{\text{def}}{=} \{\eta(\rho') \mid \rho' \in S \cup \{\rho\}\}$ of actual regions. Admissible instantiation mappings should map ρ_{self} to the topmost region and other region types to lower regions.

Definition 3.1 Assuming that k denotes the topmost region of a given heap, we say that the mapping η is admissible, denoted *admissible* (η, k), if:

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

We introduce a notion of consistency between the static information $\theta,\ R$ and the dynamic one $E,\ \eta,\ h,\ h'$. Essentially, consistency tells us that the static region types, its instantiation to actual regions, and the actual regions where the data structures are stored in the heap, do not contradict each other.

Definition 3.2 We say that the mappings $\theta,\ \eta$, the runtime environment E, and the heap h are *consistent*, denoted *consistent* (θ, η, E, h), if:

(i) $\forall x \in dom(E)\ .\ \theta(x) = (S, \rho) \rightarrow regions\ (closure\ (E, x, h) - recReach\ (E, x, h)) \subseteq \eta(S)$
$\wedge\ regions\ (recReach\ (E, x, h)) \subseteq \eta(\rho)$

(ii) $\forall r \in dom(E)\ .\ \{E(r)\} = (\eta \cdot \theta)(r)$

(iii) $self \in dom(E) \wedge \theta(self) = \rho^f_{self}$

Likewise, we define *consistent* (R, η, v, h) as:

$$regions\ (closure\ (v, h) - recReach\ (v, h)) \subseteq \eta(S) \wedge regions\ (recReach\ (v, h)) \subseteq \eta(\rho)$$

where $regions\,(P, h)$ is defined as the set: $regions\,(P, h) \stackrel{\text{def}}{=} \{j \mid p \in P,\ h(p) = (j, w)\}$

When dealing with function (constructor) application, the region types used in the polymorphic signature of a function g (a constructor C) must be related to the actual region types used in the invocation. Let us denote by μ the *type instantiation mapping* used by the compiler. This mapping should correctly map the region types of the formal arguments, to the types of the corresponding actual arguments.

$$c \vdash \theta \rightsquigarrow (\emptyset, \bot)\ \mathit{LIT} \qquad x \vdash \theta \rightsquigarrow \theta(x)\ \mathit{VAR1} \qquad x! \vdash \theta \rightsquigarrow \theta(x)\ \mathit{VAR3}$$

$$\frac{\theta(x) = (S, \rho) \quad \theta(r) = (\{\rho'\}, \rho')}{x@r \vdash \theta \rightsquigarrow (S, \rho')}\ \mathit{VAR2} \qquad \frac{e_1 \vdash \theta \rightsquigarrow R_1 \quad e_2 \vdash \theta \cup [x_1 \mapsto R_1] \rightsquigarrow R_2}{\textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2 \vdash \theta \rightsquigarrow R_2}\ \mathit{LET}$$

$$\frac{C\ \overline{x_i}^n@r \vdash (\theta_C, R_T) \quad \mathit{argP}\ (\theta_C, \mu, \theta) \quad e_2 \vdash \theta \cup [x_1 \mapsto \mu(R_T)] \rightsquigarrow R_2}{\textbf{let}\ x_1 = C\ \overline{a_i}^n@r'\ \textbf{in}\ e_2 \vdash \theta \rightsquigarrow R_2}\ \mathit{LET_C}$$

$$\frac{\forall i\ (C_i\ \overline{y_{ij}}@r \vdash (\theta_{C_i}, R_T) \quad \theta_i = \overline{[x_{ij} \mapsto \mu(\theta_{C_i}(y_{ij}))]} \quad e_i \vdash \theta \cup \theta_i \rightsquigarrow R) \quad \mu(R_T) = \theta(x)}{\textbf{case}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}} \rightarrow e_i}^n \vdash \theta \rightsquigarrow R}\ \mathit{CASE}$$

$$\frac{\forall i\ (C_i\ \overline{y_{ij}}@r \vdash (\theta_{C_i}, R_T) \quad \theta_i = \overline{[x_{ij} \mapsto \mu(\theta_{C_i}(y_{ij}))]} \quad e_i \vdash \theta \cup \theta_i \rightsquigarrow R) \quad \mu(R_T) = \theta(x)}{\textbf{case!}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}} \rightarrow e_i}^n \vdash \theta \rightsquigarrow R}\ \mathit{CASE!}$$

$$\frac{\Sigma(g) = (g\ \overline{x_i}^n@\ \overline{r_j}^m = e_g) \quad e_g \vdash \theta_g \rightsquigarrow R_g \quad \mathit{argP}\ (\theta_g, \mu, \theta_f) \quad \rho^g_{self} \notin R_g}{g\ \overline{a_i}^n@\ \overline{r'_j}^m \vdash \theta_f \rightsquigarrow \mu(R_g)}\ \mathit{APP}$$

Fig. 7. Proof obligations (each one is an Isabelle/HOL theorem)

Definition 3.3 Given the typing mappings θ_g and θ_f, and a type instantiation mapping μ, we say that the triple $(\theta_g, \mu, \theta_f)$ is *argument preserving*, denoted $\mathit{argP}\ (\theta_g, \mu, \theta_f)$, if:

$$(\forall i \in \{1, \ldots, n\}\ .\ (\mu \cdot \theta_g)(x_i) = \theta_f(a_i)) \wedge (\forall j \in \{1, \ldots, m\}\ .\ (\mu \cdot \theta_g)(r_j) = \theta_f(r'_j))$$

In the case of a constructor C of an algebraic type T, let us assume that the typing system provides: $C :: \forall \overline{\rho_j}\ .\ \overline{t_i}^n \rightarrow \rho \rightarrow T$. Then, for arbitrary fresh names x_i, $1 \leq i \leq n$, and r we can build $\theta_C = \{\overline{x_i \mapsto (S_i, \rho_i)}^n\} \cup \{r \mapsto (\{\rho\}, \rho)\}$ and $R_T = (S, \rho)$, being ρ_i the most external region of type t_i, S_i the rest of its regions, ρ the most external region of type T, and S the rest of its regions. Then, the pair (θ_C, R_T) is an abstraction of the type signature of expression $C\ \overline{x_i}^n@r$.

Let $C\ \overline{a_i}^n@r'$ be a constructor application, where its free variables belong to the domain of a typing mapping θ_f. Then, we can apply the Definition 3.3 above and say that $(\theta_C, \mu, \theta_f)$ is an *argument preserving* type instantiation mapping, if

$$(\forall i \in \{1, \ldots, n\}\ .\ (\mu \cdot \theta_C)(x_i) = \theta_f(a_i)) \wedge (\mu \cdot \theta_C)(r) = \theta_f(r')$$

A judgement of the form $e \vdash \theta \rightsquigarrow R$ states that, if expression e is evaluated within an environment E, heap h, and region mapping η consistent with θ, then η, the final heap h', and the final value v are consistent with R. Formally:

Definition 3.4 An expression e satisfies the pair (θ, R), denoted $e \vdash \theta \rightsquigarrow R$ if

$$\begin{aligned} \forall E\ h\ k\ h'\ v\ \eta\ .\ & E \vdash h, k, e \Downarrow h', k, v \ \wedge\ \mathit{dom}(E) \subseteq \mathit{dom}(\theta) \ \wedge\ \mathit{range}\ (\theta) \subseteq \mathit{dom}(\eta) \\ & \wedge\ \mathit{consistent}\ (\theta, \eta, E, h) \ \wedge\ \mathit{admissible}\ (\eta, k) \quad \rightarrow\ \mathit{consistent}\ (R, \eta, v, h') \end{aligned}$$

The key property here is *admissible* (η, k). It guarantees that only ρ^f_{self} is mapped to the topmost region k of f. Hence, when f terminates only the bindings there are deleted. This, together with the static check that ρ^f_{self} does not occur in f's result type (see rule *APP* of Fig. 7), proves that region deallocation does not create dangling pointers. In Fig. 7, we show the proof obligations for this property. As before, each one is a theorem interactively proved by Isabelle/HOL.

4 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 abstract syntax tree and producing the following information:

- 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, typing environments, sets of region types, etc.
- A set of Isabelle/HOL proof scripts proving a lemma for each expression, consisting of first proving the premises of the proof obligation (theorem) associated to the syntactic form of the expression, and then applying the theorem.

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 theorems— has been done in advance and is reused by each certified program.

5 Related Work

The first approaches to PCC generated type-based certificates at the assembly code level [8]. In [1], a type system similar to ours is presented, but they lack certificate generation. Some recent work [2] connects the information provided by static analyses to certificate generation. Our work is more closely related to [3], where a resource property obtained by a special type system is transformed into a certificate. Our static assertions have been inspired by their *derived assertions*, used also to connect static with dynamic properties.

References

[1] D. Aspinall, M. Hofmann, and M. Konečný. A Type System with Usage Aspects. *Journal of Functional Programming*, 18(2):141–178, 2008.

[2] G. Barthe, B. Grégoire, C. Kunz, and T. Rezk. Certificate Translation for Optimizing Compilers. In *Static Analysis Symp., SAS'06, LNCS 4134*, pages 301–317. Springer, 2006.

[3] L. Beringer, M. Hofmann, A. Momigliano, and O. Shkaravska. Automatic Certification of Heap Consumption. In *LPAR'04, LNAI 3452*, pages 347–362. Springer, 2005.

[4] M. Montenegro, R. Peña, and C. Segura. A Simple Region Inference Algorithm for a First-Order Functional Language. In *Trends in Functional Programming, TFP'08, Nijmegen (The Netherlands)*, pages 194–208, May 2008.

[5] 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.

[6] M. Montenegro, R. Peña, and C. Segura. An Inference Algorithm for Guaranteeing Safe Destruction. In *Selected papers of Logic-Based Program Sinthesis and Transformation, LOPSTR'08, LNCS 5438, Springer.*, pages 135–151, 2009.

[7] G. C. Necula. Proof-Carrying Code. In *ACM SIGPLAN-SIGACT Principles of Programming Languages, POPL'97*, pages 106–119. ACM Press, 1997.

[8] 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.

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