Certified Absence of Dangling Pointers in a Language with Explicit Deallocation [⋆] (Extended Version)

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 for analysing their safety in an automatic way.

The above analyses have been manually proved correct in [11], but we embarked ourselves 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 one such theorem for each syntactic construction of the language. Then, these theorems or *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 on. 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. Sec. 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 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:

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

Fig. 1. *Core-Safe* syntax

```
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 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 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. By $h[p \mapsto w]$ we denote a heap h where the binding $[p \mapsto w]$ is highlighted. On the contrary, by $h \uplus [p \mapsto w]$ we denote the disjoint union of heap h with the binding $[p \mapsto w]$. By $h\mid_k$ we denote the heap obtained by deleting from h those bindings living in regions greater than k.

The semantics of a program $\mathit{dec}_1; \ldots; \mathit{dec}_n; e$ is the semantics of the main expression e in an environment Σ containing all the function declarations.

$$E \vdash (h,k), c \Downarrow (h,k), c \quad [\mathit{Lit}] \qquad E[x \mapsto v] \vdash (h,k), x \Downarrow (h,k), v \quad [\mathit{Var}_1]$$

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

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

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

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

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

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

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

$$\frac{(\text{if } \exists r\,.\, c = c_r \text{ then } e_a = e_r \text{ else } e_a = e_d) \quad E \vdash (h,k), e_a \Downarrow (h',k), v}{E[x \mapsto c] \vdash (h,k), \mathbf{case}\ x\ \mathbf{of}\ \{\overline{c_i \ \to e_i}^m; \_ \to e_d\} \Downarrow (h',k), v} \ [\mathit{Case}_{Lit}]$$

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

Fig. 2. Operational semantics of *Safe* expressions

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 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 informaton is available in our runtime system so that this function can be implemented. The pointers in non recursive positions of all the copied cells 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.

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

Fig. 3. Type expressions

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.

Rule $Case_{Lit}$ for literal values could be considered as a particular case of rule *Case* in which the constructors introduce no patterns. But in fact it is a different rule because a default clause e_d is also present . This alternative is taken when it is no possible a match with any of the literals of the remaining alternatives.

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

2.2 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 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

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

Fig. 4. Operators on type environments

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:

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

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

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

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. 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. 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.

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 ($\rho_{\mathit{self}} \notin \mathit{regions}(s)$). 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 $\mathit{sharerec}(x, e)$ 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 [17].

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 1) telling that if a

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

Fig. 5. Rule for function definitions

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 $\mathit{type}(y)$ represents the Hindley-Milner type inferred for variable y[1].

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

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

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.

Rules [LET1] and [LET2] control the intermediate results by means of operator $\rhd^L$. Rule [LET1] is applied when the intermediate result is safely used in the main expression. Rule [LET2] allows the intermediate result x_1 to be used destructively in the main expression e_2 if desired. In both **let** rules operator $\rhd$, 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. $y \notin \mathit{fv}(e_2)$), 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.

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

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

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

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

$$\frac{\Gamma_1 \vdash e_1 : s_1 \quad \Gamma_2 + [x_1 : s_1] \vdash e_2 : s}{\Gamma_1 \rhd^{fv(e_2)} \Gamma_2 \vdash \textbf{let } x_1 = e_1 \textbf{ in } e_2 : s}\text{[LET1]} \qquad \frac{\Gamma_1 \vdash e_1 : s_1 \quad \Gamma_2 + [x_1 : d_1] \vdash e_2 : s \quad utype?(d_1, s_1)}{\Gamma_1 \rhd^{fv(e_2)} \Gamma_2 \vdash \textbf{let } x_1 = e_1 \textbf{ in } e_2 : s}\text{[LET2]}$$

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

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

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

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

Fig. 6. Type rules for expressions

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.

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

Fig. 7. Definitions of inheritance compatibility

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.

Lemma 1. *If $\Gamma \vdash e : s$ and $\Gamma(x) = d$ then:*

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

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 [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 $\mathit{fv}(e)$ we denote the set of free variables of expression e, excluding function names and region variables, and by $\mathit{dom}(h)$ the set $\{p \mid [p \mapsto w] \in h\}$. A *static assertion* has the form $[\![L, \Gamma]\!]$, where $L \subseteq \mathit{dom}(\Gamma)$ is a set of program variables and Γ a *variable 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 $[\![L, \Gamma]\!]$ if $\mathit{fv}(e) \subseteq L$ 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:

$\mathit{closure}\ (E, X, h)$	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
$\mathit{closed}\ (E, L, h)$	There are no dangling pointers in h, i.e. $\mathit{closure}\ (E, L, h) \subseteq \mathit{dom}(h)$
$p \rightarrow_h^* V$	There is a pointer path in h from p to a $q \in V$

By abuse of notation, we will write $\mathit{closure}(E, x, h)$ and also $\mathit{closed}(v, h)$. The formal definitions of these predicates and functions are the following:

Definition 1. *Given a heap h, we define the child ($\rightarrow_h$) and recursive child ($\twoheadrightarrow_h$) relations on heap pointers as follows:*

$$\begin{array}{l}
p \rightarrow_h q \overset{def}{=} h(p) = C\ \overline{p_i}^n \ \wedge\ q = p_i \text{ for some } i \in \{1..n\} \\
p \twoheadrightarrow_h q \overset{def}{=} h(p) = C\ \overline{p_i}^n \ \wedge\ q = p_i \text{ for some } i \in \mathit{RecPos}(C)
\end{array}$$

The reflexive and transitive closure of these relations are respectively denoted by $\rightarrow_h^*$ and $\twoheadrightarrow_h^*$.

Definition 2.

$$\begin{array}{lcl}
\mathit{closure}\ (E, X, h) & \overset{def}{=} & \{q \mid E(x) \rightarrow_h^* q \ \wedge\ x \in X\} \\
\mathit{closure}\ (p, h) & \overset{def}{=} & \{q \mid p \rightarrow_h^* q\} \\
\mathit{live}\ (E, L, h) & \overset{def}{=} & \mathit{closure}\ (E, L, h) \\
\mathit{scope}\ (E, h) & \overset{def}{=} & \mathit{closure}\ (E, \mathit{dom}(E), h) \\
\mathit{recReach}\ (E, x, h) & \overset{def}{=} & \{q \mid E(x) \twoheadrightarrow_h^* q\} \\
\mathit{recReach}\ (p, h) & \overset{def}{=} & \{q \mid p \twoheadrightarrow_h^* q\} \\
\mathit{closed}\ (E, L, h) & \overset{def}{=} & \mathit{live}\ (E, L, h) \subseteq \mathit{dom}(h) \\
p \rightarrow_h^* V & \overset{def}{=} & \exists q \in V.\, p \rightarrow_h^* q
\end{array}$$

We make note that these predicates are well defined even if for some x, $x \notin \mathit{dom}(E)$ or $E(x) \notin \mathit{dom}(h)$. In the first case $\mathit{closure}\ (E, \{x\}, h) = \emptyset$, and in the second case $\mathit{closure}\ (E, \{x\}, h) = \{E(x)\}$.

Now, we define the following two sets, respectively of safe and unsafe heap locations, as functions of L, Γ, E, and h:

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

In a *Core-Safe* program, an expression e belongs either to a function's body or to the main expression. In both cases it may contain applications to previously defined functions. The relevant information about previously defined functions will be kept in a *function mark environment* Σ_M, which is a partial function Σ_M : *string*$\rightarrow$ [*mark*] from function names to marks. If $\Sigma_M(f) = \overline{m_i}^n$, then f can be safely called with marks m_i in actual arguments a_i. Due to restrictions of our type system, these 'top-level' marks can only belong to $\{s, d\}$. This environment is incrementally built as functions are certified. The following definitions introduce two concepts: what is meant by an expression to *satisfy* an static assertion in the context of a mark environment, and what is meant by an environment to be *valid*.

Definition 3. *Let e be a subexpression of f's body different from a recursive call (i.e. $e \neq f\ \overline{a_i}$ @ $\overline{r_j}$), Σ_M a function mark environment, L a set of program variables, and Γ a variable mark environment. Given the following properties:*

$P1 \equiv E \vdash h, k, e \Downarrow h', k, v$ *and the functions which may be called by e, except perhaps f, are defined in Σ_M*
$P2 \equiv dom(\Gamma) \subseteq dom(E)$
$P3 \equiv L \subseteq dom(\Gamma)$
$P4 \equiv fv(e) \subseteq L$
$P5 \equiv \forall x \in dom(E).\ \forall z \in L\ .$
$\quad \Gamma[z] = d\ \wedge\ recReach(E, z, h) \cap closure(E, x, h) \neq \emptyset \rightarrow x \in dom(\Gamma) \wedge \Gamma[x] \neq s$
$P6 \equiv \forall x \in dom(E)\ .\ closure\ (E, x, h) \not\equiv closure\ (E, x, h') \rightarrow x \in dom(\Gamma) \wedge \Gamma[x] \neq s$
$P7 \equiv S_{L,\Gamma,E,h} \cap R_{L,\Gamma,E,h} = \emptyset$
$P8 \equiv closed(E, L, h)$
$P9 \equiv closed(v, h')$

having as free variables e, L, Γ and Σ_M, we say that the expression e satisfies *the static assertion $[\![L, \Gamma]\!]$ in the context of Σ_M, denoted $e, \Sigma_M : [\![L, \Gamma]\!]$, if*

$$P3 \wedge P4 \wedge (\forall E\ h\ k\ h'\ v\ .\ P1 \wedge P2 \rightarrow P5 \wedge P6 \wedge (P7 \wedge P8 \rightarrow P9))$$

If $e = f\ \overline{a_i}$ @ $\overline{r_j}$ and $f \in dom(\Sigma_T)$, we say that e satisfies the assertion $[\![L, \Gamma]\!]$ in the context of Σ_M, denoted also $e, \Sigma_M : [\![L, \Gamma]\!]$, if $\Sigma_M(f) = \overline{m_i}^n$, $L = \{\overline{a_i}^n\}$, $\Gamma_0 = \bigoplus_{i=1}^n [a_i : m_i]$ is defined, and $\Gamma \supseteq \Gamma_0$ is well-formed.

Definition 4. *A mark environment Σ_M is* valid, *denoted $\models \Sigma_M$, if it can be deduced from the following inductive rules.*

1. *The empty environment is valid, i.e. $\models \emptyset$.*
2. *If $\models \Sigma_M$ holds, function f is defined as $f\ \overline{x_i}^n$@ $\overline{r_j}^m = e_f$, $L_f = \overline{x_i}^n$, $\Gamma_f = [\overline{x_i \mapsto m_i}^n]$, and $e_f, \Sigma_M : [\![L_f, \Gamma_f]\!]$ holds, then $\models \Sigma_M \uplus [f \mapsto \overline{m_i}^n]$ also holds.*

Property $P1$ defines any runtime evaluation of e. Properties $P2$ to $P4$ just guarantee that each free variable has a type and a meaning. Properties $P5$ to $P7$ formalise the meaning of safe and condemned types: if some variable can

$$\frac{}{c, \Sigma_M : \llbracket \emptyset, \emptyset \rrbracket}\ \mathit{LIT} \qquad \frac{}{x, \Sigma_M : \llbracket \{x\}, \Gamma + [x : s] \rrbracket}\ \mathit{VAR1} \qquad \frac{x \in \mathit{dom}\ \Gamma \quad \Gamma \text{ well-formed}}{x@r, \Sigma_M : \llbracket \{x\}, \Gamma \rrbracket}\ \mathit{VAR2}$$

$$\frac{\Gamma[x] = d \quad \Gamma \text{ well-formed}}{x!, \Sigma_M : \llbracket \{x\}, \Gamma \rrbracket}\ \mathit{VAR3} \qquad \frac{L = \{a_1, a_2\} \quad \Gamma = [a_1 : s, a_2 : s]}{a_1 \oplus a_2, \Sigma_M : \llbracket L, \Gamma \rrbracket}\ \mathit{PRIMOP}$$

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

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

$$\frac{L_1 = \{\overline{a_i}^n\} \quad \Gamma_1 = [\overline{a_i \mapsto s}^n] \quad x_1 \notin L_1 \quad e_2, \Sigma_M : \llbracket L_2, \Gamma_2' + [x_1 : s] \rrbracket \quad \mathit{def}(\Gamma_1 \rhd^{L_2} \Gamma_2')}{\mathbf{let}\ x_1 = C\ \overline{a_i}^n@r\ \mathbf{in}\ e_2, \Sigma_M : \llbracket L_1 \cup (L_2 - \{x_1\}), \Gamma_1 \rhd^{L_2} \Gamma_2' \rrbracket}\ \mathit{LET1}_C$$

$$\frac{L_1 = \{\overline{a_i}^n\} \quad \Gamma_1 = [\overline{a_i \mapsto s}^n] \quad x_1 \notin L_1 \quad e_2, \Sigma_M : \llbracket L_2, \Gamma_2 + [x_1 : d] \rrbracket \quad \mathit{def}(\Gamma_1 \rhd^{L_2} \Gamma_2')}{\mathbf{let}\ x_1 = C\ \overline{a_i}^n@r\ \mathbf{in}\ e_2, \Sigma_M : \llbracket L_1 \cup (L_2 - \{x_1\}), \Gamma_1 \rhd^{L_2} \Gamma_2' \rrbracket}\ \mathit{LET2}_C$$

$$\frac{\begin{array}{c}\forall i\,.\,(e_i, \Sigma_M : \llbracket L_i, \Gamma_i \rrbracket \quad \forall j.\Gamma_i[x_{ij}] \neq d) \quad \Gamma \text{ well-formed} \\ \Gamma \supseteq \bigotimes_i (\Gamma_i \backslash \{\overline{x_{ij}}\}) \quad x \in \mathit{dom}(\Gamma) \quad L = \{x\} \cup (\bigcup_i (L_i - \{\overline{x_{ij}}\}))\end{array}}{\mathbf{case}\ x\ \mathbf{of}\ \overline{C_i \overline{x_{ij}} \to e_i}, \Sigma_M : \llbracket L, \Gamma \rrbracket}\ \mathit{CASE}$$

$$\frac{\forall i\,.\,(e_i, \Sigma_M : \llbracket L_i, \Gamma_i \rrbracket) \quad \Gamma = \bigotimes_i \Gamma_i \quad \Gamma[x] = s \quad L = \{x\} \cup (\bigcup_i L_i)}{\mathbf{case}\ x\ \mathbf{of}\ \{\overline{c_i \to e_i}^n; \_ \to e_{n+1}\}, \Sigma_M : \llbracket L, \Gamma \rrbracket}\ \mathit{CASEL}$$

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

$$\frac{\models \Sigma_M \backslash \{f\} \quad \Sigma_M(g) = \overline{m_i}^n \quad L = \{\overline{a_i}^n\} \quad \Gamma_0 = \bigoplus_{i=1}^n [a_i : m_i] \text{ defined} \quad \Gamma \supseteq \Gamma_0 \text{ and well-formed}}{g\ \overline{a_i}^n@\ \overline{r_j'}^m, \Sigma_M : \llbracket L, \Gamma \rrbracket}\ \mathit{APP}_{nonrec}$$

$$\frac{\begin{array}{lll}\models \Sigma_M & f\ \overline{x_i}^n@\ \overline{r_j}^m = e_f & L_f = \{\overline{x_i}^n\} \\ \Gamma_f \supseteq [\overline{x_i \mapsto m_i}^n] & \Gamma_f \text{ well-formed} & e_f, \Sigma_M \uplus [f \mapsto \overline{m_i}^n] : \llbracket L_f, \Gamma_f \rrbracket\end{array}}{\models \Sigma_M \uplus [f \mapsto \overline{m_i}^n]}\ \mathit{FUN}$$

Fig. 8. Proof rules for explicit deallocation (each one is an Isabelle/HOL theorem)

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 are proved for all subexpressions of e, they guarantee that across the whole derivation of e the live part of the heap remains closed, hence there will not be dangling pointers. In Fig. 8 we show the proof rules related to this property (we assume that expressions are subexpressions of the body of a given function f). Each one is a separate theorem interactively proved by Isabelle/HOL, which is kept in its database of proved theorems. The predicate 'Γ *well-formed*' is equivalent to the property:

$$P3 \wedge P4 \wedge (\forall E\ h\ k\ h'\ v\,.\,P1 \wedge P2 \to P5)$$

It expresses that all the variables in scope that may share at runtime a recursive descendant of a condemned one, should occur in Γ with unsafe marks. This essentially asserts that the sharing analysis done by the compiler is sound, i.e. that the set $\mathit{sharerec}(x, e)$ occurring in some rules of the type system (see Fig. 6) is a correct upper approximation of this set of variables. For the moment no attempt has been done to certify this property. It would need a separate set of proof rules and a corresponding certificate, probably as complex as the one presented here. This effort is foreseen as future work.

Notice that the APP_{nonrec} rule can only be applied when the function called is defined in Σ_M and $\Sigma_M \backslash \{f\}$ is a valid environment (i.e. we do not require the marks given for the function f being certified to be valid). This enforces functions to be certified from low-level ones not calling to any other function, to high-level ones which may call to those functions already certified. For recursive calls, the Def. 3 admits any set of marks given by $\Sigma_M(f)$ as valid, and it only checks that the actual arguments have these marks assigned in Γ and that the operator $\oplus$ is well-defined. The *FUN* proof-rule is the one establishing the validity of this assumption.

3.1 Proof obligations discharged by the certificate

For each expression e, the compiler generates a pair (L, Γ). According to e's syntax, the certificate will perform the following actions:

$\boxed{c}$ The certificate checks $(L, \Gamma) = (\emptyset, \emptyset)$ and applies the proof rule *LIT*.

$\boxed{x}$ The certificate checks $L = \{x\}, \Gamma[x] = s$, and applies the proof rule *VAR1*.

$\boxed{x@r}$ The certificate checks $L = \{x\}, x \in dom(\Gamma)$, and applies the proof rule *VAR2*. The well-formedness of Γ make reference to the fact that all unsafe variables in scope must satisfy property $P5$. There is nothing to check here because the property is guaranteed by a sharing analysis manually proved correct and whose correctness has not been incorporated in the proof rules.

$\boxed{x!}$ The certificate checks $L = \{x\}, \Gamma[x] = d$, and applies the proof rule *VAR3*. As before, the well-formedness of Γ implies no extra checking.

$\boxed{a_1 \oplus a_2}$ The certificate checks $L = vars(\{a_1, a_2\})$,$\forall x \in L.\Gamma[x] = s$, and applies the proof rule *PRIMOP*.

$\boxed{\textbf{let } x_1 = e_1 \textbf{ in } e_2}$ Let us assume $e_1 \neq C\ \overline{a_i}^n$ and $\Gamma_2[x_1] = s$. The certificate has already proved $e_1, \Sigma_M : [\![L_1, \Gamma_1]\!]$ and $e_2, \Sigma_M : [\![L_2, \Gamma_2]\!]$, and receives (L, Γ) for the **let** expression. It checks $\Gamma_2 = \Gamma_2' + [x_1 : s]$, $x_1 \notin L_1$, $def(\Gamma_1 \triangleright^{L_2} \Gamma_2')$, $L = L_1 \cup (L_2 - \{x_1\})$, $\Gamma = \Gamma_1 \triangleright^{L_2} \Gamma_2'$, and then applies the proof rule *LET1*. If $\Gamma_2 = \Gamma_2' + [x_1 : d]$, then it applies the proof rule *LET2*. Note that there is no proof rule for $\Gamma_2[x_1] = r$. In fact this typing is not correct and it should not occur in a well-typed program.
If $e_1 = C\ \overline{a_i}^n$, then the certificate checks that all the constructor arguments which are variables have a safe mark in Γ_1. The rest of checkings are similar to the cases above. The corresponding proof rules are $LET1_C$ and $LET2_C$.

$\boxed{\textbf{case } x \textbf{ of } \overline{C_i \overline{x_{ij}} \rightarrow e_i}}$ The certificate has already proved $e_i, \Sigma_M : [\![L_i, \Gamma_i]\!]$ for all the subsidiary expressions, and receives (L, Γ) for the whole **case** expression. It then checks the premises of the *CASE* proof rule and applies it. The premises $\forall i\ j\ .\ \Gamma_i[x_{ij}] \neq d$ and $\Gamma \supseteq \bigotimes_i(\Gamma_i \backslash \{\overline{x_{ij}}\})$ hold for every well-typed program, and express that **case** patterns are never condemned, and that the rest of free variables occur with the same mark both in the subsidiary environments Γ_i and in the whole one Γ. The rule *CASEL* is invoked by the certificate when the type of the discriminant variable x is a basic one (either an integer or a boolean). This fact is detected by the compiler, which generates the appropriate certificate in each case.

$\boxed{\textbf{case!}\ x\ \textbf{of}\ \overline{C_i \overline{x_{ij}} \to e_i}}$ The differences with the conventional **case** are the following: (1) the discriminant variable x should occur as condemned in environment Γ; (2) the condemned patterns must be recursive children of x (function $recPos(C)$ gives the recursive positions of constructor C); and (3) the premise $\Gamma[z] \neq s \to (\forall i \,.\, z \notin L_i)$ requires that unsafe variables in Γ should not occur free in the subsidiary expressions (this property holds for well-typed programs).

$\boxed{g\ \overline{a_i}^n @\ \overline{r'_j}^m}$ The certificate generated by the compiler contains the incremental definition of the function mark environment Σ_M having the marks m_i assigned to the formal arguments of previously defined functions. Then, for each application of g, the certificate gets this information from Σ_M, and checks that the given mark environment Γ contains the actual arguments a_i with these marks assigned. It also checks that operator $\bigoplus$, requiring a duplicated actual argument to be safe (see Fig. 4), is well-defined. Then, it applies the APP_{nonrec} proof rule.

$\boxed{f\ \overline{a_i}^n @\ \overline{r'_j}^m}$ If f is recursive, the FUN rule allows introducing in Σ_M the assumption $[f \mapsto \overline{m_i}^n]$ so that the internal applications to f could be checked. Then, for each internal application of f, the certificate gets this information from Σ_M, and checks that $L = \{\overline{a_i}\}$, and that the given mark environment Γ contains the actual arguments a_i with these marks assigned. It also checks that operator $\bigoplus$ is well-defined. Then it applies Def. 3, and establishes $f\ \overline{a_i}\ @\ \overline{r_j}, \Sigma_M : [\![L, \Gamma]\!]$.

$\boxed{f\ \overline{x_i}^n @\ \overline{r_j}^m = e_f}$ For each recursive function definition as this one, the certificate builds the sets $L_f = \{\overline{x_i}^n\}$, $\Gamma_f = [\overline{x_i \mapsto m_i}^n]$ and proves $e_f, \Sigma_M \uplus [f \mapsto \overline{m_i}^n] : [\![L_f, \Gamma_f]\!]$. The validity of Σ_M should have been established previously to finding this definition. Then, it applies the FUN rule and establishes that $\Sigma_M \uplus [f \mapsto \overline{m_i}^n]$ is a valid environment which can be used to certify subsequent functions.

3.2 Proof schemes of the proof rules

$\boxed{\boldsymbol{LIT}}$ The rule to be proved is:

$$c, \Sigma_M : [\![\emptyset, \emptyset]\!]$$

Proof. We must prove $P_3 \wedge P_4 \wedge (\forall E\ h\ k\ h'\ v . P_1 \wedge P_2 \to P_5 \wedge P_6 \wedge (P_7 \wedge P_8 \to P_9))$. Properties P_3, P_4 hold trivially. Assuming property P_1 we get $h = h'$ and $v = c$. With this, property P_6 holds trivially. Properties P_5, and P_9 also hold trivially, so the assertion is true, even disregarding P_7 and P_8. □

$\boxed{\boldsymbol{VAR1}}$ The rule to be proved is:

$$x, \Sigma_M : [\![\{x\}, \Gamma + [x : s]]\!]$$

Proof. We must prove $P_3 \wedge P_4 \wedge (\forall E\ h\ k\ h'\ v . P_1 \wedge P_2 \rightarrow P_5 \wedge P_6 \wedge (P_7 \wedge P_8 \rightarrow P_9))$. Properties P_3 and P_4 hold trivially. Assuming property P_1 we get $h = h'$ and $v = E(x)$. With this, property P_6 trivially holds. Property P_5 also holds trivially. Assuming $P_8 = \mathit{closed}(E, L, h)$ we immediately get $\mathit{closed}(E(x), h)$ which is P_9, so the assertion is true, even disregarding P_7. □

$\boxed{\textbf{\textit{VAR2}}}$ The rule to be proved is:

$$\frac{x \in \mathit{dom}\ \Gamma \quad \Gamma \text{ well-formed}}{x@r, \Sigma_M : [\![\{x\}, \Gamma]\!]}$$

Proof. We must prove $P_3 \wedge P_4 \wedge (\forall E\ h\ k\ h'\ v . P_1 \wedge P_2 \rightarrow P_5 \wedge P_6 \wedge (P_7 \wedge P_8 \rightarrow P_9))$. Property P_4 holds trivially, and P_3 holds by assumption. Assuming P_1 we get $(h', v) = \mathit{copy}(h, E(x), j)$; P_5 is exactly the well-formedness of Γ holding by assumption; P_6 is true because no closure changes during the evaluation of *copy*. This runtime function makes a copy of $\mathit{recReach}(E, x, h)$ and the resulting structure shares the rest of $\mathit{closure}(E, x, h)$. This semantics guarantees that closedness is preserved. Then, P_8 implies P_9 and we are done. □

$\boxed{\textbf{\textit{VAR3}}}$ The rule to be proved is:

$$\frac{\Gamma[x] = d \quad \Gamma \text{ well formed}}{x!, \Sigma_M : [\![\{x\}, \Gamma]\!]}$$

Proof. We must prove $P_3 \wedge P_4 \wedge (\forall E\ h\ k\ h'\ v\ .\ P_1 \wedge P_2 \rightarrow P_5 \wedge P_6 \wedge (P_7 \wedge P_8 \rightarrow P_9))$. Properties P_3 and P_4 hold trivially. Assuming P_1 we get $h' = h \backslash [E(x) \mapsto w] \uplus [v \mapsto w]$, and $\mathit{fresh}(v)$; P_5 holds by the well-formedness assumption; P_6 holds because the only closures changing during the evaluation of $x!$ are those including $E(x)$. These are exactly the same ones considered in P_5, so P_5 implies P_6. Finally P_8, and a lemma proving that starting from an empty heap the semantics never creates cycles in it, imply P_9. □

$\boxed{\textbf{\textit{PRIMOP}}}$ The rule to be proved is:

$$\frac{L = \{a_1, a_2\} \quad \Gamma = [a_1 : s, a_2 : s]}{a_1 \oplus a_2, \Sigma_M : [\![L, \Gamma]\!]}\ \mathit{PRIMOP}$$

Proof. We must prove $P_3 \wedge P_4 \wedge (\forall E\ h\ k\ h'\ v . P_1 \wedge P_2 \rightarrow P_5 \wedge P_6 \wedge (P_7 \wedge P_8 \rightarrow P_9))$. Properties P_3, P_4 hold trivially. Assuming property P_1 we get $h = h'$ and v is a constant. So, property P_6 trivially holds; P_5 and P_9 also hold trivially, so the assertion is true, even disregarding P_2, P_7, and P_8. □

$\boxed{\textbf{\textit{LET1}}}$ The rule to be proved is:

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

Proof. We must prove $P_3 \wedge P_4 \wedge (\forall E\ h\ k\ h'\ v \,.\, P_1 \wedge P_2 \rightarrow P_5 \wedge P_6 \wedge (P_7 \wedge P_8 \rightarrow P_9))$ for the **let**. The steps are the following:

1. P_3 for e_1 and P_3 for e_2 lead to P_3 for **let**. Trivial from the definition of $\Gamma_1 \rhd^{L_2} \Gamma_2$.
2. P_4 for e_1 and P_4 for e_2 lead to P_4 for **let**. Trivial from the definition of fv.
3. Assuming P_1 for **let** (for E, h, k, h'', v_2), we show P_1 for e_1 (for E, h, k, h', v_1) and P_1 for e_2 (for $E \cup [x_1 \mapsto v_1], h', k, h''$) by using the operational semantics rule Let_1.
4. Assuming P_2 for **let** (for $\Gamma = \Gamma_1 \rhd^{L_2} \Gamma_2, E$), we show P_2 for e_1 (for Γ_1, E) and P_2 for e_2 (for $\Gamma_2 + [x_1 : s], E \cup [x_1 \mapsto v_1]$). This comes from $\mathit{dom}(\Gamma_1), \mathit{dom}(\Gamma_2) \subseteq \mathit{dom}(\Gamma_1 \rhd^{L_2} \Gamma_n 2)$.
5. By hypothesis, we get P_5, P_6, and $P_7 \wedge P_8 \rightarrow P_9$ both for e_1 and e_2.
6. P_5 and P_6 for e_1, and P_5 and P_6 for e_2 lead to P_5 and P_6 for **let**. We must prove:

$$
\begin{aligned}
P_5 &\equiv \forall x \in \mathit{dom}(E),\ z \in L \,.\, \Gamma[z] = d \ \wedge\ \mathit{recReach}(E, z, h) \cap \mathit{closure}(E, x, h) \neq \emptyset \\
&\qquad \rightarrow x \in \mathit{dom}(\Gamma) \wedge \Gamma[x] \neq s \\
P_6 &\equiv \forall x \in \mathit{dom}(E) \,.\, \mathit{closure}\ (E, x, h) \not\equiv \mathit{closure}\ (E, x, h'') \\
&\qquad \rightarrow x \in \mathit{dom}(\Gamma) \wedge \Gamma[x] \neq s
\end{aligned}
$$

First we prove P_5. Let us assume $z \in L$, $x \in \mathit{dom}(E)$, $\Gamma[z] = d$, and $\mathit{recReach}(E, z, h) \cap \mathit{closure}(E, x, h) \neq \emptyset$. We distinguish the following cases:

$z \in L_1, \Gamma_1[z] = d$ Then, by P_5 of e_1 and the triangle properties, we conclude $x \in \mathit{dom}(\Gamma) \wedge \Gamma[x] \neq s$.

$z \in L_1, \Gamma_1[z] = s$ Then, by the triangle properties $\Gamma_2[z] = d$ and $z \in L_2$. By P_6 of e_1 we have $\mathit{closure}\ (E, z, h) \equiv \mathit{closure}\ (E, z, h')$. Let us assume $\mathit{closure}\ (E, x, h) \equiv \mathit{closure}\ (E, x, h')$. Otherwise by P_6 of e_1 we would directly have the conclusion. Then $\mathit{recReach}(E, z, h) \cap \mathit{closure}(E, x, h) \neq \emptyset$ implies $\mathit{recReach}(E, z, h') \cap \mathit{closure}(E, x, h') \neq \emptyset$, and by P_5 of e_2, and the triangle properties, we conclude $x \in \mathit{dom}(\Gamma) \wedge \Gamma[x] \neq s$.

$z \in L_2, \Gamma_2[z] = d$ We distinguish here two subcases:

$\mathit{closure}\ (E, z, h) \equiv \mathit{closure}\ (E, z, h')$ Following the same reasoning as above, by P_5 of e_2 we would have the conclusion.

$\mathit{closure}\ (E, z, h) \not\equiv \mathit{closure}\ (E, z, h')$ Then, by P_6 of e_1 we would have $z \in \mathit{dom}(\Gamma_1) \wedge \Gamma_1[z] \neq s$. This combination of marks for z is forbidden by the operator $\rhd^{L_2}$. So this case is not possible.

For P_6, and assuming $\mathit{closure}\ (E, x, h) \not\equiv \mathit{closure}\ (E, x, h'')$, we distinguish two cases:

(a) $\mathit{closure}\ (E, x, h) \not\equiv \mathit{closure}\ (E, x, h')$. Then, by P_6 of e_1 we have $x \in \mathit{dom}(\Gamma_1) \wedge \Gamma_1[x] \neq s$, and by the triangle properties we have the conclusion.

(b) $\mathit{closure}(E, x, h) \equiv \mathit{closure}(E, x, h') \wedge \mathit{closure}(E, x, h') \not\equiv \mathit{closure}(E, x, h'')$. Then, by P_6 of e_2 we have $x \in \mathit{dom}(\Gamma_2) \wedge \Gamma_2[x] \neq s$, and by the triangle properties we have the conclusion.

7. P_7 for **let** (i.e. $S_{L, \Gamma_1 \rhd^{L_2} \Gamma_2, E, h} \cap R_{L, \Gamma_1 \rhd^{L_2} \Gamma_2, E, h} = \emptyset$), and P_5 for e_1 lead to P_7 for e_1 (i.e. $S_{L_1, \Gamma_1, E, h} \cap R_{L_1, \Gamma_1, E, h} = \emptyset$). The definitions of S_1 and R_1 are as follows:

(a) $S_1 = S_{1s} \cup S_{1r} \cup S_{1d}$, where:

$$\begin{aligned}
S_{1s} &\stackrel{\text{def}}{=} \textstyle\bigcup_{x \in L_1 \wedge \varGamma_1[x]=s \wedge \varGamma[x]=s} \{\mathit{closure}(E,x,h)\}, \quad S_{1s} \subseteq S\\
S_{1r} &\stackrel{\text{def}}{=} \textstyle\bigcup_{x \in L_1 \wedge \varGamma_1[x]=s \wedge \varGamma[x]=r} \{\mathit{closure}(E,x,h)\},\\
S_{1d} &\stackrel{\text{def}}{=} \textstyle\bigcup_{x \in L_1 \wedge \varGamma_1[x]=s \wedge \varGamma[x]=d} \{\mathit{closure}(E,x,h)\},
\end{aligned}$$

(b) $R_1 = \bigcup_{x \in L_1 \wedge \varGamma_1[x]=d} \{p \in \mathit{live}(E, L_1, h) \mid p \rightarrow_h^* \mathit{recReach}(E,x,h)\}$
We have the inclusion $R_1 \subseteq R$ because $\triangleright^{L_2}$ ensures that $\varGamma_1[x] = d$ implies $\varGamma[x] = d$.

Then, $S \cap R \neq \emptyset$ implies $S_{1s} \cap R_1 \neq \emptyset$. We must show now $(S_{1r} \cup S_{1d}) \cap R_1 = \emptyset$. This follows from the hypothesis $e_1, \varSigma_M : [\![L_1, \varGamma_1]\!]$. Should this set be non-empty, then there would exist $x \in L_1 \subseteq \mathit{dom}(E)$, $z \in L_1$ such that $\varGamma_1[z] = d, \varGamma_1[x] = s$, and $\mathit{recReach}(E,z,h) \cap \mathit{closure}(E,x,h) \neq \emptyset$. Then, by P_5 of e_1 we would have the contradiction $\varGamma_1[x] \neq s$.

8. Assuming P_8 for **let**, i.e. $\mathit{closed}(E, L, h)$, leads to P_8 for e_1, i.e. $\mathit{closed}(E, L_1, h)$. Trivial from $L_1 \subseteq L$.
9. Now, by the hypothesis $e_1, \varSigma_M : [\![L_1, \varGamma_1]\!]$ we get P_9 for e_1.
10. P_5, P_6 for e_1, P_5, P_6 for e_2, and P_7 for **let** lead to P_7 for e_2. We must show:

$$S_{L_2, \varGamma_2 + [x_1:s], E \cup [x_1 \mapsto v_1], h'} \cap R_{L_2, \varGamma_2 + [x_1:s], E \cup [x_1 \mapsto v_1], h'} = \emptyset$$

The two sets associated to e_2 are as follows:

(a) $S_2 = S_{2s} \cup \mathit{closure}(v_1, h')$, being $S_{2s} \bigcup_{x \in L_2 - \{x_1\} \wedge \varGamma_2[x]=s} \{\mathit{closure}(E', x, h')\}$. We have $S_{2s} \subseteq S$ because $\triangleright^{L_2}$ ensures that $\varGamma_2[x] = s$ implies $\varGamma[x] = s$.

(b) $R_2 = \bigcup_{x \in L_2 \wedge \varGamma_2[x]=d} \{p \in \mathit{live}(E', L_2, h') \mid p \rightarrow_{h'}^* \mathit{recReach}(E,x,h')\}$. We have the inclusion $R_2 \subseteq R$ because $\triangleright^{L_2}$ ensures that $\varGamma_2[x] = d$ implies $\varGamma[x] = d$ and $x \notin \mathrm{L}_1 \vee \varGamma_1[x] = s$, and by P_5 of e_1 as before.

So, $S \cap R \neq \emptyset$ implies $S_{2s} \cap R_2 \neq \emptyset$. We must show now $\mathit{closure}(v_1, h') \cap R_2 = \emptyset$. Should this not be the case, then by P_5 of e_2 we would have for x_1 a mark different from s which contradicts the hypothesis of the theorem.

11. Assuming P_8 for **let**, and by P_6, P_9 for e_1 we get P_8 for e_2, i.e. $\mathit{closed}(E \cup [x_1 \mapsto v_1], L_2, h')$. We prove first

$$\mathit{live}\ (E_2, L_2 - \{x_1\}, h) \equiv \mathit{live}\ (E_2, L_2 - \{x_1\}, h')$$

Should this not be true, by P_6 of e_1 we would have $\varGamma_1[z] \neq s$ for some variable $z \in L_2 - \{x_1\}$. This would contradict the definition of $\triangleright^{L_2}$. In these conditions, P_8 for **let** and P_9 for e_1 imply P_8 for e_2.

12. Now, by the hypothesis $e_2, \varSigma_M : [\![L_2, \varGamma_2 + [x_1 : s]]\!]$ we get P_9 for e_2.
13. Finally, using P_9 for e_2, it is trivial to show P_9 for **let**.

□

$\boxed{\textbf{\textit{LET1}}_{\boldsymbol{C}}}$ The rule to be proved is:

$$\frac{L_1 = \{\overline{a_i}^n\} \quad \varGamma_1 = [\overline{a_i \mapsto s}^n] \quad x_1 \notin L_1 \quad e_2, \varSigma_M : [\![L_2, \varGamma_2 + [x_1 : s]]\!] \quad \mathit{def}(\varGamma_1 \triangleright^{L_2} \varGamma_2)}{\textbf{let}\ x_1 = C\ \overline{a_i}^n @r\ \textbf{in}\ e_2, \varSigma_M : [\![L_1 \cup (L_2 - \{x_1\}), \varGamma_1 \triangleright^{L_2} \varGamma_2]\!]}$$

Proof. The steps are the following:

1-5 These steps are simplified versions of the equally numbered steps of *LET1*, because only the subexpression e_2 is involved.

6. P_5 and P_6 for e_2 leads to P_5 and P_6 for **let**. We must prove:

$$\begin{array}{l} (\forall x \in dom(E),\ z \in L\ .\ \Gamma[z] = d\ \wedge\ recReach(E,z,h) \cap closure(E,x,h) \neq \emptyset \\ \qquad \rightarrow x \in dom(\Gamma) \wedge \Gamma[x] \neq s) \\ \wedge\ (\forall x \in dom(E)\ .\ closure\ (E,x,h) \not\equiv closure\ (E,x,h') \\ \qquad \rightarrow x \in dom(\Gamma) \wedge \Gamma[x] \neq s) \end{array}$$

Let us call $h^+ = h \uplus [p \mapsto (j, C\ \overline{v_i})]$, where $fresh(p)$, $E(r) = j$, and for all i, $v_i = E(a_i)$. We know:

$$\begin{array}{l} (\forall x \in dom(E) \cup \{x_1\},\ z \in L_2\ .\ \Gamma_2[z] = d\ \wedge\ recReach(E,z,h^+) \cap closure(E,x,h^+) \neq \emptyset \\ \qquad \rightarrow x \in dom(\Gamma_2) \wedge \Gamma_2[x] \neq s) \\ \wedge\ (\forall x \in dom(E) \cup \{x_1\}\ .\ closure\ (E,x,h^+) \not\equiv closure\ (E,x,h') \\ \qquad \rightarrow x \in dom(\Gamma_2) \wedge \Gamma_2[x] \neq s) \end{array}$$

By being $fresh(p)$ and h^+ a conservative extension of h, we have:

$$closure(E,x,h^+) \equiv closure(E,x,h) \wedge recReach(E,z,h^+) \equiv recReach(E,z,h)$$

The rest follows from the properties of $\triangleright^{L_2}$.

7-10 These are simplified versions of the steps (10) to (13) of *LET1*.

□

$\boxed{\textbf{\textit{LET2}}}$ The rule to be proved is:

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

Proof. We must prove $\forall E\ h\ k\ h''\ v_2 . (P_1 \wedge P_2 \rightarrow P_3 \wedge P_4 \wedge P_5 \wedge P_6 \wedge (P_7 \wedge P_8 \rightarrow P_9))$ for the **let**. The steps are the following:

1-9 The steps (1) to (9) are identical to the equally numbered steps of *LET1*.

10. P_5, P_6 for e_1, P_5, P_6 for e_2, and P_7 for **let** lead to P_7 for e_2. We must show:

$$S_{L_2, \Gamma_2 + [x_1 : d], E \cup [x_1 \mapsto v_1], h'} \cap R_{L_2, \Gamma_2 + [x_1 : d], E \cup [x_1 \mapsto v_1], h'} = \emptyset$$

The two sets associated to e_2 are as follows:

(a) $S_2 = \bigcup_{x \in L_2 \wedge \Gamma_2[x] = s} \{closure(E,x,h')\}$, $\quad S_2 \subseteq S$. This inclusion is because $\triangleright^{L_2}$ ensures that $\Gamma_2[x] = s$ implies $\Gamma[x] = s$, and because all values $\{E(x) \mid x \in L_2 \wedge \Gamma_2[x] = s\}$ in h, either they have not been used in e_1, or they have been used in read-only mode and are still in h'.

(b) $R_2 = R_{2x_1} \cup R_{2d}$, where:

$$R_{2x_1} \stackrel{\text{def}}{=} \{p \in \mathit{live}(E', L_2, h') \mid p \rightarrow^*_{h'} \mathit{recReach}(E', x_1, h')\}$$
$$R_{2d} = \bigcup_{x \in L_2 \wedge \Gamma_2[x]=d} \{p \in \mathit{live}(E, L_2, h') \mid p \rightarrow^*_{h'} \mathit{recReach}(E, x, h')\}$$

We have $R_{2d} \subseteq R$ because $\triangleright^{L_2}$ ensures that $\Gamma_2[x] = d$ implies $\Gamma[x] = d$, and because all values $\{E(x) \mid x \in L_2 \wedge \Gamma_2[x] = d\}$ in h, either they have not been used in e_1, or they have been used in read-only mode and are still in h'.
Then, $R_{2d} \cap S_2 = \emptyset$ trivially holds. We must show $R_{2x_1} \cap S_2 = \emptyset$. This follows from the hypothesis $e_2, \Sigma_M : [\![L_2, \Gamma'_2]\!]$, being $\Gamma'_2[x_1] = d$.

11-13 These steps are identical to the equally numbered steps of *LET1*.

□

$\boxed{\boldsymbol{LET2_C}}$ The rule to be proved is:

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

Proof. The steps are the following:

1-5 These steps are simplified versions of the equally numbered steps of *LET1*, because only the subexpression e_2 is involved.
6. P_5 and P_6 for e_2 leads to P_5 and P_6 for **let**. This step is identical to the step (6) of *LET1*$_C$.
7-10 These are simplified versions of the steps (10) to (13) of *LET2*.

□

$\boxed{\boldsymbol{CASE}}$ The rule to be proved is:

$$\frac{\begin{array}{ll}\forall i\,.\,(e_i, \Sigma_M : [\![L_i, \Gamma_i]\!] \quad \Gamma_i[\overline{x_{ij}}] \neq d) & \Gamma \supseteq \bigotimes_i(\Gamma_i \backslash \{\overline{x_{ij}}\}) \\ L = \{x\} \cup (\bigcup_i(L_i - \{\overline{x_{ij}}\})) & x \in \mathit{dom}(\Gamma) \quad \Gamma \text{ well-formed}\end{array}}{\textbf{case}\ x\ \textbf{of}\ \overline{C_i \overline{x_{ij}} \rightarrow e_i}, \Sigma_M : [\![L, \Gamma]\!]}$$

Proof. We must prove $P_3 \wedge P_4 \wedge (\forall E\ h\ k\ h'\ v\,.\,P_1 \wedge P_2 \rightarrow P_5 \wedge P_6 \wedge (P_7 \wedge P_8 \rightarrow P_9))$ for the **case**. The steps are the following:

1. P_3 for all the e_i, and the premises given for L, x and Γ lead to P_3 for **case**.
2. P_4 for all the e_i and the definition of *fv* lead to P_4 for **case**.
3. Assuming P_1 for **case** (for E, h, k, h', v), we show P_1 for one e_i (for E_i, h, k, h', v), being C_i the constructor matched by x. It suffices to use the operational semantics rule *Case*.
4. Assuming P_2 for **case** (for Γ, E), we show P_2 for e_i (for Γ_i, E_i). This comes from $\mathit{dom}(\Gamma_i \backslash \{\overline{x_{ij}}\}) \subseteq \mathit{dom}(\Gamma)$.
5. By the hypothesis $e_i, \Sigma_M : [\![L_i, \Gamma_i]\!]$ we get P_5, P_6, and $P_7 \wedge P_8 \rightarrow P_9$ for e_i.

6. The premise 'Γ well-formed' directly leads to P_5 for **case**.
7. P_6 for the executed e_i leads to P_6 for **case**. We must prove:

$$\begin{aligned} P_6 \equiv \forall x \in \mathit{dom}(E) \,.\, \mathit{closure}\,(E, x, h) \not\equiv \mathit{closure}\,(E, x, h') \\ \rightarrow x \in \mathit{dom}(\Gamma) \wedge \Gamma[x] \neq s \end{aligned}$$

   It is obtained by using that $x \in \mathit{dom}(E)$ implies $x \in \mathit{dom}(E_i)$, and $\Gamma_i[x] \neq s$ implies $\Gamma[x] \neq s$.
8. P_7 for **case** leads to P_7 for the executed e_i. Let us call S, S_i, R and R_i to respectively $S_{L,\Gamma,E,h}, S_{L_i,\Gamma_i,E_i,h}, R_{L,\Gamma,E,h}$ and $R_{L_i,\Gamma_i,E_i,h}$. We decompose $S_i = S'_i \cup S''_i$ and $R_i = R'_i \cup R''_i$, being their definitions as follows:

$$\begin{aligned} S'_i &\overset{\text{def}}{=} \textstyle\bigcup_{z \in L_i \cap L, \Gamma_i[z]=s} \{\mathit{closure}(E_i, z, h)\} \\ S''_i &\overset{\text{def}}{=} \textstyle\bigcup_{z \in L_i \cap \overline{x_{ij}}, \Gamma_i[z]=s} \{\mathit{closure}(E_i, z, h)\} \\ R'_i &\overset{\text{def}}{=} \textstyle\bigcup_{z \in L_i \cap L, \Gamma_i[z]=d} \{p \in \mathit{live}(E_i, L_i, h) \mid p \rightarrow^*_h \mathit{recReach}(E_i, z, h)\} \\ R''_i &\overset{\text{def}}{=} \textstyle\bigcup_{z \in L_i \cap \overline{x_{ij}}, \Gamma_i[z]=d} \{p \in \mathit{live}(E_i, L_i, h) \mid p \rightarrow^*_h \mathit{recReach}(E_i, z, h)\} \end{aligned}$$

   Obviously, $S'_i \subseteq S$ and $R'_i \subseteq R$. So, $S \cap R = \emptyset$ implies $S'_i \cap R'_i = \emptyset$. Also, no pattern x_{ij} has a mark $\Gamma_i[x_{ij}] = d$. This is forbidden by the type system and we include this fact as a premise of the rule. So, the variables with a condemned mark should belong to L, and consequently $R''_i = \emptyset$. Now, we distinguish two cases according to the mark of the discriminant x:

   $\Gamma[x] = s$ For those x_{ij} such that $\Gamma_i[x_{ij}] = s$ we have $\mathit{closure}\,(E_i, x_{ij}, h) \subseteq \mathit{closure}\,(E, x, h) \subseteq S$. So, $S''_i \subseteq S$, concluding that $S_i \cap R_i = \emptyset$.

   $\Gamma[x] \neq s$ For each pattern x_{ij} such that $\Gamma_i[x_{ij}] = s$ we distinguish two cases:
   (a) There exists $z \in L$ with $\Gamma_i[z] = \Gamma[z] = d$ such that $\mathit{closure}(E_i, x_{ij}, h) \cap \mathit{recReach}\,(E_i, z, h) \neq \emptyset$. By P_5 for e_i we would have $\Gamma_i[x_{ij}] \neq s$, a contradiction. So, this case is not possible.
   (b) For all $z \in L$ with $\Gamma_i[z] = \Gamma[z] = d$ we have $\mathit{closure}\,(E_i, x_{ij}, h) \cap \mathit{recReach}\,(E_i, z, h) = \emptyset$. By definition of R_i, the contribution of x_{ij} to S''_i does not intersect with R_i. So, also in this case we get $S_i \cap R_i = \emptyset$.
9. P_8 for **case** leads to P_8 for all the e_i. By $L_i \subseteq L \cup \{\overline{x_{ij}}\}$ and $E_i(x_{ij}) = b_j$ we have $\mathit{closure}(E_i, L_i, h) \subseteq \mathit{closure}(E, L, h)$ and then $\mathit{closed}(E, L, h)$ implies $\mathit{closed}(E_i, L_i, h)$.
10. Now, by the hypothesis $e_i, \Sigma_M : [\![L_i, \Gamma_i]\!]$ we get P_9 for e_i.
11. Finally, P_9 for e_i is the same property as P_9 for **case**.

□

$\boxed{\boldsymbol{CASE_L}}$ The rule to be proved is:

$$\frac{\forall i \,.\, (e_i, \Sigma_M : [\![L_i, \Gamma_i]\!]) \quad \Gamma = \bigotimes_i \Gamma_i \quad \Gamma[x] = s \quad L = \{x\} \cup (\bigcup_i L_i)}{\textbf{case}\ x\ \textbf{of}\ \{\overline{c_i \rightarrow e_i}^n; \_ \rightarrow e_{n+1}\}, \Sigma_M : [\![L, \Gamma]\!]}$$

Proof. The steps are the same as the steps for *Case* but they are much simpler because there are no patterns and we can also assume $\Gamma[x] = s$.

□

$\boxed{\textbf{\textit{CASE!}}}$ The rule to be proved is:

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

Proof. The steps are the following:

1-5 These steps are small variations of the equally numbered steps of *CASE*.

6. The premise 'Γ well-formed' directly leads to P_5 for **case**!.
7. P_6 for the executed e_i leads to P_6 for **case**!. We must prove:

$$\begin{array}{l} P_6 \equiv \forall y \in \mathit{dom}(E) \,.\, \mathit{closure}\ (E, y, h^+) \not\equiv \mathit{closure}\ (E, y, h') \\ \qquad \to y \in \mathit{dom}(\Gamma) \wedge \Gamma[y] \neq s \end{array}$$

where $h^+ = h \uplus [p \mapsto w]$ and $E(x) = p$. Let us assume $y \in \mathit{dom}(E)$ such that $\mathit{closure}\ (E, y, h^+) \not\equiv \mathit{closure}\ (E, y, h')$. We distinguish two cases:

(a) $\mathit{closure}(E, y, h^+) \not\equiv \mathit{closure}(E, y, h)$ In this case we have $\mathit{recReach}(E, x, h) \cap \mathit{closure}\ (E, y, h) \neq \emptyset$ and by P_5 for **case**! we get $y \in \mathit{dom}(\Gamma) \wedge \Gamma[y] \neq s$.

(b) $\mathit{closure}(E, y, h^+) \equiv \mathit{closure}(E, y, h)$ Then it must hold $\mathit{closure}(E, y, h) \not\equiv \mathit{closure}(E, y, h')$. Then, by P_6 for the executed e_i we have $y \in \mathit{dom}(\Gamma_i) \wedge \Gamma_i[y] \neq s$. By the properties of operator $\otimes$ we get $\Gamma[y] \neq s$.

8. P_5, P_7 for **case**, and P_5 for the executed e_i lead to P_7 for e_i. Let us call S, S_i, R and R_i to respectively $S_{L,\Gamma,E,h}, S_{L_i,\Gamma_i,E_i,h'}, R_{L,\Gamma,E,h}$ and $R_{L_i,\Gamma_i,E_i,h'}$. We decompose $S_i = S'_i \cup S''_i$ and $R_i = R'_i \cup R''_i$, being their definitions as follows:

$$\begin{array}{lcl} S'_i & \stackrel{\text{def}}{=} & \bigcup_{z \in L_i \cap L, \Gamma_i[z]=s} \{\mathit{closure}(E_i, z, h')\} \\ S''_i & \stackrel{\text{def}}{=} & \bigcup_{z \in L_i \cap \overline{x_{ij}}, \Gamma_i[z]=s} \{\mathit{closure}(E_i, z, h')\} \\ R'_i & \stackrel{\text{def}}{=} & \bigcup_{z \in L_i \cap L, \Gamma_i[z]=d} \{p \in \mathit{live}(E_i, L_i, h') \mid p \to^*_{h'} \mathit{recReach}(E_i, z, h')\} \\ R''_i & \stackrel{\text{def}}{=} & \bigcup_{z \in L_i \cap \overline{x_{ij}}, \Gamma_i[z]=d} \{p \in \mathit{live}(E_i, L_i, h') \mid p \to^*_{h'} \mathit{recReach}(E_i, z, h')\} \end{array}$$

By P_5 for **case**, clearly $E(x) \notin S$, so $S'_i \subseteq S$. Also clearly, $E(x) \in R$, so, it is possible that some $p \in R'_i$ such that $p \to^*_h \mathit{recReach}(E_i, z, h)$, for some $z \in L_i \cap L$ with $\Gamma[z] = d$, now it cannot reach it in h' because the path in h was through $E(x)$. However, the rest of the paths are still in h', so $R'_i \subseteq R$. Then, $S \cap R = \emptyset$ implies $S'_i \cap R'_i = \emptyset$.
By P_5 for e_i, $S''_i \cap R_i = \emptyset$. Otherwise, there would exist x_{ij} shuch that $\Gamma_i[x_{ij}] = s$ and $E_i(x_{ij})$ would reach a recursive descendant of a condemned location. Property P_5 would imply $\Gamma_i[x_{ij}] \neq s$.
It remains to be shown $S'_i \cap R''_i = \emptyset$. This comes from the premise $\Gamma_i[x_{ij}] = d \to j \in \mathit{RecPos}(C_i)$, which implies $R''_i \subseteq R$.

9. P_5 and P_8 for **case** lead to P_8 for all the e_i. Eventhough $L_i \subseteq L \cup \{\overline{x_{ij}}\}$ and $E_i(x_{ij}) = b_j$, we do not have in general $\mathit{closure}(E_i, L_i, h') \subseteq \mathit{closure}(E, L, h)$ because the heap h' does not include the cell of h pointed to by $E(x)$. By P_5 for **case** we have that all variables z sharing $E(x)$ are marked with $\Gamma[z] \neq s$. The last rule's premise establishes that these variables do not belong to any L_i. In these conditions, $\mathit{closed}(E, L, h)$ implies $\mathit{closed}(E_i, L_i, h')$.
10. Now, by hypothesis $e_i, \Sigma_M : [\![L_i, \Gamma_i]\!]$ and then we get P_9 for the executed e_i.
11. Finally, P_9 for e_i is the same property as P_9 for **case**.

□

$\boxed{\boldsymbol{APP}_{\mathit{nonrec}}}$ The rule to be proved is:

$$\frac{\begin{array}{lll} \models \Sigma_M & \Sigma_M(g) = \overline{m_i}^n & L = \{\overline{a_i}^n\} \\ \Gamma_0 = \bigoplus_{i=1}^n [a_i : m_i] \text{ defined} & \Gamma \supseteq \Gamma_0 \text{ well-formed} & \end{array}}{g\ \overline{a_i}^n @\ \overline{r'_j}^m, \Sigma_M : [\![L, \Gamma]\!]}$$

Proof. In the first place, $\models \Sigma_M$ implies that for every certified function g with a mark signature $\Sigma_M(g) = \overline{m_i}^n$, $m_i \in \{s, d\}$, the following lemma has been proved:

Lemma 2. *If* $g\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_g$ *is* g*'s definition, and* $\Sigma_M(g) = \overline{m_i}^n$ *then* $e_g, \Sigma_M : [\![L_g, \Gamma_g]\!]$*, being* $L_g = \{\overline{x_i}^n\}$ *and* $\Gamma_g \supseteq [\overline{x_i : m_i}^n]$*.*

We must prove $P_3 \wedge P_4 \wedge (\forall E\ h\ k\ h'\ v\ .\ P_1 \wedge P_2 \rightarrow P_5 \wedge P_6 \wedge (P_7 \wedge P_8 \rightarrow P_9))$ for the expression $g\ \overline{a_i}^n @\ \overline{r'_j}^m$. The steps are the following:

1. P_3 and P_4 for the application hold trivially.
2. Let us assume P_1 for $g\ \overline{a_i}^n$, i.e. $E \vdash h, k, g\ \overline{a_i}^n @\ \overline{r'_j}^m \Downarrow h' \mid_k, k, v$. This implies $E_0 \vdash h, k+1, e_g \Downarrow h', k+1, v$, being

$$E_0 = [\overline{x_i \mapsto E(a_i)}^n, \overline{r_j \mapsto E(r'_j)}^m, \mathit{self} \mapsto k+1]$$

by using the semantic rule *App*. Let us define $E_g = E + E_0$. As, the only free variables in e_g are the x_i, we trivially have $E_g \vdash h, k+1, e_g \Downarrow h', k'+1, v$. Then we get P_1 for e_g with $E_g, h, k+1, h'$, and v.
3. Let us define $\Gamma_g = \Gamma + [\overline{x_i : m_i}^n]$. Assuming P_2 for the application, we get P_2 for e_g, i.e. $\mathit{dom}(\Gamma_g) \subseteq \mathit{dom}(E_g)$.
4. Then, by $e_g, \Sigma_M : [\![L_g, \Gamma_g]\!]$ we get P_5, P_6, and $P_7 \wedge P_8 \rightarrow P_9$ for e_g.
5. P_5 for $g\ \overline{a_i}^n$, i.e.

$$\begin{array}{l}\forall x \in \mathit{dom}(E).\ z \in L\ .\ \Gamma[z] = d\ \wedge\ \mathit{recReach}(E, z, h) \cap \mathit{closure}(E, x, h) \neq \emptyset \\ \qquad \rightarrow x \in \mathit{dom}(\Gamma) \wedge \Gamma[x] \neq s\end{array}$$

is guaranteed by the premise 'Γ well-formed'.
6. P_6 for e_g implies P_6 for $g\ \overline{a_i}^n$, provided that removing from h' the region $k+1$ does not modify $\mathit{closure}\,(E, \mathit{dom}(E), h)$. This will need a lemma establishing that deallocating the *self* region will not create dangling pointers in the heap. This is done in Sec. 4.

7. Let us assume P_7 for $g\ \overline{a_i}^n$, i.e. $S_{L,\Gamma,E,h} \cap R_{L,\Gamma,E,h} = \emptyset$. By the properties of $\bigoplus$, we may have $E_g(x_i) = E_g(x_j)$ with $i \neq j$ only when $\Gamma_g[x_i] = \Gamma_g[x_j] = s$. This leads to $S_{L_g,\Gamma_g,E_g,h} = S_{L,\Gamma,E,h}$. Also, we have $R_{L_g,\Gamma_g,E_g,h} = R_{L,\Gamma,E,h}$ because L, L_g, and Γ, Γ_g, and E, E_g are essentially the same. So, property P_7 holds for e_g.
8. Let us assume P_8 for $g\ \overline{a_i}^n$, i.e. $\mathit{closed}(E, L, h)$. Then P_8 for e_g trivially holds.
9. Then we get P_9 for e_g, i.e. $\mathit{closed}(v, h')$.
10. In order to show P_9 for $g\ \overline{a_i}^n$, i.e. $\mathit{closed}(v, h' \mid_k)$, we must show that removing from h' the region $k+1$ does not destroy the property $\mathit{closed}(v, h')$. This is done in Sec. 4.

$\square$

$\boxed{\textbf{\textit{FUN}}}$ The property $\models \Sigma_M$ (Σ_M is valid) is inductively generated by the *FUN* rule. The current mark environment Σ_M is an explicit argument of the $e, \Sigma_M : [\![L, \Gamma]\!]$ relation. The proof rule defines when a list $\overline{m_i}^n$ of marks is valid for a function f. As f may be recursive, the mark environment is extended with the assumption we are trying to prove. This rule is proved by induction on the depth of the call chains to f launched from e_f. We must prove:

$$\frac{\begin{array}{c}\models \Sigma \quad f\ \overline{x_i}^n @\ \overline{r_j}^m = e_f \quad L_f = \{\overline{x_i}^n\} \quad \Gamma_f \supseteq [\overline{x_i \mapsto m_i}^n] \\ \Gamma_f \text{ well-formed} \quad e_f, \Sigma \cup \{f \mapsto \overline{m_i}^n\} : [\![L_f, \Gamma_f]\!]\end{array}}{\models \Sigma_M \cup \{f \mapsto \overline{m_i}^n\}}\ \mathit{FUN}$$

We give previously some infrastructure for the Lemma. First, we enrich the big-step semantics with a side-effect counter n_f counting the *maximum length* of the recursive call chains to a given function f. Then, $E \vdash h, k, e \Downarrow_f h', k, v, n_f$ means that, in the derivation tree of expression e, there are 0 or more direct calls to f, and the longest call-chain involving f and starting at e has a length $n_f \geq 0$. A length $n_f = 0$ means that there are no calls to f during e's evaluation.

Definition 5. *We define a restricted big-step semantics with an upper bound n to the longest chain of f's:*

$$E \vdash h, k, e \Downarrow_{f,n} h', k, v \overset{def}{=} E \vdash h, k, e \Downarrow_f h', k, v, n_f \wedge n_f \leq n$$

If we write $P1(f, n, \Sigma)$ we refer to the following property:

$$E \vdash h, k, e \Downarrow_{f,n} h', k, v \ \wedge\ g \neq f \text{ may be called by } e \text{ implies } g \in \mathit{dom}(\Sigma)$$

which is similar to $P1(\Sigma)$ of Def. 3 but using the $\Downarrow_{f,n}$ relation instead of the $\Downarrow$ one. The following Lemma is proved by induction on n.

Lemma 3. $E \vdash h, k, e \Downarrow h', k, v$ *if and only if* $\exists n \,.\, E \vdash h, k, e \Downarrow_{f,n} h', k, v$.

We modify definitions 3 and 4 in order to introduce the depth of the derivation:

Definition 6. *Given the properties $P1(f, n, \Sigma), P2, \ldots, P9$ as in Def. 3 except for $P1$, we say that the subexpression e of f's body* satisfies *the assertion $[\![L, \Gamma]\!]$ up to depth n in the context of Σ, denoted $e, \Sigma :_{f,n} [\![L, \Gamma]\!]$, if*

$$P3 \wedge P4 \wedge (\forall E\ h\ k\ h'\ v \,.\, P1(f, n, \Sigma) \wedge P2 \rightarrow P5 \wedge P6 \wedge (P7 \wedge P8 \rightarrow P9))$$

Definition 7. *A function mark environment Σ is* valid *up to depth n for function f, denoted $\models_{f,n} \Sigma$, if it can be deduced from the following inductive rules.*

1. *The empty environment is valid, i.e. $\models_{f,n} \emptyset$.*
2. *If $\models \Sigma$ and $\forall n' < n.\ \models_{f,n'} \Sigma \uplus \{f \mapsto \overline{m_i}\}$ hold, function f is defined as $f\ \overline{x_i} @\ \overline{r_j} = e_f$, $L_f = \overline{x_i}$, $\Gamma_f = [\overline{x_i \mapsto m_i}]$, and $e_f, \Sigma :_{f,n} [\![L_f, \Gamma_f]\!]$ holds, then $\models_{f,n} \Sigma \uplus \{f \mapsto \overline{m_i}\}$ also holds.*

We prove now that the premises of the *FUN* rule lead to the following intermediate assertion:

Lemma 4. *If $\models \Sigma$, $f\ \overline{x_i} @\ \overline{r_j} = e_f$, $L_f = \overline{x_i}$, $\Gamma_f = [\overline{x_i \mapsto m_i}]$, and $e_f, \Sigma \uplus \{f \mapsto \overline{m_i}\} : [\![L_f, \Gamma_f]\!]$ hold, then*

$$\forall n\ .\ \models_{f,n} \Sigma \uplus \{f \mapsto \overline{m_i}\}$$

also holds.

Proof. By induction on n and then by cases on e_f.

$\boxed{n = 0}$ We must show $\models_{f,0} \Sigma \uplus \{f \mapsto \overline{m_i}\}$. By Def. 7(2), the only not implied premise is $e_f, \Sigma :_{f,0} [\![L_f, \Gamma_f]\!]$. By Def. 6, this is equivalent to:

$$P3 \wedge P4 \wedge (\forall E\ h\ k\ h'\ v\ .\ P1(f, 0, \Sigma) \wedge P2 \rightarrow P5 \wedge P6 \wedge (P7 \wedge P8 \rightarrow P9))$$

Now we proceed by cases on e_f:

$\boxed{e_f \neq f\ \overline{a_i}\ @\ \overline{r_j}}$ The premise $e_f, \Sigma \uplus \{f \mapsto \overline{m_i}\} : [\![L_f, \Gamma_f]\!]$ for $e_f \neq f\ \overline{a_i}\ @\ \overline{r_j}$ is equivalent to:

$$P3 \wedge P4 \wedge (\forall E\ h\ k\ h'\ v\ .\ P1(\Sigma) \wedge P2 \rightarrow P5 \wedge P6 \wedge (P7 \wedge P8 \rightarrow P9))$$

This guarantees the conclusion because, given $E\ h\ k\ h'\ v$ such that $P1(f, 0, \Sigma)$ holds, then $P1(\Sigma)$ also holds for the same $E\ h\ k\ h'\ v$.

$\boxed{e_f = f\ \overline{a_i}\ @\ \overline{r_j}}$ Then the assumption $P1(f, 0, \Sigma)$ does not hold and the implication $P1(f, 0, \Sigma) \wedge P2 \rightarrow \cdots$ is true. The premise $e_f, \Sigma \uplus \{f \mapsto \overline{m_i}\} : [\![L_f, \Gamma_f]\!]$, for $e_f = f\ \overline{a_i}\ @\ \overline{r_j}$, satisfies the conditions of Def. 3 and these guarantee $P3$ and $P4$.

$\boxed{n > 0}$ By induction hypothesis $\forall n' < n.\ \models_{f,n'} \Sigma \uplus \{f \mapsto \overline{m_i}\}$ holds, and we must show $\models_{f,n} \Sigma \uplus \{f \mapsto \overline{m_i}\}$. By Def. 7(2), this is equivalent to $\forall n' < n.\ \models_{f,n'} \Sigma \uplus \{f \mapsto \overline{m_i}\}$ and $e_f, \Sigma :_{f,n} [\![L_f, \Gamma_f]\!]$. The first conjunction is guaranteed by induction hypothesis. For the second one we proceed by cases on e_f:

$\boxed{e_f \neq f\ \overline{a_i}\ @\ \overline{r_j}}$ By Def. 3, the premise $e_f, \Sigma \uplus \{f \mapsto \overline{m_i}\} : [\![L_f, \Gamma_f]\!]$ is equivalent to:

$$P3 \wedge P4 \wedge (\forall E\ h\ k\ h'\ v . P1(\Sigma \uplus \{f \mapsto \overline{m_i}\}) \wedge P2 \rightarrow P5 \wedge P6 \wedge (P7 \wedge P8 \rightarrow P9))$$

We must show:

$$P3 \wedge P4 \wedge (\forall E\ h\ k\ h'\ v\ .\ P1(f, n, \Sigma) \wedge P2 \rightarrow P5 \wedge P6 \wedge (P7 \wedge P8 \rightarrow P9))$$

But $P1(f, n, \Sigma)$ for some $E\ h\ k\ h'\ v$ implies $P1(\Sigma \uplus \{f \mapsto \overline{m_i}\})$ for the same $E\ h\ k\ h'\ v$, so the implication holds.

$\boxed{e_f = f\ \overline{a_i}\ @\ \overline{r_j}}$ As $\forall n' < n.\ \models_{f,n'} \Sigma \uplus \{f \mapsto \overline{m_i}\}$ holds, by Def. 7 we can assume $\forall n' < n\ .\ e_f, \Sigma :_{f,n'} [\![L_f, \Gamma_f]\!]$ holds, and then forall $n' < n$, and for the given e_f, L_f, and Γ_f we have:

$$P3 \wedge P4 \wedge (\forall E\ h\ k\ h'\ v\ .\ P1(f, n', \Sigma) \wedge P2 \rightarrow P5 \wedge P6 \wedge (P7 \wedge P8 \rightarrow P9))$$

On the other hand, the premise $e_f, \Sigma \uplus \{f \mapsto \overline{m_i}\} : [\![L_f, \Gamma_f]\!]$, for $e_f = f\ \overline{a_i}\ @\ \overline{r_j}$, satisfies $L = \{\overline{a_i}\}$, $\Gamma \supseteq \bigoplus_{i=1}^{n}[a_i : m_i]$, and Γ well-formed, by Def. 3. We must show:

$$P3 \wedge P4 \wedge (\forall E\ h\ k\ h'\ v\ .\ P1(f, n, \Sigma) \wedge P2 \rightarrow P5 \wedge P6 \wedge (P7 \wedge P8 \rightarrow P9))$$

for L, Γ, and $f\ \overline{a_i}\ @\ \overline{r_j}$. Let $E\ h\ k\ h'\ v$ be such that $P1(f, n, \Sigma)$ holds for $f\ \overline{a_i}\ @\ \overline{r_j}$. Then $P1(f, n-1, \Sigma)$ holds for e_f and appropriate runtime environment, heaps, and value derived from $E\ h\ k\ h'\ v$. By reasoning in a similar way than in the APP_{nonrec} rule, we can show the conclusion.

Lemma 5. *If* $\forall n\ .\ \models_{f,n} \Sigma \uplus \{f \mapsto \overline{m_i}\}$ *holds, then* $\models \Sigma \uplus \{f \mapsto \overline{m_i}\}$ *also holds.*

Proof. By equational reasoning:

$$\begin{array}{ll}
 & \forall n\ .\ \models_{f,n} \Sigma \uplus \{f \mapsto \overline{m_i}\} \\
\equiv & \quad \{\text{By Def. 7(2)}\} \\
 & \forall n\ .\ (e_f, \Sigma :_{f,n} [\![L_f, \Gamma_f]\!] \wedge \forall n' < n.\ \models_{f,n'} \Sigma \uplus \{f \mapsto \overline{m_i}\} \wedge \models \Sigma) \\
\equiv & \quad \{\text{By Def. 6}\} \\
 & \forall n\ .\ (\forall n' < n.\ \models_{f,n'} \Sigma \uplus \{f \mapsto \overline{m_i}\} \wedge \models \Sigma \\
 & \qquad \wedge\ P3 \wedge P4 \wedge (\forall E\ h\ k\ h'\ v\ .\ P1(f, n, \Sigma) \wedge P2 \rightarrow P5 \wedge P6 \wedge (P7 \wedge P8 \rightarrow P9))) \\
\Rightarrow & \quad \{\text{By first-order logic}\} \\
 & \models \Sigma \wedge P3 \wedge P4 \wedge (\forall E\ h\ k\ h'\ v\ .\ (\exists n. P1(f, n, \Sigma)) \wedge P2 \rightarrow P5 \wedge P6 \wedge (P7 \wedge P8 \rightarrow P9)) \\
\equiv & \quad \{\text{By Lemma 3}\} \\
 & \models \Sigma \wedge P3 \wedge P4 \wedge (\forall E\ h\ k\ h'\ v\ .\ P1(\Sigma) \wedge P2 \rightarrow P5 \wedge P6 \wedge (P7 \wedge P8 \rightarrow P9)) \\
\equiv & \quad \{\text{By Def. 3}\} \\
 & \models \Sigma \wedge e_f, \Sigma : [\![L_f, \Gamma_f]\!] \\
\equiv & \quad \{\text{By Def. 4(2)}\} \\
 & \models \Sigma \uplus \{f \mapsto \overline{m_i}\}
\end{array}$$

By putting together lemmas 4 and 5, we get the proof of the *FUN* rule.

4 Region deallocation

In this section we present the proof-rules, proved correct in Isabelle/HOL, used to certify that region deallocation does not create dangling pointers in the heap. As before, the compiler delivers static information about the region types used by the program variables and expressions, and then the proof-rule relates this information to runtime properties about 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.

When dealing with 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. Given a type t, $t' = \mu(t)$ is the type used in the application of the instantiated function or constructor. This mapping should correctly map the types of the formal arguments to the types of the corresponding actual arguments.

Definition 8. *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 keeping the most general types of the functions already certified. For constructors, the compiler provides a global environment Γ giving its polymorphic most general type. If $\Gamma(C) = \overline{t_i}^n \rightarrow \rho \rightarrow T\ \overline{t_j}^l @\ \overline{\rho_i}^m$, the following property, satisfied by the type system, is needed for proving the proof-rules below:

Definition 9. *Predicate wellT$(\overline{t_i}^n, \rho, T\ \overline{t_j}^l @\ \overline{\rho_i}^m)$, read* well-typed*, is defined:*

$$\rho_m = \rho \ \wedge \ \bigcup_{i=1}^{n} \mathit{regions}\ (t_i) \subseteq \mathit{regions}\ (T\ \overline{t_j}^l @\ \overline{\rho_i}^m)$$

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

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

We introduce now a notion of consistency between the static information θ and the dynamic one $E,\ \eta,\ h,\ h'$. Essentially, consistency 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 11. *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. $\forall x \in \mathit{dom}(E)\ .\ \mathit{consistent}\ (\theta(x), \eta, E(x), h)$ *where:*

$$\begin{array}{lll}
\mathit{consistent}\ (B, \eta, c, h) & = \mathit{true} & \text{-- } B \textit{ is a basic type} \\
\mathit{consistent}\ (a, \eta, v, h) & = \mathit{true} & \text{-- } \textit{for any value } v \\
\mathit{consistent}\ (t, \eta, p, h) & = \mathit{true} & \textit{if } p \notin \mathit{dom}(h) \\
\mathit{consistent}\ (T\ \overline{t'_i}^m @\ \overline{\rho_j}^l, \eta, p, h) & \multicolumn{2}{l}{= \exists j\ C\ \overline{v_k}^n\ \mu\ \overline{t_{kC}}^n\ \overline{\rho_{jC}}^l\ .\ h(p) = (j, C\ \overline{v_k}^n)} \\
& \multicolumn{2}{l}{\wedge\ \rho_l \in \mathit{dom}(\eta)\ \wedge\ \eta(\rho_l) = j} \\
& \multicolumn{2}{l}{\wedge\ \Gamma(C) = \overline{t_{kC}}^n \to \rho_{lC} \to T\ \overline{t'_{iC}}^m @\ \overline{\rho_{jC}}^l} \\
& \multicolumn{2}{l}{\wedge\ \mathit{wellT}(\overline{t_{kC}}^n, \rho_{lC}, T\ \overline{t'_{iC}}^m @\ \overline{\rho_{jC}}^l)} \\
& \multicolumn{2}{l}{\wedge\ \mu(T\ \overline{t'_{iC}}^m @\ \overline{\rho_{jC}}^l) = T\ \overline{t'_i}^m @\ \overline{\rho_j}^l} \\
& \multicolumn{2}{l}{\wedge\ \forall k \in \{1..n\}\ .\ \mathit{consistent}\ (\mu(t_{kC}), \eta, v_k, h))}
\end{array}$$

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

Notice that the third clause of *consistent* states that a dangling pointer is consistent with any type t, mapping η, and heap h. This is needed because the notion of consistency is used to prove absence of dangling pointers in the *APP* rule of Sec. 3 and it is not known in advance whether dangling pointers may exist.

A judgement of the form $e, \Sigma_T \vdash \theta \leadsto t$ defines that, in the context of a *function typing environment* Σ_T, if expression e is evaluated with a runtime environment E, a heap h, and an admissible mapping η consistent with θ, then η, the final heap h', and the final value v are consistent with t. The following definitions make it precise the satisfaction of a typing static assertion and the validity of a function typing environment.

Definition 12. *An expression e belonging to f's body, $e \neq f\ \overline{a_i}\ @\ \overline{r_j}$, satisfies the pair (θ, t) in the context of the function typing environment Σ_T, denoted $e, \Sigma_T \vdash \theta \leadsto t$ if*

$$\begin{array}{lll}
\forall E\ h\ k\ h'\ v\ \eta\ . & E \vdash h, k, e \Downarrow h', k, v & \\
& \textit{and if } g \neq f \textit{ may be called by } e \textit{ then } g \in \mathit{dom}(\Sigma_T) & \text{-- } P1 \\
\wedge & (\mathit{fv}(e) \cup \mathit{fvregs}(e)) \subseteq \mathit{dom}(E) & \text{-- } P1' \\
\wedge & \mathit{dom}(E) \subseteq \mathit{dom}(\theta) & \text{-- } P2 \\
\wedge & \mathit{admissible}\ (\eta, k) & \text{-- } P3 \\
\wedge & \mathit{consistent}\ (\theta, \eta, E, h) & \text{-- } P4 \\
\to & \mathit{consistent}\ (t, \eta, v, h') & \text{-- } P5
\end{array}$$

If $e = f\ \overline{a_i}\ @\ \overline{r_j}$ and $f \in \mathit{dom}(\Sigma_T)$, we say that e satisfies the pair (θ, t) in the context of Σ_T, denoted also $e, \Sigma_T \vdash \theta \leadsto t$, if $\Sigma_T(f) = \overline{t_i}^n \to \overline{\rho_j}^m \to t_f$, $\rho^f_{\mathit{self}} \notin \mathit{regions}(t_f)$, and there exists μ such that $\mathit{argP}(\overline{\mu(t_i)}^n, \overline{\mu(\rho_j)}^m, \overline{a_i}^n, \overline{r_j}^m, \theta)$, and $t = \mu(t_f)$.

$$\frac{}{c, \Sigma_T \vdash \theta \leadsto B}\ \mathit{LIT} \qquad \frac{}{x, \Sigma_T \vdash \theta \leadsto \theta(x)}\ \mathit{VAR1} \qquad \frac{}{x!, \Sigma_T \vdash \theta \leadsto \theta(x)}\ \mathit{VAR3}$$

$$\frac{\theta(x) = T\ \overline{t_i}^m @\ \rho_1 \ldots \rho_l \quad \theta(r) = \rho'}{x@r, \Sigma_T \vdash \theta \leadsto T\ \overline{t_i}^m @\ \rho_1 \ldots \rho_{l-1}\rho'}\ \mathit{VAR2} \qquad \frac{e_1, \Sigma_T \vdash \theta \leadsto t_1 \quad e_2, \Sigma_T \vdash \theta \uplus [x_1 \mapsto t_1] \leadsto t_2}{\textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2 \vdash \theta \leadsto t_2}\ \mathit{LET}$$

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

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

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

$$\frac{\models \Sigma_T \backslash \{f\} \quad \Sigma_T(g) = \overline{t_i}^n \to \overline{\rho_j}^m \to t_g \quad \rho^g_{\mathit{self}} \notin \mathit{regions}\ (t_g) \quad \mathit{argP}\ (\overline{\mu(t_i)}^n, \overline{\mu(\rho_j)}^m, \overline{a_i}^n, \overline{r'_j}^m, \theta) \quad t = \mu(t_g)}{g\ \overline{a_i}^n @\ \overline{r'_j}^m, \Sigma_T \vdash \theta \leadsto t}\ \mathit{APP}_{\mathit{nonrec}}$$

$$\frac{\models \Sigma_T \quad f\ \overline{x_i}^n @\ \overline{r_j}^m = e_f \quad \theta_f = [\overline{x_i \mapsto t_i}^n, \overline{r_j \mapsto \rho_j}^m, \mathit{self} \mapsto \rho_{\mathit{self}}] \quad e_f, \Sigma_T \vdash_{\Sigma_T \cup \{f \mapsto \overline{t_i}^n \to \overline{\rho_j}^m \to t_f\}} \theta_f \leadsto t_f}{\models \Sigma_T \cup \{f \mapsto \overline{t_i}^n \to \overline{\rho_j}^m \to t_f\}}\ \mathit{FUN}$$

Fig. 9. Proof rules for region deallocation (each one is an Isabelle/HOL theorem)

Definition 13. *A function typing environment* Σ_T *is* valid*, denoted* $\models \Sigma_T$*, if it can be deduced from the following inductive rules.*

1. *The empty environment is valid, i.e.* $\models \emptyset$.
2. *If* $\models \Sigma_T$, $f\ \overline{x_i}^n @\ \overline{r_j}^m = e_f$, $\theta_f = [\overline{x_i \mapsto t_i}^n, \overline{r_j \mapsto \rho_j}^m, \mathit{self} \mapsto \rho^f_{\mathit{self}}]$*, and* $e_f, \Sigma_T \vdash \theta_f \leadsto t_f$ *hold, then* $\models \Sigma_T \uplus [f \mapsto \overline{t_i}^n \to \overline{\rho_j}^m \to t_f]$ *also holds.*

As we have said, region allocation/deallocation takes place at function call/return, so the absence of dangling pointers property should be apparent in the function application rule. The key idea is showing that the data structure returned by a function has no cells in the deallocated region. In this respect the most relevant properties are $P3$, $P4$ and $P5$.

In Fig. 9 we show the proof rules related to regions. Each one is a separate theorem interactively proved by Isabelle/HOL, which is kept in its database of proved theorems.

4.1 Proof obligations discharged by the certificate

For each expression e, the compiler generates a pair (θ, t). According to e's syntax, the certificate will perform the following actions:

$\boxed{c}$ The certificate checks that $t = B$ and uses the proof rule *LIT*.

$\boxed{x}$ The certificate checks that $t = \theta(x)$ and uses the proof rule *VAR1*.

$\boxed{x!}$ The certificate checks that $t = \theta(x)$ and uses the proof rule *VAR3*.

$\boxed{x@r}$ The certificate checks that $t = T\ \overline{t_i}^m @\ \rho_1 \ldots \rho_{l-1}\rho'$, $\theta(x) = T\ \overline{t_i}^m @\ \rho_1 \ldots \rho_l$, and $\theta(r) = \rho'$ and uses the proof rule *VAR2*.

$\boxed{\textbf{let } x_1 = e_1 \textbf{ in } e_2}$ The certificate has already proved $e_1, \varSigma_T \vdash \theta \leadsto t_1$, $e_2, \varSigma_T \vdash \theta_2 \leadsto t_2$, and receives (θ, t) for the **let** expression. It checks $\theta_2 = \theta \uplus [x_1 \mapsto t_1]$, $t_2 = t$, and uses the proof rule *LET*.

$\boxed{\textbf{let } x_1 = C\ \overline{a_i}^n \textbf{ in } e_2}$ The certificate has already proved $e_2, \varSigma_T \vdash \theta_2 \leadsto t_2$, and receives (θ, t', μ) for the **let** expression. It gets C's signature from $\varGamma$ and checks whether $\theta_2 = \theta \uplus [x_1 \mapsto \mu(t)]$, and $t_2 = t'$ hold. It also checks the rest of the premises of the proof rule LET_C and applies it.

$\boxed{\textbf{case } x \textbf{ of } \overline{C_i\ \overline{x_{ij}}^{n_i} \rightarrow e_i}^n}$ The certificate has already proved $e_i, \varSigma_T \vdash \theta_i \leadsto t_i$ for all the subsidiary expressions, and receives (θ, t', μ) for the whole **case** expression. It checks $\theta(x) = \mu(t)$, and for each $i \in \{1 \dots n\}$, $\mathit{wellT}(\overline{t_{ij}}^{n_i}, \rho, t)$, $\theta_i = \theta \uplus [\overline{x_{ij} \mapsto \mu(t_{ij})}^{n_i}]$, and $t_i = t'$. Then, it applies the proof rule *CASE*.

$\boxed{\textbf{case! } x \textbf{ of } \overline{C_i\ \overline{x_{ij}}^{n_i} \rightarrow e_i}^n}$ The certificate does here the same checks as in a **case** expression, but it applies the *CASE*! proof rule instead.

$\boxed{g\ \overline{a_i}^n @\ \overline{r'_j}^m}$ We assume that the most general type of the called function g is kept in the incremental global environment $\varSigma_T$. The certificate receives the (θ, t, μ) for this particular application, checks that $\varSigma_T$ has been proved valid, and gets g's signature from $\varSigma_T$. It then checks $t = \mu(t_g)$ and the rest of premises of the proof rule APP_{nonrec}, and applies it. The premise $\rho^g_{\mathit{self}} \notin \mathit{regions}\ (t_g)$, together with properties $P3$ and $P5$ of the judgement $g\ \overline{a_i}^n @\ \overline{r'_j}^m, \varSigma_T \vdash \theta \leadsto t$ obtained as the proof rule's conclusion, guarantee that the value resulting from the application has no cell living in the topmost region during g's body evaluation, corresponding to $\eta(\rho^g_{\mathit{self}})$. So, deallocating this region cannot cause dangling pointers. If f is recursive, the certificate checks the same premises as above except the validity of $\varSigma_T$. The *FUN* rule allows introducing in $\varSigma_T$ an assumption $\{f \mapsto t\}$ so that the internal applications to f could be checked. It is the responsibility of this rule to prove $\models \varSigma_T$.

$\boxed{f\ \overline{x_i}^n @\ \overline{r_j}^m = e_f}$ For each function definition as this one, the certificate builds the environment $\theta_f = [\overline{x_i \mapsto t_i}^n, \overline{r_j \mapsto \rho_j}^m, \mathit{self} \mapsto \rho_{\mathit{self}}]$ and proves

$$e_f, \varSigma_T \cup \{f \mapsto \overline{t_i}^n \rightarrow \overline{\rho_j}^m \rightarrow t_f\} \vdash \theta_f \leadsto t_f$$

The validity of $\varSigma_T$ should have been established previously to finding this definition. Then, it applies the *FUN* rule and establishes that $\varSigma_T \uplus \{f \mapsto \overline{t_i}^n \rightarrow \overline{\rho_j}^m \rightarrow t_f\}$ is a valid environment which can be used to certify subsequent functions.

4.2 Proof schemes of the proof rules

$\boxed{\textbf{\textit{LIT}}}$ The rule to be proved is $c, \varSigma_T \vdash \theta \leadsto B$.

Proof. We must prove $\forall E\ h\ k\ h'\ v\ \eta\ .\ P_1 \wedge P_2 \wedge P_3 \wedge P_4 \rightarrow P_5$. As c is a normal form, P_5 is of the form *consistent* (B, η, c, h), which is true by Def. 11. So, the predicate holds.

$\boxed{\textbf{\textit{VAR1}}}$ The rule to be proved is $x, \Sigma_T \vdash \theta \rightsquigarrow \theta(x)$.

Proof. We must prove $\forall E\ h\ k\ h'\ v\ \eta\ .\ P_1 \wedge P_2 \wedge P_3 \wedge P_4 \rightarrow P_5$. In this case, P_4 implies *consistent* $(\theta(x), \eta, E(x), h)$ which it is exactly P_5.

$\boxed{\textbf{\textit{VAR2}}}$ The rule to be proved is:

$$\frac{\theta(x) = T\ \overline{t_i}^m @\ \rho_1 \ldots \rho_l \quad \theta(r) = \rho'}{x@r, \Sigma_T \vdash \theta \rightsquigarrow T\ \overline{t_i}^m @\ \rho_1 \ldots \rho_{l-1}\rho'}$$

Proof. We must prove $\forall E\ h\ k\ h'\ v\ \eta\ .\ P_1 \wedge P_2 \wedge P_3 \wedge P_4 \rightarrow P_5$. The steps are the following:

1. By P_1 we know $E[x \mapsto p, r \mapsto j] \vdash h, k, x@r \Downarrow h', k, p'$, where $j \leq k$ and $(h', p') = copy(h, p, j)$.
2. The semantics of $copy(h, p, j)$ is creating in region j an exact copy of the recursive part of the DS[2] pointed to by p, while the non-recursive parts of p' are shared with the corresponding non-recursive parts of p. This implies that the constructors contained in corresponding cells are the same.
3. By P_4 we know *consistent* $(T\ \overline{t_i}^m @\ \overline{\rho_j}^l, \eta, p, h)$. We must show P_5, i.e. *consistent* $(T\ \overline{t_i}^m @\ \rho_1 \ldots \rho_{l-1}\rho', \eta, p', h')$.
4. We can prove consistency of the root cell pointed to by p' and of all its recursive descendants, by using the using a modified version μ' of the instantiation mapping μ used to guarantee *consistent* $(T\ \overline{t_i}^m @\ \overline{\rho_j}^l, \eta, p, h)$. This modified version μ' maps the most external region type of the constructor to ρ' instead of to ρ_l. Also, by *consistent* $(T\ \overline{t_i}^m @\ \overline{\rho_j}^l, \eta, p, h)$ we have $\rho' = \theta(r) \in dom(\eta)$ and $E(r) = j = \eta(\rho')$. The consistency of the non-recursive substructures is trivially guaranteed by *consistent* $(T\ \overline{t_i}^m @\ \overline{\rho_j}^l, \eta, p, h)$.

$\boxed{\textbf{\textit{VAR3}}}$ The rule to be proved is $x!, \Sigma_T \vdash \theta \rightsquigarrow \theta(x)$.

Proof. We must prove $\forall E\ h\ k\ h'\ v\ \eta\ .\ P_1 \wedge P_2 \wedge P_3 \wedge P_4 \rightarrow P_5$. The steps are the following:

1. By P_1 we have $E[x \mapsto p] \vdash h_1, k, x! \Downarrow h_2, k, q$, where $h_1 = h \uplus [p \mapsto w]$, $h_2 = h \uplus [q \mapsto w]$ and $fresh(q)$.
2. By P_4 we have *consistent* $(\theta(x), \eta, p, h_1)$.
3. We must prove *consistent* $(\theta(x), \eta, q, h_2)$. This follows because the cell w assigned to q in h_2 is a copy (in the same region) of the cell asigned to p in h_1. A non shown hypothesis on the coherence between the constructor typing environment Γ and the constructor table *RecPos* mentioned in Def. 1 is needed. Also, some auxiliary lemmas are needed about the absence of cyclic structures in the heap.

[2] Data Structure.

$\boxed{\textbf{\textit{LET}}}$ The rule to be proved is:

$$\frac{e_1, \Sigma_T \vdash \theta \leadsto t_1 \quad e_2, \Sigma_T \vdash \theta \uplus [x_1 \mapsto t_1] \leadsto t_2}{\textbf{let } x_1 = e_1 \textbf{ in } e_2 \vdash \theta \leadsto t_2}$$

Proof. We must prove $\forall E\ h\ k\ h'\ v\ \eta . P_1 \wedge P_2 \wedge P_3 \wedge P_4 \rightarrow P_5$ for the **let** expression. The steps are the following:

1. $P1$ for **let** leads to $P1$ for e_1. In effect, $E \vdash h, k, \textbf{let} \Downarrow h'', k, v_2$ leads to $E \vdash h, k, e_1 \Downarrow h', k, v_1$ and $E \uplus [x_1 \mapsto v_1] \vdash h', k, e_2 \Downarrow h'', k, v_2$ by the operational semantics. Let us call E_2 to $E \uplus [x_1 \mapsto v_1]$.
2. $P2$ for **let** leads to $P2$ for e_1 because E and θ are the same.
3. $P3$ for **let** and $P3$ for e_1 are the same property, i.e. $\mathit{admissible}\ (\eta, k)$.
4. $P4$ for **let** and $P4$ for e_1 are the same property, i.e. $\mathit{consistent}\ (\theta, \eta, E, h)$.
5. Then, by hypothesis, we can assume that $P5$ for e_1, i.e. $\mathit{consistent}(t, \eta, v_1, h')$, holds.
6. $P1$ for **let** leads to $P1$ for e_2, i.e. $E_2 \vdash h', k, e_2 \Downarrow h'', k, v_2$.
7. $P2$ for **let** leads to $P2$ for e_2 because $\mathit{dom}(E_2) = \mathit{dom}(E) \uplus \{x_1\}$ and $\mathit{dom}(\theta_2) = \mathit{dom}(\theta) \uplus \{x_1\}$.
8. $P3$ for **let** and $P3$ for e_2 are the same property, i.e. $\mathit{admissible}\ (\eta, k)$.
9. $P4$ for **let** and $P5$ for e_1 lead to $P4$ for e_2. We must show that

   $$\mathit{consistent}\ (\theta_2, \eta, E_2, h') = \mathit{consistent}\ (\theta, \eta, E, h') \wedge \mathit{consistent}\ (t_1, \eta, v_1, h')$$

   holds. The second part is $P5$ for e_1, and the first one comes from $P4$ for **let**, i.e. $\mathit{consistent}\ (\theta, \eta, E, h)$, and the property of the semantics that, during evaluation, a pointer either disappears from the heap domain, or it remains pointing to the same cell, but it is never updated.
10. Then, by hypothesis, we can assume that $P5$ for e_2, i.e. $\mathit{consistent}(t_2, \eta, v_2, h'')$, holds. This is the same property as $P5$ for **let**, so we are done.

$\boxed{\textbf{\textit{LET}}_{\boldsymbol{C}}}$ The rule to be proved is:

$$\frac{\begin{array}{lll} & \Gamma(C) = \overline{t_i}^n \rightarrow \rho \rightarrow t & \mathit{wellT}(\overline{t_i}^n, \rho, t) \\ t' = \mu(t) & e_2, \Sigma_T \vdash \theta \uplus [x_1 \mapsto t'] \leadsto t'' & \mathit{argP}(\overline{\mu(t_i)}^n, \mu(\rho), \overline{a_i}^n, r, \theta) \end{array}}{\textbf{let } x_1 = C\ \overline{a_i}^n\ @\ r \textbf{ in } e_2 \vdash \theta \leadsto t''}$$

Proof. We must prove $\forall E\ h\ k\ h'\ v\ \eta . P_1 \wedge P_2 \wedge P_3 \wedge P_4 \rightarrow P_5$ for the **let** expression. The steps are the following:

1. $P1$ for e_2 follows from $P1$ for **let**, since

   $$E \cup [x_1 \mapsto p] \vdash h \uplus \left[p \mapsto (E(r), C\ \overline{E(a_i)}^n)\right], k, e_2 \Downarrow h', k, v$$

   where p is a pointer not occurring in $\mathit{dom}(h)$.
2. $P2$ for e_2 holds, since $\mathit{dom}(\theta) \subseteq \mathit{dom}(E)$ implies $\mathit{dom}(\theta) \cup \{x_1\} \subseteq \mathit{dom}(E) \cup \{x_1\}$.

3. $P3$ holds for e_2 if it holds for **let**, since η and k are the same.
4. Let us denote the extended environments and heap by E^+, θ^+ and h^+. We must prove *consistent* $(\theta^+, \eta, E^+, h^+)$. We need the following lemma:

   Lemma 6. *For all h and h' such that $h \sqsubseteq h'$ it holds that:*

   $$\mathit{consistent}\ (t, \eta, p, h) \Rightarrow \mathit{consistent}\ (t, \eta, p, h')$$

   for all t, η and p provided closure $(p, h) \cap (\mathit{dom}(h') - \mathit{dom}(h)) = \emptyset$.

5. For every variable x distinct from x_1 and from $P4$ for **let** it follows $\mathit{consistent}(\theta(x), \eta, E(x), h)$, and by Lemma 6 it does $\mathit{consistent}(\theta(x), \eta, E(x), h^+)$.
6. We prove consistency for x_1, that is, *consistent* (t', η, p, h^+). Let us assume $\Gamma(C) = \overline{t_i}^n \to \rho \to T@\rho_m$. By using the *wellT* assumption we get $\rho = \rho_m$ and hence we can ensure that $\mu(\Gamma(C))$ has the form $\overline{t'_i}^n \to \rho' \to T@\rho'$ for some $\overline{t'_i}^n$ and ρ'. By the definition of *argP* we get $\theta(r) = \rho'$. Since $P4$ holds for **let** we get $\theta(r) \in \mathit{dom}(\eta)$ and hence $\rho' \in \mathit{dom}(\eta)$. Moreover, $\eta(\rho') = \eta(\theta(r)) = E(r)$, which corresponds exactly to the first component in $h(p)$. Finally we prove $\forall i \in \{1 \ldots n\}.\ \mathit{consistent}\ (t'_i, \eta, E(a_i), h^+)$ as follows:

   $$\begin{array}{ll} \quad \mathit{consistent}\ (\theta, \eta, E, h) & \{\text{by } P4 \text{ for } \textbf{let}\} \\ \Rightarrow \forall i \in \{1 \ldots n\}\ .\ \mathit{consistent}\ (\theta(a_i), \eta, E(a_i), h) & \\ \Rightarrow \forall i \in \{1 \ldots n\}\ .\ \mathit{consistent}\ (\theta(a_i), \eta, E(a_i), h^+) & \{\text{by Lemma 6}\} \\ \Rightarrow \forall i \in \{1 \ldots n\}\ .\ \mathit{consistent}\ (t'_i, \eta, E(a_i), h^+) & \{\text{by } \mathit{argP}\} \end{array}$$

7. By hypothesis we get $P5$ for e_2, i.e. *consistent* (t'', η, v, h'), which is exactly the same property $P5$ for **let**.

$\boxed{\textbf{\textit{CASE}}}$ The rule to be proved is:

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

Proof. We must prove $\forall E\ h\ k\ h'\ v\ \eta\ .\ P_1 \wedge P_2 \wedge P_3 \wedge P_4 \to P_5$ for the **case** expression. The steps are the following:

1. By $P1$ for **case** we get, for some $r \in \{1 \ldots n\}$, the corresponding $P1$ for e_r:

   $$E \cup [\overline{x_{rj} \mapsto v_j}^{n_r}] \vdash h, k, e_r \Downarrow h', k, v$$

   where $E(x) = p$ and $h(p) = (l, C_r\ \overline{v_j}^{n_r})$. In the following we shall show P2, P3 and P4 for the expression e_r.
2. $P2$ for e_r follows trivially from $\mathit{dom}(E) \subseteq \mathit{dom}(\theta)$.
3. $P3$ for e_r is the same judgement as $P3$ for the whole **case**, so it also holds.
4. We denote the extended environments by θ_r and E_r. We must check whether *consistent* (θ_r, η, E_r, h) holds. Let $z \in \mathit{dom}(\theta_r)$. If $z \in \mathit{dom}(\theta)$ consistency

follows trivially from $P4$ for **case**. If $z \notin dom(\theta)$ then $z = x_{rj}$ for some $j \in \{1 \ldots n_r\}$. In this case we prove *consistent* $(\mu(t_{rj}), \eta, v_j, h)$ as follows:

$$\begin{array}{lll}
& \mathit{consistent}\ (\theta, \eta, E, h) & \{P4 \text{ for } \mathbf{case}\} \\
\Rightarrow & \mathit{consistent}\ (\theta(x), \eta, E(x), h) & \\
\equiv & \mathit{consistent}\ (\mu(t), \eta, p, h) & \{\text{assumptions in rule Case}\} \\
\Rightarrow & \forall k \in \{1 \ldots n_r\}.\ \mathit{consistent}\ (\mu(t_{rk}), \eta, v_k, h) & \\
\Rightarrow & \mathit{consistent}\ (\mu(t_{rj}), \eta, v_j, h) &
\end{array}$$

5. By hypothesis we get $P5$ for e_r, which is the same as $P5$ for **case**.

$\boxed{\textbf{\textit{CASE!}}}$ The rule for **case**! is similar to that of **case**,

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

and its proof follows a similar pattern. The fact that a cell is removed before branching to the executed alternative makes things more difficult. In particular, some auxiliary lemmas are needed showing that creating dangling pointers does not destroy consistency (remember that consistency is a property telling where the cells live in the heap, so removing cells does not change the regions where the remaining cells live). Also a non-shown additional premise about the coherence between the type of constructors and the *RecPos* table is needed.

$\boxed{\textbf{\textit{APP}}_{\textbf{\textit{nonrec}}}}$ We assume that g's most general type is kept in the accumulated global environment Σ_T. We know that $\models \Sigma_T$ and $\Sigma_T(g) = \overline{t_i}^n \to \overline{\rho_j}^m \to t_g$ is equivalent to:

$$\begin{array}{ll}
g\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_g & \text{is } g\text{'s definition for some } x_i, r_j, e_g \\
\theta_g = [\overline{x_i \mapsto t_i}^n, \overline{r_j \mapsto \rho_j}^m, \mathit{self} \mapsto \rho^g_{\mathit{self}}] & \text{has been defined, and} \\
e_g, \Sigma_T \vdash \theta_g \rightsquigarrow t_g & \text{has been proved}
\end{array}$$

We need the following:

Lemma 7. *If μ maps range (θ) to some other region types, and polymorphic type variables of the types in θ to arbitrary types, and $e, \Sigma_T \vdash \theta \rightsquigarrow t$ holds, then $e \vdash \mu(\theta) \rightsquigarrow \mu(t)$ also holds.*

Proof. Let us assume that $e, \Sigma_T \vdash \theta \rightsquigarrow t$ holds, and some E, h, k, h_e, v_e, η satisfying $P1$ to $P4$. We choose η' such that $\eta' \cdot \mu = \eta$.

Now, let E', h' be an environment and a heap such that $dom(E') = dom(E)$ and *consistent* $(\mu(\theta), \eta', E', h')$ hold, and constructed in such a way that for all x_i, *closure* $(E(x_i), h)$ and *closure* $(E'(x_i), h')$ are identical except for the subvalues v in the first one satisfying *consistent* (t, η, v, h) with the type t being a polymorphic type variable, which have been replaced in the second one by arbitrary values v' satisfying *consistent* $(\mu(t), \eta', v', h')$. Formally, we require for all $x \in \mathit{dom}(E)$, $\mathit{instance}\ (\theta(x), \eta, E(x), h, \mu(\theta(x)), \eta', E'(x), h')$ to hold, where we define:

$$\begin{array}{l}
\mathit{instance}\ (B, \eta, c, h, B, \eta', c, h') = \mathit{true} \\
\mathit{instance}\ (a, \eta, v, h, \mu(a), \eta', v', h') = \mathit{consistent}\ (\mu(a), \eta', v', h') \\
\mathit{instance}\ (T\ \overline{t_i}^m @\ \overline{\rho_j}^l, \eta, p, h, \mu(T\ \overline{t_i}^m @\ \overline{\rho_j}^l), \eta', p', h') = \\
\quad \forall j, C, v_k, \mu'\ .\ h(p) = (j, C, \overline{v_k}^n) \wedge (\mu'(\Gamma(C)) = \overline{t_k}^n \to \rho_l \to T\ \overline{t_i}^m @\ \overline{\rho_j}^l) \to \exists v_k\ . \\
\quad (h'(p') = (j, C, \overline{v'_k}^n) \wedge \forall k \in \{1..n\}\ .\ \mathit{instance}\ (t_k, \eta, v_k, h, \mu(t_k), \eta', v'_k, h'))
\end{array}$$

It is easy to show by structural induction on e that it must exist v'_e, h'_e such that $E' \vdash h', k, e \Downarrow h'_e, k, v'_e$ and $\mathit{instance}\ (t, \eta, v_e, h_e, \mu(t), \eta', v'_e, h'_e)$ hold. So,

$$\mathit{consistent}\ (t, \eta, v_e, h_e) \quad \text{implies} \quad \mathit{consistent}\ (\mu(t), \eta', v'_e, h'_e)$$

Then $E', h', k, h'_e, v'_e, \eta'$ satisfy $P1$ to $P5$, and we are done. □

The rule to be proved now is:

$$\frac{\begin{array}{ll} \Sigma_T(g) = \overline{t_i}^n \to \overline{\rho_j}^m \to t_g & \rho^g_{\mathit{self}} \notin \mathit{regions}\ (t_g) \\ \mathit{argP}\ (\overline{\mu(t_i)}^n, \overline{\mu(\rho_j)}^m, \overline{a_i}^n, \overline{r'_j}^m, \theta) & t = \mu(t_g) \end{array}}{g\ \overline{a_i}^n @\ \overline{r'_j}^m, \Sigma_T \vdash \theta \leadsto t}$$

Proof. We must prove $\forall E\ h\ k\ h'\ v\ \eta\ .\ P_1 \wedge P_2 \wedge P_3 \wedge P_4 \to P_5$ for the application expression. The steps are the following:

1. $P1$ for the application, i.e. $E \vdash h, k, g\ \overline{a_i}^n @\ \overline{r'_j}^m \Downarrow h', k, v_g$ implies $E_g \vdash h, k+1, e_g \Downarrow h_g, k+1, v_g$, where

$$E_g = [\overline{x_i \mapsto E(a_i)}^n, \overline{r_j \mapsto E(r'_j)}^m, \mathit{self} \mapsto k+1] \text{ and } h' = h_g\mid_k$$

2. By hypothesis $\Sigma(g) = \overline{t_i}^n \to \overline{\rho_j}^m \to t_g$ we know $e_g, \Sigma_T \vdash \theta_g \leadsto t_g$. We define $\theta'_g = [\overline{x_i \mapsto \mu'(t_i)}^n, \overline{r_j \mapsto \mu'(\rho_j)}^n, \mathit{self} \mapsto \mu'(\rho^g_{\mathit{self}})]$, where μ' is the identity for ρ^g_{self} and identical to μ for the rest of region types. Then, $\theta'_g = \mu'(\theta_g)$.
3. By Lemma 7, we have $e_g \vdash \mu'(\theta_g) \leadsto \mu'(t_g)$.
4. Let us assume that E, h, k, h', v_g, η satisfy $P1$ and $P2$ in the conclusion. Obviously, $P1$ and $P2$ hold for $E_g, h, k+1, h_g, v_g$ in the premise. Let us define $\eta' = \eta \uplus \{\rho^g_{\mathit{self}} \mapsto k+1\}$. Then $P3$, i.e. $\mathit{admissible}\ (\eta', k+1)$, also holds.
5. By $\mathit{consistent}\ (\theta, \eta, E, h)$, and $\mathit{argP}\ (\overline{\mu(t_i)}^n, \overline{\mu(\rho_j)}^m, \overline{a_i}^n, \overline{r'_j}^m, \theta)$, and $\rho^g_{\mathit{self}} \notin \mathit{range}\ (\theta)$ we have $\mathit{consistent}\ (\mu'(\theta_g), \eta', E_g, h)$.
6. Then by $e_g \vdash \mu'(\theta_g) \leadsto \mu'(t_g)$, $P5$ holds, i.e. $\mathit{consistent}\ (\mu'(t_g), \eta', v_g, h_g)$.
7. By $\rho^g_{\mathit{self}} \notin \mathit{regions}\ (t_g)$, $\mathit{admissible}\ (\eta', k+1)$, and $\mathit{consistent}\ (\mu'(t_g), \eta', v_g, h_g)$ we have that $\mathit{closure}\ (v_g, h_g)$ does not have any cell in region $k+1$. So, $\mathit{consistent}\ (\mu(t_g), \eta, v_g, h')$ holds and we are done.

$\boxed{\textbf{\textit{FUN}}}$ This proof-rule inductively establishes the validity of the function typing environments. For a recursive function f a functional type t is valid if extending the current environment Σ_T with the assumption $\{f \mapsto t\}$ we can prove a concrete typing static assertion built from t for the f's body expression.

Proof. The rule to be proved is:

$$\frac{\begin{array}{ll} \models \Sigma_T & f\ \overline{x_i}^n @\ \overline{r_j}^m = e_f \\ \theta_f = [\overline{x_i \mapsto t_i}^n, \overline{r_j \mapsto \rho_j}^m, \mathit{self} \mapsto \rho^f_{\mathit{self}}] & e_f, \Sigma_T \cup \{f \mapsto \overline{t_i}^n \to \overline{\rho_j}^m \to t_f\} \vdash \theta_f \leadsto t_f \end{array}}{\models \Sigma_T \cup \{f \mapsto \overline{t_i}^n \to \overline{\rho_j}^m \to t_f\}}\ \mathit{FUN}$$

The proof follows the same pattern as that of the *FUN* rule of Sec. 3. The steps are the following:

1. Property $P_1(\Sigma_T)$ can be safely replaced by $P_1(f, n, \Sigma_T) \stackrel{\text{def}}{=} \exists n.E \vdash h, k, e \Downarrow_{f,n} h', k, v$.
2. A notion of satisfying a typing assertion up to depth n for function f is defined. It is denoted $e, \Sigma_T \vdash_{f,n} \theta \leadsto t$.
3. A notion of validity of a function typing environment up to depth n for function f is defined. It is denoted $\models_{f,n} \Sigma_T$.
4. The following lemma is proved by induction on n:

   Lemma 8. *If* $\models \Sigma_T$, $f\ \overline{x_i} @\ \overline{r_j} = e_f$, $\theta_f = [\overline{x_i \mapsto t_i}^n, \overline{r_j \mapsto \rho_j}^m, \mathit{self} \mapsto \rho^f_{\mathit{self}}]$, *and* $e_f, \Sigma_T \cup \{f \mapsto \overline{t_i}^n \to \overline{\rho_j}^m \to t_f\} \vdash \theta_f \leadsto t_f$ *hold, then*

   $$\forall n\ .\ \models_{f,n} \Sigma_T \uplus \{f \mapsto \overline{t_i}^n \to \overline{\rho_j}^m \to t_f\}$$

   also holds.

5. The following lemma completes the proof:

   Lemma 9. *If* $\forall n\,.\ \models_{f,n} \Sigma_T \uplus \{f \mapsto t\}$ *holds, then* $\models \Sigma \uplus \{f \mapsto t\}$ *also holds.*

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 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, 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.

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

Fig. 10. Isabelle/HOL definitions of *Core-Safe* expressions, free variables, and mark environments for `unshuffle`

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. 10 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. 11 we show (this time top-down for a better understanding) the components θ, t, and μ of the static assertions for the expressions of Fig. 10, proving the absence of dangling pointers for region deallocation. We show also the most general types of the constructors given by the global environment Γ.

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 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 proof rules 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

$$
\begin{array}{lll@{\qquad}lll}
\theta_{16} & \stackrel{\text{def}}{=} & [x_{34} : [a]@\rho_4,\ r_1 : \rho_1,\ r_2 : \rho_2,\ r_3 : \rho_3, \mathit{self} : \rho_{\mathit{self}}] & t_{16} & \stackrel{\text{def}}{=} & ([a]@\rho_1, [a]@\rho_2)@\rho_3 \\
\theta_{15} & \stackrel{\text{def}}{=} & \theta_{16} & 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] & t_{14} & \stackrel{\text{def}}{=} & ([a]@\rho_1, [a]@\rho_2)@\rho_3 \\
\theta_{13} & \stackrel{\text{def}}{=} & \theta_{14} + [x_{35} : [a]@\rho_2] & t_{13} & \stackrel{\text{def}}{=} & ([a]@\rho_1, [a]@\rho_2)@\rho_3 \\
\theta_{12} & \stackrel{\text{def}}{=} & \theta_{13} + [x_{37} : ([a]@\rho_1, [a]@\rho_2)@\rho_3] & t_{12} & \stackrel{\text{def}}{=} & ([a]@\rho_1, [a]@\rho_2)@\rho_3 \\
\theta_{11} & \stackrel{\text{def}}{=} & \theta_{16} + [x_{49} : a,\ x_{50} : [a]@\rho_4] & t_{11} & \stackrel{\text{def}}{=} & ([a]@\rho_1, [a]@\rho_2)@\rho_3 \\
\theta_{10} & \stackrel{\text{def}}{=} & \theta_{11} + [x_{40} : ([a]@\rho_2, [a]@\rho_1)@\rho_{\mathit{self}}] & t_{10} & \stackrel{\text{def}}{=} & ([a]@\rho_1, [a]@\rho_2)@\rho_3 \\
\theta_9 & \stackrel{\text{def}}{=} & \theta_{10} + [x_{15} : [a]@\rho_2] & t_9 & \stackrel{\text{def}}{=} & ([a]@\rho_1, [a]@\rho_2)@\rho_3 \\
\theta_8 & \stackrel{\text{def}}{=} & \theta_9 + [x_{16} : [a]@\rho_1] & t_8 & \stackrel{\text{def}}{=} & ([a]@\rho_1, [a]@\rho_2)@\rho_3 \\
\theta_7 & \stackrel{\text{def}}{=} & \theta_8 + [x_{38} : [a]@\rho_1] & t_7 & \stackrel{\text{def}}{=} & ([a]@\rho_1, [a]@\rho_2)@\rho_3 \\
\theta_6 & \stackrel{\text{def}}{=} & \theta_7 + [x_{39} : ([a]@\rho_1, [a]@\rho_2)@\rho_3] & t_6 & \stackrel{\text{def}}{=} & ([a]@\rho_1, [a]@\rho_2)@\rho_3 \\
\theta_5 & \stackrel{\text{def}}{=} & \theta_9 & t_5 & \stackrel{\text{def}}{=} & [a]@\rho_1 \\
\theta_4 & \stackrel{\text{def}}{=} & \theta_5 + [x_{47} : [a]@\rho_2,\ x_{48} : [a]@\rho_1] & t_4 & \stackrel{\text{def}}{=} & [a]@\rho_1 \\
\theta_3 & \stackrel{\text{def}}{=} & \theta_{10} & t_3 & \stackrel{\text{def}}{=} & [a]@\rho_2 \\
\theta_2 & \stackrel{\text{def}}{=} & \theta_3 + [x_{45} : [a]@\rho_2,\ x_{46} : [a]@\rho_1] & t_2 & \stackrel{\text{def}}{=} & [a]@\rho_2 \\
\theta_1 & \stackrel{\text{def}}{=} & \theta_{11} & t_1 & \stackrel{\text{def}}{=} & ([a]@\rho_2, [a]@\rho_1)@\rho_{\mathit{self}} \\
\\
\mu_1 & \stackrel{\text{def}}{=} & \{a \mapsto a,\ \rho_4 \mapsto \rho_4,\ \rho_1 \mapsto \rho_2,\ \rho_2 \mapsto \rho_1,\ \rho_3 \mapsto \rho_{\mathit{self}}\} & \Gamma((,)) & = & a_1 \to a_2 \to \rho_1 \to (a_1, a_2)@\rho_1 \\
\mu_3 & \stackrel{\text{def}}{=} & \{a_1 \mapsto [a]@\rho_2,\ a_2 \mapsto [a]@\rho_1,\ \rho_1 \mapsto \rho_{\mathit{self}}\} & \Gamma([\,]) & = & \rho_1 \to [a]@\rho_1 \\
\mu_7 & \stackrel{\text{def}}{=} & \{a_1 \mapsto [a]@\rho_1,\ a_2 \mapsto [a]@\rho_2,\ \rho_1 \mapsto \rho_3\} & \Gamma(:) & = & a \to [a]@\rho_1 \to \rho_1 \to [a]@\rho_1 \\
\mu_8 & \stackrel{\text{def}}{=} & \{a \mapsto a,\ \rho_1 \mapsto \rho_1\} & & & \mu_5 = \mu_3 \\
\mu_{14} & \stackrel{\text{def}}{=} & \{a \mapsto a,\ \rho_1 \mapsto \rho_2\} & & & \mu_{13} = \mu_7 \\
\mu_{16} & \stackrel{\text{def}}{=} & \{a \mapsto a,\ \rho_1 \mapsto \rho_4\} & & & \mu_{15} = \mu_8
\end{array}
$$

Fig. 11. Isabelle/HOL definitions of typing mappings, and types for `unshuffle`

is also an on-line 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.

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 Metha and Nipkow in [10], where they add a complete soundness proof. All these systems are aimed at conducting interactive proofs.

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.

They conjecture a set of proof rules and claim they could be used to prove the safety of destruction in the above mentioned [1] type system. But in fact they do not provide an Isabelle/HOL or a manual proof of them. Our experience in proving the proof rules of Sec. 3 is that it is a rather daunting task full of unexpected difficulties which have forced us to frequently modify the proof rules.

Appart from the proofs themselves, our contribution has been defining the appropriate functions, predicates and relations such as *closure, recReach, live, closed, consistent,...* relating the static information with the dynamic one, such that the proof rules could be proved correct. We are not aware of any other system automatically producing certificates on the absence of dangling pointers property.

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 1–16. 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.
17. R. Peña, C. Segura, and M. Montenegro. A Sharing Analysis for SAFE. In *Selected Papers Trends in Functional Programming, TFP'06*, pages 109–128. Intellect, 2007.