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

(Technical report SIC-5-08)

Manuel Montenegro Ricardo Peña Clara Segura
montenegro@fdi.ucm.es {ricardo,csegura}@sip.ucm.es

Universidad Complutense de Madrid, Spain

Abstract. We present a destruction-aware type system for the functional language *Safe*, which is a first-order eager language with facilities for programmer controlled destruction and copying of data structures. It provides also *regions*, i.e. disjoint parts of the heap, where the program allocates data structures. The runtime system does not need a garbage collector and all allocation/deallocation actions are done in constant time. This research is targeted to mobile code applications with limited resources in a Proof Carrying Code framework.
The type system guarantees that, in spite of sharing and of the use of implicit and explicit memory deallocation operations, well-typed programs will be free of dangling pointers at runtime. We also prove its correctness with respect to the operational semantics of the language.

1 Introduction

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

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

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

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

In our functional language *Safe*, we have chosen a semi-explicit approach to memory control in which programmers may cooperate with the memory management system by providing some information about the intended use of data structures (in what follows, abbreviated as DS). For instance, they may indicate that some particular DS will not be needed in the future and that it should be destroyed by the runtime system and its memory recovered. Programmers may also launch copies of a DS and control the degree of sharing between DSs. In order to use these facilities in safe way, we have developed a type system which guarantees that dangling pointers will never arise at runtime in the living heap.

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

The language is targeted to mobile code applications with limited resources in a Proof Carrying Code framework [Nec97,NL98]. The final aim is to endow programs with formal certificates proving the above properties. This aspect, as well as region size inference, are however beyond the scope of the current paper.

The *Safe* language and a sharing analysis for it were published in [PSM07a]. The use of regions in functional languages to avoid garbage collection is not new. Tofte and Talpin [TT97] introduced in ML-Kit —a variant of ML— the use of nested regions by means of a **letregion** construct. A lot of work has been done on this system [AFL95,BTV96,HMN01,TBE$^+$06]. Their main contribution is a *region inference* algorithm adding region annotations at the intermediate language level. Hughes and Pareto [HP99] incorporate regions in Embedded-ML. This language uses a sized-types system in which the programmer annotates heap and stack sizes and these annotations can be type-checked. So, regions can be proved to be bounded. A small difference with these approaches is that, in Safe, region allocation and deallocation are synchronized with function calls instead of being introduced by a special language construct. A more relevant difference is that Safe has an additional mechanism allowing the programmer to selectively destroy data structures inside a region. More recently, Hofmann and Jost [HJ03] have developed a type system to infer heap consumption. Theirs is also a first-order eager functional language with a construct $match'$ that destroys constructor cells. Its operational behaviour is similar to that of Safe **case!**. The main difference is that they lack a compile time analysis guaranteeing the safe use of this dangerous feature. Also, their language do not use regions. In [PSM07a] a more detailed comparison with all these works can be found.

Our safety type system has some characteristics of linear types (see [Wad90] as a basic reference). A number of variants of linear types have been developed

for years for coping with the related problems of achieving safe updates in place in functional languages [Ode92] or detecting program sites where values could be safely deallocated [Kob99]. The work closest to our system is [AH02], which proposes a type system for a language explicitly reusing heap cells. They prove that well-typed programs can be safely translated into an imperative language with an explicit deallocation/reusing mechanism. We summarise here the differences and similarities with our work.

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

1. Their uses 2 and 3 (read-only and shared, or just read-only) could be roughly assimilated to our use s (read-only), and their use 1 (destructive), to our use d (condemned), both defined in Section 4. We add a third use r (in-danger) arising from a sharing analysis based on abstract interpretation [PSM07a]. This use allows us to know more precisely which variables are in danger when some other one is destroyed.
2. Their uses form a total order $1 < 2 < 3$. A type assumption can always be worsened without destroying the well-typedness. Our marks s, r, d do not form a total order. Only in some expressions (**case** and $x@r$) we allow the partial order $s \leq r$ and $s \leq d$. It is not clear whether that order gives or not more power to the system. In principle it will allow diferent uses of a variable in different branches of a conditional being the use of the whole conditional the worst one. For the moment our system does not allow this.
3. Their system forbids non-linear applications such as $f(x, x)$. We allow them for s-type arguments.
4. Our typing rules for **let** $x_1 = e_1$ **in** e_2 allow more use combinations than theirs. Let $i \in \{1, 2, 3\}$ the use assigned to x_1, j the use of a variable z in e_1, and k the use of the variable z in e_2. We allow the following combinations (i, j, k) that they forbid: $(1, 2, 2)$, $(1, 2, 3)$, $(2, 2, 2)$, $(2, 2, 3)$. The deep reason is our more precise sharing information and the new in-danger type.
5. They need explicit declaration of uses while we infer them [PSM07b].

The plan of the paper is as follows; In Section 2 we informally introduce and motivate the language features. Section 3 formally defines its operational semantics. The kernel of the paper are sections 4 and 5 where respectively the destruction-aware type system is presented and proved correct. By lack of space, the detailed proofs are included in a separate appendix. Finally, Section 6 shows examples of successful type derivations and Section 7 concludes.

2 Summary of *Safe*

Safe is a first-order polymorphic functional language similar to (first-order) Haskell or ML with some facilities to manage memory. The memory model is

based in heap regions where data structures are built. However, in *Full-Safe* in which programs are written, regions are implicit. These are inferred when *Full-Safe* is desugared into *Core-Safe*, where they are explicit. As all the analyses mentioned in this paper happen at *Core-Safe* level, later in this section we will describe it in detail.

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

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

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

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

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

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

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

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

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

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

$$
\begin{array}{lcll}
prog & \rightarrow & dec_1; \ldots; dec_n; e & \\
dec & \rightarrow & f\ \overline{x_i}^n\ @\ \overline{r_j}^l = e & \{\text{recursive, polymorphic function}\} \\
e & \rightarrow & a & \{\text{atom: literal } c \text{ or variable } x\} \\
 & | & x@r & \{\text{copy}\} \\
 & | & x! & \{\text{reuse}\} \\
 & | & f\ \overline{a_i}^n\ @\ \overline{r_j}^l & \{\text{function application}\} \\
 & | & \textbf{let}\ x_1 = be\ \textbf{in}\ e & \{\text{non-recursive, monomorphic}\} \\
 & | & \textbf{case}\ x\ \textbf{of}\ \overline{alt_i}^n & \{\text{read-only case}\} \\
 & | & \textbf{case!}\ x\ \textbf{of}\ \overline{alt_i}^n & \{\text{destructive case}\} \\
alt & \rightarrow & C\ \overline{x_i}^n \rightarrow e & \\
be & \rightarrow & C\ \overline{a_i}^n\ @\ r & \{\text{constructor application}\} \\
 & | & e &
\end{array}
$$

Fig. 1. *Core-Safe* language definition

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

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

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

Polymorphic algebraic data types definitions are defined separately through **data** declarations. Algebraic types declarations have additional parameters indicating the regions where the constructed values of that type are allocated. For example, trees are represented as follows:

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

There may be several region parameters when nested types are used: different components of the data structure may live in different regions. In that case the

last region variable is the *outermost region* where the constructed values of this type are allocated. In the following example

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

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

The **data** declarations must be well-formed: Every type or region variable appearing in the left hand side must appear somewhere in the right hand side and the other way around. Also, the recursive occurrences must be identical to the left-hand side (polymorphic recursion is not allowed).

Function `splitD` shows an example with several output regions. In order to save space we show here a semi-desugared version with explicit regions:

```
splitD :: Int -> [a]!@rh2 -> rh1 -> rh2 -> rh3 -> ([a]@rh1,[a]@rh2)@rh3
splitD 0 zs!     @ r1 r2 r3 = ([]@r1, zs!)@r3
splitD n []!     @ r1 r2 r3 = ([]@r1, []@r2)@r3
splitD n (y:ys)! @ r1 r2 r3 = ((y:ys1)@r1, ys2)@r3
      where (ys1, ys2) = splitD (n-1) ys @r1 r2 r3
```

Notice that the tuple and its components may live in different regions.

3 Operational Semantics

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

A judgement of the form $E \vdash h, k, e \Downarrow h', k', v$ means that expression e is successfully reduced to normal form v under runtime environment E and heap h with $k+1$ regions, ranging from 0 to k, and that a final heap h' with $k'+1$ regions is produced as a side effect. Runtime environments E map program variables to values and region variables to actual region identifiers. We adopt the convention that for all E, if c is a constant, $E(c) = c$.

A heap h is a finite mapping from fresh variables p (we call them heap pointers) to construction cells w of the form $(j, C\,\overline{v_i}^n)$, meaning that the cell resides in region j. Actual region identifiers j are just natural numbers. Formal regions appearing in a function body are either region variables r corresponding to formal arguments or the constant *self*. By $h[p \mapsto w]$ we denote a heap h where the binding $[p \mapsto w]$ is highlighted. On the contrary, by $h \uplus [p \mapsto w]$ we denote the disjoint union of heap h with the binding $[p \mapsto w]$. By $h\,|_k$ we denote the heap obtained by deleting from h those bindings living in regions greater than k.

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

Rules *Lit* and Var_1 just say that basic values and heap pointers are normal forms. Rule Var_2 executes a copy expression copying the DS pointed to by p

$$E \vdash h, k, c \Downarrow h, k, c \qquad [Lit]$$

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

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

$$\frac{fresh(q)}{E[x \mapsto p] \vdash h \uplus [p \mapsto w], k, x! \Downarrow h \uplus [q \mapsto w], k, q} \; [Var_3]$$

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

$$\frac{E \vdash h, k, e_1 \Downarrow h', k', v_1 \quad E \cup [x_1 \mapsto v_1] \vdash h', k', e_2 \Downarrow h'', k'', v}{E \vdash h, k, \mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2 \Downarrow h'', k'', v} \; [Let_1]$$

$$\frac{j \leq k \quad fresh(p) \quad E \cup [x_1 \mapsto p] \vdash h \uplus [p \mapsto (j, C\ \overline{v_i}^n)], k, e_2 \Downarrow h', k', v}{E[r \mapsto j, \overline{a_i \mapsto v_i}^n] \vdash h, k, \mathbf{let}\ x_1 = C\ \overline{a_i}^n @ r\ \mathbf{in}\ e_2 \Downarrow h', k', v} \; [Let_2]$$

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

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

Fig. 2. Operational semantics of *Safe* expressions

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. We foresee that 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.

Rules Let_1, Let_2, and *Case* are the usual ones for an eager language, while rule *Case*! expresses what happens in a destructive pattern matching: the binding of the discriminant variable disappears from the heap. This action is the last source of possible dangling pointers.

In the following, we will feel free to write the derivable judgements as $E \vdash h, k, e \Downarrow h', k, v$ because of the following:

Proposition 1. *If $E \vdash h, k, e \Downarrow h', k', v$ is derivable, then $k = k'$.*
Proof: Straightforward, by induction on the depth of the derivation. □

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

Fig. 3. Type expressions

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

4 *Safe* Type System

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

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

We will write $T@\overline{\rho}^m$ instead of $T\ \overline{s}@\overline{\rho}^m$ to abbreviate whenever the $\overline{s}$ are not relevant. We shall even use $T@\rho$ to highlight only the outermost region. A partial order between types is defined: $\tau \geq \tau$, $T!@\overline{\rho}^m \geq T@\overline{\rho}^m$, and $T\#@\overline{\rho}^m \geq T@\overline{\rho}^m$. This partial order is extended below to type environments in the context of the expression being typed.

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

Constructor types have one region argument ρ which coincides with the outermost region variable of the resulting algebraic type $T\ \overline{s}@\overline{\rho}^m$. As recursive

sharing of DSs may happen only inside the same region, the constructors are given types indicating that the recursive substructure and the structure itself must live in the same region. For example, in the case of lists and trees:

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

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

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

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

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

In the type environments, Γ, we can find region type assignments $r : \rho$, variable type assignments $x : t$, and polymorphic scheme assignments to functions $f : \sigma$. In the rules we will also use $\mathit{gen}(\mathit{tf}, \Gamma)$ and $\mathit{tf} \preceq \sigma$ to respectively denote (standard) generalization of a monomorphic type and restricted instantiation of a polymorphic type. The instantiation of polymorphic type variables must not generate illegal types:

- Inside safe types, type variables may be instatiated only with safe types.
- Inside a condemned type, type variables may be instatiated with safe or condemned types.
- In-danger types are forbidden in an instantiation.

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

We now explain in detail the typing rules. In Fig. 5 we present the rule [FUNB] for function definitions. Function definitions make the environment grow with their types. Notice that the only regions in scope are the region parameters $\overline{r}^l$ and *self*, which gets a fresh region type ρ_{self}. The latter cannot appear in the type of the result as *self* dies when the function returns its value ($\rho_{self} \notin \mathit{regions}(s)$). 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

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

Fig. 4. Operators on type environments

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

Fig. 5. Rule for function definitions

a recursive descendant of the DS starting at x. This set is computed by the abstract interpretation based sharing analysis defined in [PSM07a].

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 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. Notation $type(y)$ 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:

$$\Gamma_1 \geq_e \Gamma_2 \equiv dom(\Gamma_2) \subseteq dom(\Gamma_1)\ \wedge\ \forall x \in dom(\Gamma_2).\Gamma_1(x) \geq \Gamma_2(x)\ \wedge$$
$$\forall x \in dom(\Gamma_1).cmd?(\Gamma_1(x)) \rightarrow \forall z \in sharerec(x, e).z \in dom(\Gamma_1)\ \wedge\ unsafe?(\Gamma_1(z))$$

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

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

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

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

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

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

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

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

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

Fig. 6. Type rules for expressions

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

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

Rule [APP] deals with function application. The use of the operator $\oplus$ avoids a variable to be used in two or more different positions unless they are all read-only parameters. Otherwise undesired side-effects could happen. There is also a rule for functions returning basic types but we do not show it here. 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 read-only variables can be used to construct a DS.

Rule [CASE] allows its discriminant variable to be read-only, in-danger, or condemned as it only reads the variable. Relation *inh*, defined in Figure 7, de-

$inh(s', s', s)$.
$inh(t, s, r) \leftarrow utype?(t, s)$
$inh(r, s, d) \leftarrow utype?(s, d) \wedge utype?(r, s)$
$inh(t, s, d) \leftarrow \neg utype?(s, d) \wedge utype?(t, s)$

$inh!(d, s, d) \leftarrow utype?(s, d)$
$inh!(t, s, d) \leftarrow \neg utype?(s, d) \wedge utype?(t, s)$

Fig. 7. Definitions of inheritance compatibility

termines 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 read-only, so must be all the pattern variables; if it is in-danger, the pattern variables may have any type; if it is condemned, recursive pattern variables are in-danger while non-recursive ones may have any type.

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 may have any type.

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* $\forall y \in sharerec(x, e) - \{x\}.y \in dom(\Gamma) \wedge unsafe?(\Gamma(y))$.
Proof: By induction on the depth of the type derivation. □

5 Correctness of the Type System

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

5.1 Absence of Dangling Pointers due to Cell Destruction

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

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

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

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

$$\forall x, y \in scope(e).closure(E, x, h) \cap recReach(E, y, h) \neq \emptyset \rightarrow x \in sharerec(y, e) \quad (1)$$

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

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

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

Definition 1. *We say that the initial configuration (E, h, L, s) is good whenever:*

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

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

We claim that the property $closed(E, L, h)$ is invariant along the execution of any well-typed Safe program. This will prove that dangling pointers never arise at runtime.

Theorem 1. *Let e be a Core-Safe expression. Let us assume that (Γ, E, h, L, s) is good. Then, (s, v, h') is good, and all the intermediate configurations in the derivation tree of $\Downarrow$ are good.*
Proof: By induction on the depth of the $\Downarrow$ derivation. □

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

5.2 Correctness of Region Deallocation

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

Lemma 2. *Let e_0 be the main expression of a Core-Safe program and let us assume that $[self \mapsto 0] \vdash \emptyset, 0, e_0 \Downarrow h_f, 0, v_f$ can be derived. Then in every judgment $E \vdash h, k, e \Downarrow h', k, v$ belonging to this derivation it holds that:*

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

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

This lemma allows us to leave out the condition $j \leq k$ in rule $[Let_2]$ and $[Var_2]$ of Fig. 2. The rest of the correctness proof is to establish a correspondence between type region variables ρ and region numbers j. If a variable admits the algebraic type $T@\overline{\rho_i}^n$ and it is related by E to a pointer p, we have to find out which concrete region of the structure pointed to by p corresponds to every ρ_i. This correspondence is called *region instantiation* whose formal definition can be found in the Appendix A. Intuitively a region instantiation is a function which maps type region variables to dynamic regions (in fact, natural numbers). The union of region instantiations (denoted by $\cup$) is defined only if they bind common type region variables to the same region, that is, they do not contradict each other. Given a pointer and a type, the function *build* returns the corresponding region instantiation:

$$
\begin{array}{ll}
build(h, c, B) & = \emptyset \\
build(h, p, T\ \overline{t_i}^n@\overline{\rho_i}^m) & = \emptyset \qquad \text{if } p \notin dom(h) \\
build(h, p, T\ \overline{t_i}^n@\overline{\rho_i}^m) & = [\rho_m \to j] \cup \bigcup_{i=1}^{n_k} build(h, b_i, t_{ki}) \text{ if } p \in dom(h) \\
\multicolumn{2}{r}{\textbf{where } h(p) = (j, C_k\ \overline{v_i}^{n_k})} \\
\multicolumn{2}{r}{\overline{t_{ki}}^{n_k} \to \rho_m \to T\ \overline{t_i}^n@\overline{\rho_i}^m \trianglelefteq \Sigma(C_k)}
\end{array}
$$

If p is a dangling pointer, its corresponding *build* is well-defined. However, dangling pointers are never accessed by a program (Sec 5.1). Now we define a notion of *consistency* between the variables belonging to a variable environment E. Intuitively it means that the correspondences between region type variables and concrete regions of each element of $dom(E)$ do not contradict each other.

Definition 2. *Let E be a variable environment, h a heap and Γ a type environment. We say that E is consistent with h under type environment Γ iff:*

1. *For all non-region variables $x \in dom(E)$: $build(h, E(x), \Gamma(x))$ is well-defined.*
2. *The region instantiation $\theta_X = \bigcup_{z \in dom(E)} build(h, E(z), \Gamma(z))$ is well-defined.*
3. *If we define $\theta_R = \{[\Gamma(r) \to E(r)] \mid r$ is a region variable and $r \in dom(E)\}$ then θ_X and θ_R are consistent.*

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

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

Fig. 8. Desugared versions of *concatD*, *treesortD* and *treesort*

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

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

Fig. 9. Simplified typing derivation for *concatD*

The following theorem proves that consistency is preserved by evaluation.

Theorem 2. *Let us assume that* $E \vdash h, k, e \Downarrow h', k, v$ *and that* $\Gamma \vdash e : t$*. If* E *and* h *are consistent under* Γ *with witness* θ*, then* $build(h', v, t)$ *is well-defined and consistent with* θ*.*

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

So far we have set up a correspondence between the actual regions where a data structure resides and the corresponding region types assigned by the type system: if two variables have the same outer region ρ in their type, the cells bound to them at runtime will live in the same actual region. Since the type system (see rule [FUNB] in Fig. 5) enforces that the variable ρ_{self} does not occur in the type of the function result, then every data structure returned by the function call does not have cells in *self*. This implies that the deallocation of the $(k+1)$-th region (which always is bound to *self*, as Lemma 2 states) at the end of a function call does not generate dangling pointers.

6 Examples

Now we shall consider the *concatD*, *treesort* and *treesortD* functions defined in Sec. 2. The desugared versions of their definitions are shown in Fig. 8. The first column is the result of the region inference phase, which inserts the $@r$ annotations into the code. Temporary structures are assigned the working region *self*. The second column shows the translation to *Core-Safe*.

Function *concatD* has type $[a]!@\rho_1 \rightarrow [a]@\rho \rightarrow \rho \rightarrow [a]@\rho$. Rule [FUNB] establishes that its body must be typed with *zs* being condemned and *ys* being

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

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

Fig. 10. Simplified typing derivation for *treesortD*

safe. The typing derivation is shown in Fig. 9. The typing rule [CASE!] is applied in (1). The branch guarded by [] can be typed by means of the [VAR] and [EXTS] rules (2). With respect to the second branch, the definition of *inh*! specifies that *xs* must have a condemned type in Γ, since it is a recursive child of *zs* (i.e. has the same underlying type). In (3) the rule [LET1] can be applied, as x_1 is not used destructively in the main expression of the **let** binding. We have $\Gamma_2 = \Gamma_3 \rhd^{\{x,x_1\}} \Gamma_4$, which is well-defined since the unsafe variables in $dom(\Gamma_2)$ (i.e. *xs* and *zs*) do not occur free in the expression $(x : x_1)@r$. The bound expression of **let** $x_1 = \dots$ is typed via the [APP] rule (4) and in its main expression the rule [CONS] is applied (5).

For the definition of *treesortD* (Fig. 10) we assume that *mkTreeD* and *inorder* have been already typed, obtaining σ_1 and σ_2, respectively. The rule [LET1] is applied in (1) since x_1 is not destroyed in the call to *inorder*. In addition, variable *xs* does not occur free there, so the environment $\Gamma = \Gamma_1 \rhd^{\emptyset} \Gamma_2$ is well-defined. In (2) the rule [APP] is applied, while in (3) first we apply [EXTS] in order to exclude the binding $[\mathit{treesortD} : \sigma]$ of Γ_2 and then [APP]. With respect to *treesort*, we get the following type scheme: $\forall \rho_1, \rho.[\mathit{Int}]@\rho_1 \to \rho \to [\mathit{Int}]@\rho$. To type its body, rule [LET2] is now applied, since xs' is destroyed in the *treesortD* call.

7 Conclusions and Future Work

We have presented a destruction-aware type system for a functional language with regions and explicit destruction and proved it correct, in the sense that the live heap will never contain dangling pointers. The compiler's front-end, including all the analyses mentioned in this paper —region inference, sharing analysis, and safe types inference— is fully implemented[2] and, by using it, we have successfully typed a significant number of small examples. We are currently working on the space consumption analysis. Preliminary work on a previously needed termination analysis has been reported in [LP07].

We are also working in the code generation and certification phases, trying to express the correctness proofs of our analyses as certificates which could be mechanically proof-checked by the proof assistant Isabelle [NPW02]. Longer term work include the extension of the language and of the analyses to higher-order.

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

References

[AFL95] A. Aiken, M. Fähndrich, and R. Levien. Better static memory management: improving region-based analysis of higher-order languages. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation, PLDI'95*, pages 174–185. ACM Press, 1995.

[AH02] D. Aspinall and M. Hofmann. Another Type System for in-place Updating. In *ESOP'02, LNCS 2305*, pages 36–52. Springer-Verlag, 2002.

[BTV96] L. Birkedal, M. Tofte, and M. Vejlstrup. From region inference to von neumann machines via region representation inference. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT*, pages 171–183, 1996.

[HJ03] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 185–197. ACM Press, 2003.

[HMN01] F. Henglein, H. Makholm, and H. Niss. A direct approach to control-flow sensitive region-based memory management. In *Proceedings of the 3rd ACM SIGPLAN international conference on Principles and Practice of Declarative Programming, PPDP'01*, pages 175–186. ACM Press, 2001.

[HP99] R. J. M. Hughes and L. Pareto. Recursion and Dynamic Data-Structures in Bounded Space; Towards Embedded ML Programming. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming, ICFP'99*, ACM Sigplan Notices, pages 70–81, Paris, France, September 1999. ACM Press.

[Kob99] N. Kobayashi. Quasi-linear Types. In *POPL'99*, pages 29–42. ACM, 1999.

[LP07] S. Lucas and R. Peña. Termination and Complexity Bounds for SAFE Programs. In *Proceedings of the 19th International Symposium on Implementation and Application of Functional Languages, IFL'07, Freiburg, Sept. 2007*, pages 8–23, 2007.

[Nec97] G. C. Necula. Proof-Carrying Code. In *Conference Record of POPL'97: The 24TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119. ACM SIGACT and SIGPLAN, ACM Press, 1997.

[NL98] G. C. Necula and P. Lee. The Design and Implementation of a Certifying Compiler. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, pages 333–344, 1998.

[NPW02] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. Number 2283 in LNCS. Springer, 2002.

[Ode92] M. Odersky. Observers for Linear Types. In *ESOP'92, LNCS 582*, pages 390–407. Springer-Verlag, 1992.

[PSM07a] R. Peña, C. Segura, and M. Montenegro. A Sharing Analysis for SAFE. In *Trends in Functional Programming (Volume 7) Selected Papers of the Seventh Symposium on Trends in Functional Programming, TFP'06.*, pages 109–128. Intellect, 2007.

[PSM07b] R. Peña, C. Segura, and M. Montenegro. An Inference Algorithm for Guaranteeing Safe Destruction. In *Proceedings of the 8th Symposium on Trends in Functional Programming, TFP'07. New York, April 2007*, pages XIV–1–16, 2007.

[TBE+06] M. Tofte, L. Birkedal, M. Elsman, N. Hallenberg, T. H. Olesen, and P. Sestoft. Programming with regions in the MLKit (revised for version 4.3.0). Technical report, IT University of Copenhagen, Denmark, 2006.

[TT97] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[Wad90] P. Wadler. Linear types can change the world! In *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 561–581. North Holland, 1990.

A Appendix: Detailed proof of correctness

A.1 Properties of the type system

In Section 4 the following invariant of the type system was introduced: If an expression gets a type under an environment Γ and there is a variable z with condemned type in this environment, then all variables sharing a recursive descendant of z must occur also in Γ with an in-danger type. We shall now proceed with the proof of this invariant:

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

$$\forall y \in \mathit{sharerec}(z, e) - \{z\}.y \in \mathit{dom}(\Gamma) \wedge \mathit{unsafe}?(\Gamma(y)).$$

Proof. By induction on the typing derivation $\Gamma \vdash e : s$.

In rules [LIT] and [VAR] the lemma holds trivially, since there is no variable with d type in the environment. If the final typing rule used in the derivation is [REUSE], there is only a variable with a d type in the environment, but all variables belonging to the set $\mathit{sharerec}(z, x!) - \{z\}$ are also in Γ_R with an r type. In the rule [COPY], if there exists a variable y (including z) with a d type in Γ_1, then every variable belonging to $\mathit{sharerec}(y, x@r) - \{y\}$ occurs in Γ_1 with an unsafe type. This is forced by the definition of $\geq$.

For the case of [EXTS] rule, every variable with a d type occurs in Γ and the property holds by induction hypothesis. In rule [EXTD] the variable x has d type, but all variables in $\mathit{sharerec}(x, e) - \{x\}$ are included in Γ_R with r type. If there is another variable $z' \neq x$ belonging to the domain of Γ, then the property holds by induction hypothesis.

With expressions $e \equiv [\textbf{let } x_1 = e_1 \textbf{ in } e_2]$ (rules [LET1] and [LET2]) we have $\Gamma \equiv \Gamma_1 \rhd^{fv(e_2)} \Gamma_2$. Let $z \in \mathit{dom}(\Gamma)$ so that $\Gamma(z) = d$ holds. We proceed by cases:

- $\Gamma(z) = \Gamma_1(z)$
  Every variable in $\mathit{sharerec}(z, e_1) - \{z\}$ occurs with an unsafe type in Γ_1. Since it holds that $\mathit{scope}(e_1) = \mathit{scope}(e)$, then $\mathit{sharerec}(z, e_1) = \mathit{sharerec}(z, e)$. Furthermore, if y has an unsafe type in Γ_1, then it has an unsafe type in $\Gamma_1 \rhd^{fv(e_2)} \Gamma_2$, by the definition of the operator $\rhd^L$. Therefore $\mathit{sharerec}(z, e) - \{z\}$ has an unsafe type in Γ.
- $\Gamma(z) = \Gamma_2(z)$
  By induction hypothesis all variables belonging to $\mathit{sharerec}(z, e_2) - \{z\}$ occur in Γ_2 with an unsafe type. In this case we have $\mathit{scope}(e) = \mathit{scope}(e_2) - \{x_1\}$ and hence:

  $$\mathit{sharerec}(z, e) \subseteq \mathit{sharerec}(z, e_2)$$

  Therefore, $\mathit{sharerec}(z, e) - \{z\}$ occurs in Γ_2 with an unsafe type as well, and —by the definition of $\rhd^L$ operator—, it occurs in Γ.

For the case of function application (rule [APP]) we have $\Gamma \equiv \Gamma_R + \Gamma'$. If $z \in \mathit{dom}(\Gamma)$ and $\Gamma(z) = d$, it can be shown that $z \in \mathit{dom}(\Gamma')$, as Γ_R only contains variables with r type.

Since $z \in \mathit{dom}(\Gamma')$, we obtain $\Gamma'(z) = t_i$ for some i. In that case we have:

$$\mathit{sharerec}(z, e) - \{z\} \subseteq R$$

Each variable in $\mathit{sharerec}(z, e) - \{z\}$ occurs with an unsafe type in Γ_R and thus in Γ as well.

In expressions $C\ \overline{a_i}^n\ @r$ (rule [CONS]) the lemma holds trivially, since there is no variable in Γ with a d type.

For the rule [CASE] the lemma holds by the definition of $\geq$ operator, which ensures that $\mathit{sharerec}(z, e) - \{z\}$ occurs with unsafe type in Γ if $\Gamma(z) = d$.

With respect to **case!** x **of** ... expressions (rule [CASE!]), let $\Gamma = \Gamma_R \otimes \Gamma' + [x : T!@p]$. We have either $z \in \mathit{dom}(\Gamma')$ or $z = x$. In the former case the lemma holds by the induction hypothesis. In the latter case it holds due to the inclusion of Γ_R in the environment Γ. □

A.2 Absence of Dangling Pointers due to Cell Destruction

First, formal definitions of reachability and sharing are given. These were informally introduced in Section 5.1.

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

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

where $\mathit{recPos}(C)$ is the set of recursive argument positions of constructor C.

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

Definition 4.

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

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

As it has been explained, if we have $E \vdash h, k, e \Downarrow h', k, v$, and $\Gamma \vdash e : s$, and $L = \mathit{fv}(e)$, we will call *initial configuration* to the tuple (Γ, E, h, L, s). On the other hand, the tuple (s, v, h') including the final value and heap together with the static type s of the original expression (and of the final value, as well) is called the *final configuration*. Associated to each initial configuration we have the following sets:

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

In definition 1 we have established the conditions for an initial configuration (Γ, E, h, L, s) to be good:

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

Analogously, a final configuration (s, v, h') is *g*ood if $closed(v, h')$ holds. Now we shall prove the theorem that ensures the preservation during the evaluation of this notion of goodness. Previously, we need the following lemma expressing that safe pointers in the heap are preserved by evaluation:

Lemma 2. *Let (Γ, E, h, L, s) be an initial good configuration. Then, for all $x \in L$ such that $\Gamma[x] = s$ we have $closure(E, x, h) = closure(E, x, h')$.*

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

By inspection of the semantic rules of Fig. 2, the evaluation of any expression never changes a mapping $[v \mapsto C\ \overline{v_i}]$ in the heap. At most, it may create dangling pointers by deleting a cell, but this action is restricted to cells pointed to by condemned variables. Moreover, all unsafe pointers belong to the set R. As $S \cap R = \emptyset$ in a good configuration, pointers in the set S (and their associated cells) are always preserved during evaluation. □

Theorem 1. *Let e be a Core-Safe expression. Let us assume that $E \vdash h, k, e \Downarrow h', k, v$, and that (Γ, E, h, L, s) is good. Then, (s, v, h') is good, and all the intermediate configurations in the derivation tree of $\Downarrow$ are good.*

Proof. By induction on the depth of the $\Downarrow$ derivation. Let us proceed by cases on the last rule applied.

$\boxed{\boldsymbol{e \equiv \textbf{let } x_1 = e_1 \textbf{ in } e_2}}$ By hypothesis we know that (Γ, E, h, L, s) is good and $E \vdash h, k, e \Downarrow h', k, v$. Let $S,\ R$ be the two sets associated to the initial configuration. We distinguish two cases according to the rule used for typing e:

$\boxed{LET1}$

Then, there must exist Γ_1 and Γ_2 such that $\Gamma = \Gamma_1 \rhd^{L_2} \Gamma_2,\ \Gamma_1 \vdash e_1 : s_1$ and $\Gamma_2 + [x_1 : s_1] \vdash e_2 : s$, where $L_2 = fv(e_2)$. Let $L_1 = fv(e_1)$. In order to apply the induction hypothesis, we must show that $(\Gamma_1, E, h, L_1, s_1)$ is good:

The two sets associated to this configuration are as follows:

1. $S_1 = S_{1s} \cup S_{1r} \cup S_{1d}$, where:

$$
\begin{aligned}
S_{1s} &\stackrel{\text{def}}{=} \textstyle\bigcup_{x \in L_1 \wedge \Gamma_1[x] = s \wedge \Gamma[x] = s} \{closure(E, x, h)\}, \quad S_{1s} \subseteq S \\
S_{1r} &\stackrel{\text{def}}{=} \textstyle\bigcup_{x \in L_1 \wedge \Gamma_1[x] = s \wedge \Gamma[x] = r} \{closure(E, x, h)\}, \\
S_{1d} &\stackrel{\text{def}}{=} \textstyle\bigcup_{x \in L_1 \wedge \Gamma_1[x] = s \wedge \Gamma[x] = d} \{closure(E, x, h)\},
\end{aligned}
$$

2. $R_1 = \bigcup_{x \in L_1 \wedge \Gamma_1[x] = d} \{p \in live(E, L_1, h) \mid p \rightarrow_h^* recReach(E, x, h)\},\ R_1 \subseteq R$
   This inclusion is because $\rhd^{L_2}$ ensures that $\Gamma_1[x] = d$ implies $\Gamma[x] = d$.

As $L_1 \subseteq L$, we know $\mathit{live}(E, L_1, h) \subseteq \mathit{live}(E, L, h)$, so $\mathit{closed}(E, L, h)$ implies $\mathit{closed}(E, L_1, h)$. Also, $S \cap R = \emptyset$ implies $S_{1s} \cap R_1 = \emptyset$. We must show now $(S_{1r} \cup S_{1d}) \cap R_1 = \emptyset$. This follows from the fact $\Gamma_1 \vdash e_1 : s_1$. If that set were non-empty, there would exist $x, z \in L_1$ such that $\Gamma_1[z] = d, \Gamma_1[x] = s$, and $\mathit{recReach}(E, z, h) \cap \mathit{closure}(E, x, h) \neq \emptyset$. But then we would have $x \in \mathit{sharerec}(z, e_1)$ and, by the properties of Γ_1, we would also have $\mathit{unsafe}?(\Gamma_1(x))$, in contradiction with $\Gamma_1[x] = s$. Then, $(\Gamma_1, E, h, L_1, s_1)$ is good.

Now, by applying the induction hypothesis on the reduction $E \vdash h, k, e_1 \Downarrow h', k, v_1$, we have shown that (s_1, v_1, h') is good. Let us define $\Gamma_2' \stackrel{\text{def}}{=} \Gamma_2 + [x_1 : s_1]$ and $E' = E + [x_1 \mapsto v_1]$. We must show now that $(\Gamma_2', E', h', L_2, s)$ is good. The two sets associated to this configuration are as follows:

1. $S_2 = S_{2s} \cup S_{2x_1}$, where:

$$S_{2s} \stackrel{\text{def}}{=} \bigcup_{x \in L_2 \wedge \Gamma_2[x] = s} \{\mathit{closure}(E', x, h')\}, \quad S_2 \subseteq S.$$
$$S_{2x_1} \stackrel{\text{def}}{=} \mathit{closure}(v_1, h')$$

   The above inclusion is because $\triangleright^{L_2}$ ensures that $\Gamma_2[x] = s$ implies $\Gamma[x] = s$, and because Lemma 2 ensures that all values in $\mathit{closure}(\{E(x) \mid x \in L_2 \wedge \Gamma_2[x] = s\}, h)$ are still in h'.
2. $R_2 = \bigcup_{x \in L_2 \wedge \Gamma_2[x] = d} \{p \in \mathit{live}(E', L_2, h') \mid p \rightarrow_{h'}^* \mathit{recReach}(E, x, h')\}$, $R_2 \subseteq R$. This inclusion is because $\triangleright^{L_2}$ ensures that $\Gamma_2[x] = d$ implies $\Gamma[x] = d$ and $x \notin L_1 \vee \Gamma_1[x] = s$, 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 Lemma 2 ensures that are still in h'.

Then, $S_{2s} \cap R_2 = \emptyset$ trivially holds. Also $S_{2x_1} \cap R_2 = \emptyset$ holds. Otherwise there would exist $z \in L_2$ such that $\Gamma_2'[z] = d$ and $x_1 \in \mathit{sharerec}(z, e_2)$. By Lemma 1 we would have the contradiction $\mathit{unsafe}?(\Gamma_2'[x_1])$. Finally, since $\mathit{closed}(E, L, h)$ holds by hypothesis, and $\mathit{closed}(v_1, h')$ has already been shown, then $\mathit{closed}(\Gamma_2', E', L_2, h')$ also holds. Hence, $(\Gamma_2', E', h', L_2, s)$ is good, and by induction hypothesis we have that (s, v, h'') is good. Then, the conclusion of the theorem holds in this case.

$\boxed{LET2}$

In this case, there must exist Γ_1 and Γ_2 such that $\Gamma = \Gamma_1 \triangleright^{L_2} \Gamma_2$, $\Gamma_1 \vdash e_1 : s_1$ and $\Gamma_2 + [x_1 : d_1] \vdash e_2 : s$, where $L_2 = fv(e_2)$, and d_1 is the condemned version of type s_1. So, the first part of the proof is identical to that of rule $LET1$.

We can assume then that (s_1, v_1, h') is good, where $E \vdash h, k, e_1 \Downarrow h', k, v_1$. Let us define $\Gamma_2' \stackrel{\text{def}}{=} \Gamma_2 + [x_1 : d_1]$ and $E' = E + [x_1 \mapsto v_1]$. We must show now that $(\Gamma_2', E', h', L_2, s)$ is good. The two sets associated to this configuration are as follows:

1. $S_2 = \bigcup_{x \in L_2 \wedge \Gamma_2[x] = s} \{\mathit{closure}(E, x, h')\}, \quad S_2 \subseteq S$. 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 Lemma 2 ensures that are still in h'.
2. $R_2 = R_{2x_1} \cup R_{2d}$, 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 Lemma 2 ensures that 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 fact $\Gamma_2' \vdash e_2 : s$. If that set were non-empty, then there would exist $x \in L_2$ such that $\Gamma_2'[x] = s$ and $\mathit{closure}(E', x, h') \cap \mathit{recReach}(E', x_1, h') \neq \emptyset$. But then we would have $x \in \mathit{sharerec}(x_1, e_2)$ and by the properties of Γ_2' we would have $\mathit{unsafe?}(\Gamma_2'(x))$ in contradiction with $\Gamma_2'[x] = s$.

Also, since $\mathit{closed}(E, L, h)$ holds by hypothesis, and $\mathit{closed}(v_1, h')$ has already been shown, then $\mathit{closed}(\Gamma_2', E', L_2, h')$ also holds.

Then, $(\Gamma_2', E', h', L_2, s)$ is good. By applying the induction hypothesis, we conclude that (s, v, h'') is good, being $E' \vdash h', k, e_2 \Downarrow h'', k, v$. Then, the conclusion of the theorem also holds in this case.

$\boxed{\boldsymbol{e \equiv \textbf{let } x_1 = C\,\overline{a_i}^n @ r \textbf{ in } e_2}}$ By hypothesis we know that (Γ, E, h, L, s) is good and $E \vdash h, k, e \Downarrow h', k, v$. Let $S,\ R$ be the two sets associated to the initial configuration.

As $L_1 \subseteq \overline{a_i}^n \subseteq L$ and all the a_i have safe types, we immediately have $S_1 \subseteq S$, $R = \emptyset$, and $\mathit{closed}(E, L, h)$ implies $\mathit{closed}(E, L_1, h)$. So the configuration $(\Gamma_1, E, h, L_1, s_1)$ is trivially good. Here we cannot apply the induction hypothesis since $C\,\overline{a_i}^n @ r$ is not an expression, but a binding expression. By the $[\mathit{Let}_2]$ semantic rule, we have $E(\overline{a_i}^n) = \overline{v_i}^n$, $h' = h \uplus [p \mapsto (j, C\,\overline{v_i}^n)]$, $j \leq k$, $\mathit{fresh}(p)$, and $E' = E \cup [x_1 \mapsto p]$. So, $\mathit{closed}(p, h')$ and the configuration (s_1, p, h') is good.

The rest of the reasoning is identical to those done in $\boxed{\mathit{LET1}}$ or $\boxed{\mathit{LET2}}$, depending on the typing rule used for typing the **let** expression.

$\boxed{\boldsymbol{e \equiv \textbf{case! } x \textbf{ of } \overline{C_i \overline{x_{ij}} \rightarrow e_i}}}$ By hypothesis we know that $(\Gamma_0, E, h, L, t, s)$ is good, $E[x \mapsto p] \vdash h[p \mapsto (l, C_k\ \overline{v_j}^{n_k})], k', e \Downarrow h', k', v$, and $\Gamma_0 \vdash e : s$. Let $S,\ R$ be the two sets associated to the initial configuration.

By the rule $\mathit{CASE}!$ of the semantics, we know $E_k \vdash h_k, k', e_k \Downarrow h', k', v$, being $E_k = E + [\overline{x_{kj} \mapsto b_j}]$, $h_k = h - [p \mapsto C_k\ \overline{v_j}^{n_k}]$, and e_k the expression corresponding to the pattern $C_k\ \overline{x_{kj}}$. By the rule $\mathit{CASE}!$ of the type system, we know:

$$
\begin{array}{ll}
\Gamma_0 = (\Gamma_R \otimes \Gamma) + [x : d] & C_k : \overline{t_{kj}}^{n_k} \rightarrow \rho \rightarrow T@\rho \\
R_{sh} = \mathit{sharerec}(x, e) - \{x\} & \Gamma_R = [y : \mathit{danger}(\mathit{type}(y)) \mid y \in R_{sh}] \\
\Gamma_k = \Gamma + [x : r] + \overline{[x_{kj} : t_{kj}]} & \Gamma_k \vdash e_k : s \\
\forall j\ .\ \mathit{inh}!(t_{kj}, s_{kj}, d) & d = T!@\rho \quad r = T\#@\rho \\
\forall z \in R_{sh} \cup \{x\}. z \notin L_k & L_k = \mathit{fv}(e_k)
\end{array}
$$

In order to apply the induction hypothesis, we must show that the configuration $(\Gamma_k, E_k, h_k, L_k, s)$ is good. The two sets associated to this configuration are as follows:

1. $S_k = S_{ks} \cup S_x$ where:

$$
\begin{array}{ll}
S_{ks} = \bigcup_{z \in L_k \wedge \Gamma[z] = s} \{\mathit{closure}(E_k, z, h_k)\}, & S_{ks} \subseteq S \\
S_x = \bigcup_{x_{kj} \in L_k \wedge \Gamma_k[x_{kj}] = s} \{\mathit{closure}(E_k, x_{kj}, h_k)\} &
\end{array}
$$

2. $R_k = R_{kd} \cup R_x$ where

$$R_{kd} \stackrel{\text{def}}{=} \bigcup_{z \in L_k \wedge \Gamma[z]=d} \{recReach(E_k, z, h_k)\}, \quad R_{kd} \subseteq R$$
$$R_x \stackrel{\text{def}}{=} \bigcup_{x_{kj} \in L_k \wedge \Gamma_k[x_{kj}]=d} \{p \in live(E_k, L_k, h_k) \mid p \rightarrow^*_{h_k} recReach(E_k, x_{kj}, h_k)\}$$

   By predicate *inh*!, at least the x_{ij} with $j \in recPos(C_k)$ would be included in R_x. We know that $recReach(E_k, x_{kj}, h_k)$ of a recursive pattern x_{kj} is included in $recReach(\Gamma_0, E, x, h)$, but this is not true for the non-recursive patterns. So, in general $R_x \not\subseteq R$.

From the hypothesis and the above inclusions, it is obvious that $S_{ks} \cap R_{kd} = \emptyset$. We must prove that $S_x \cap R_k = \emptyset$ and $S_{ks} \cap R_x = \emptyset$. It this were not the case, we would have $y, z \in L_k$ such that $\Gamma_k[y] = s$, $\Gamma_k[z] = d$, and $closure(E_k, y, h_k) \cap recReach(E_k, z, h_k) \neq \emptyset$. Then, we would have $y \in sharerec(z, e_k)$ and, by the properties of Γ_k, we would have $unsafe?(\Gamma_k(y))$, in contradiction with $\Gamma_k[y] = s$.

We must also prove $closed(E_k, L_k, h_k)$. By hypothesis, $closed(E, L, h)$ holds. By definition of R, the cell that has been deleted from h can only be pointed to by variables z such that $closure(E, z, h) \cap R \neq \emptyset$. By the properties of $sharerec(x, e)$, all these variables belong to $R_{sh} \cup \{x\}$ and (due to the [CASE!] rule) cannot belong to L_k. Hence, $closed(E_k, L_k, h_k)$ holds.

Then, by applying the induction hypothesis, we conclude that (s, v, h') is good, being $E_k \vdash h_k, k', e_k \Downarrow h', k', v$. Then, the conclusion of the theorem holds.

$\boxed{\boldsymbol{e \equiv \textbf{case } x \textbf{ of } \overline{C_i \overline{x_{ij}} \rightarrow e_i}}}$ By hypothesis we know that (Γ, E, h, L, s) is good, $E[x \mapsto p] \vdash h[p \mapsto (l, C_k\ \overline{v_j}^{n_k})], k', e \Downarrow h', k', v$, and $\Gamma \vdash e : s$. Let S, R be the two sets associated to the initial configuration.

By the rule *CASE* of the semantics, we know $E_k \vdash h, k', e_k \Downarrow h', k', v$, being $E_k = E + [\overline{x_{kj} \mapsto v_j}]$, and e_k the expression corresponding to the pattern $C_k\ \overline{x_{kj}}$. By the rule *CASE* of the type system, we know:

$$\begin{array}{ll} \Gamma \vdash x : t, & C_k : \overline{s_{kj}^{n_k}} \rightarrow \rho \rightarrow T@\rho \\ \Gamma_k = \Gamma + [\overline{x_{kj} : t_{kj}}], & \Gamma_k \vdash e_k : s \\ \forall j \,.\, inh(t_{kj}, s_{kj}, t) & t = T@\rho \vee t = T!@\rho \vee t = T\#@\rho \end{array}$$

In order to apply the induction hypothesis, we must show that the configuration $(\Gamma_k, E_k, h, L_k, s)$ is good.

By $L_k \subseteq L \cup \{\overline{x_{kj}}\}$ and $E_k(x_{kj}) = v_j \in closure(E, x, h)$ we have that $closure(E_k, L_k, h) \subseteq closure(E, L, h)$ and therefore, if $closed(E, L, h)$ holds then $closed(E_k, L_k, h)$ holds as well. For the rest of properties we do a case distinction according to the mark of the **case** discriminant:

$\Gamma[x] = s$ In this case, the predicate *inh* guarantees that for all j we have $\Gamma_k[x_{kj}] = s$. It is easy to show that $S_k \subseteq S$ and $R_k \subseteq R$. The hypothesis immediately leads to $S_k \cap R_k = \emptyset$, and then the configuration is good.

$\Gamma[x] = r$ In this case, the predicate *inh* allows for all j $\Gamma_k[x_{kj}] = s, r$ or d. Let us assume that $\exists z, j \,.\, z \in L_k \wedge \Gamma_k[z] = d \wedge E_k(x_{kj}) \rightarrow^*_h recReach(E_k, z, h)$. Then, the type environment invariant guarantees that $\Gamma_k[x_{kj}] \neq s$ and we know also

that $\Gamma_k[x] = r$. So, these patterns do not contribute to S_k. But, $S_k \not\subseteq S$ and $R_k \not\subseteq R$ in general, as there may be patterns such that $\Gamma_k[x_{kj}] = s, d$. In this case, for all variables z such that $\Gamma_k[z] = s$ we have $closure(E_k, z, h) \cap R_k = \emptyset$, otherwise the mark assigned to z by Γ_k would have not been s. Then the configuration is good.

$\Gamma[x] = d$ In this case, the predicate *inh* ensures $\Gamma_k[x_{kj}] = r$ for the recursive positions j of C_k and allows $\Gamma_k[x_{kj}] = s, r$ or d for the non-recursive positions. Then, these patterns do not contribute to S_k. As before, $S_k \not\subseteq S$ and $R_k \not\subseteq R$ in general. The reasoning for $S_k \cap R_k = \emptyset$ is the same as above, and then the configuration is good.

So, by applying the induction hypothesis, we conclude that (s, v, h') is good, and the conclusion of the theorem holds.

$\boxed{\boldsymbol{e \equiv f\ \overline{a_i}@\overline{r_j}^m}}$ By hypothesis we know that (Γ, E, h, L, s) is good, $E \vdash h, e, k \Downarrow h', k, v$, and $\Gamma \vdash e : s$. Let $S,\ R$ be the two sets associated to the initial configuration.

By the semantic rule *APP* we know that $E_a \vdash h, k+1, e_f \Downarrow h', k+1, v$ where $\Sigma \vdash f\ \overline{x_i} = e_f$ and $E_a = [\overline{x_i \mapsto E(a_i)}] + [\overline{r_j \mapsto E(r'_j)}] + [self : k+1]$. By the typing rule [APP] we know:

$$\frac{\begin{array}{c}\overline{t_i}^n \to \overline{\rho}^l \to T\ @\overline{\rho}^m \trianglelefteq \sigma \qquad \Gamma_0 = [f : \sigma] + \bigoplus_{j=1}^{l}[r_j : \rho_j] + \bigoplus_{i=1}^{n}[a_i : t_i] \\ R = \bigcup_{i=1}^{n}\{sharerec(a_i, f\ \overline{a_i}^n @ \overline{r}^l) - \{a_i\} \mid cdm?(t_i)\} \qquad \Gamma_R = \{y : danger(type(y)) \mid y \in R\}\end{array}}{\Gamma_R + \Gamma_0 \vdash f\ \overline{a_i}^n @\ \overline{r_j}^m : T\ @\overline{\rho}^m}\ \text{[APP]}$$

and then $\Gamma = \Gamma_R + \Gamma_0$. We define $\Gamma_a = [\overline{x_i : t_i}] + [\overline{r_j : \rho_j}] + [self : \rho_{self}]$. As the only variables in scope in e_f are the x_i, then $\bigcup_{i=1}^{n}\{sharerec(x_i, e_f) - \{x_i\} \mid \Gamma_0[x_i] = d\} = \emptyset$, and it is clear that $\Gamma_a \vdash e_f : s$. Also, $L_a \stackrel{\text{def}}{=} fv(e_f)$ is a subset of $\{\overline{x_i}\}$, so $E_a(L_a) \subseteq E(L)$ and then $closure(E_a, L_a, h) \subseteq closure(E, L, h)$. We will show that the configuration (E_a, h, L_a, s) is good. Its clear that $closed(E, L, h)$ implies $closed(E_a, L_a, h)$.

Let $S_a,\ R_a$ be the two sets associated to this configuration. We must show now that $S_a \cap R_a = \emptyset$. The only difficulty is the mapping between the x_i and the a_i. Should we allow having two formal arguments x_i and x_j with $\Gamma_a[x_i] \neq \Gamma_a[x_j]$ mapped to the same actual argument, then the disjointness property between S_a and R_a would be lost. Fortunately, the condition $\bigoplus_{i=1}^{n}[a_i : t_i]$ guarantees that this could not happen. It also guarantees that it is not possible to have x_i and x_j with $\Gamma_a[x_i] = \Gamma_a[x_j] = d$ mapped to the same actual argument. Should we allow that, then there would be two free condemned variables in e_f pointing to the same heap location. The sharing analysis assumes that all function arguments are disjoint. This assumption has no harmful consequences for safe arguments but it does for condemned ones: it would invalidate the reasoning done in the expression **case**! when proving the closedness of the heap. There we assumed that all variables pointing to the deleted cell $E(x)$ were included in $sharerec(x, e)$. This would not be true should we allow having a condemned alias for x. In operational terms, if an actual argument were substituted for two formal condemned arguments of a function, the same cell could be attempted to be destroyed twice when executing the function body.

Given these conditions, the hypothesis directly implies the disjointness of the two sets, and then the configuration is good. By applying the induction hypothesis, we conclude that (s, v, h') is good, and the conclusion of the theorem holds.

$\boxed{\boldsymbol{e \equiv c \vee e \equiv x \vee e \equiv x! \vee e \equiv x@r}}$ By hypothesis we know that (Γ, E, h, L, s) is good, where $L = \emptyset$ or $L = \{x\}$. So, $closed(c, h)$ holds trivially and $closed(E(x), h)$ holds in the remaining three cases.

By the semantic rules $[Lit], [Var_1], [Var_2]$ and $[Var_3]$, we know that $E \vdash h, k, e \Downarrow h', k', v$, where v is respectively $c, E(x), q, p'$, being q, p' fresh pointers pointing either to $E(x)$ or to a copy of the data structure starting at $E(x)$. Also, $h = h'$ in the first two cases, $h' = h \uplus [p \mapsto w]$ in the third case and $(h', p') = copy(h, p, j)$ in the fourth one. So $closed(v, h')$ holds trivially in all cases. Then (s, v, h') is good, and the conclusion of the theorem holds. □

A.3 Absence of Dangling Pointers due to Region Deallocation

First we prove that the topmost region in each execution of a program is the working region and thus it is only referenced by *self*:

Lemma 2. *Let e_0 be the main expression of a Core-Safe program and let us assume that $[self \mapsto 0] \vdash \emptyset, 0, e_0 \Downarrow h_f, 0, v_f$ can be derived. Then in every judgement $E \vdash h, k, e \Downarrow h', k, v$ belonging to this derivation it holds that:*

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

Proof. Both properties hold trivially at the starting judgement and are propagated at each application of the semantic rules. This propagation can be proven by simple inspection of these rules. □

In Section 5.2 the notion of region instantiation has been informally explained. This can be formalized this way:

Definition 5. *A **region instantiation** θ is a function from type region variables to natural numbers (interpreted as regions). It can also be defined as a set of **bindings** $[\rho \to n]$, where no variable ρ occurs twice in the left-hand side of a binding unless it is bound to the same region number.*

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

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

$$(\theta \cup \theta')(\rho) = \begin{cases} \theta(\rho) & \text{if } \rho \in dom(\theta) \\ \theta'(\rho) & \text{otherwise} \end{cases}$$

Definition 6. *Given a heap h, a pointer p and a type t, the function* ***build*** *is defined as follows:*

$$\begin{array}{ll} build(h, c, B) & = \emptyset \\ build(h, p, T\ \overline{t_i}^n @ \overline{\rho_i}^m) & = \emptyset \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{if } p \notin dom(h) \\ build(h, p, T\ \overline{t_i}^n @ \overline{\rho_i}^m) & = [\rho_m \to j] \cup \bigcup_{i=1}^{n} build(h, b_i, t_{ki}) \quad \text{if } p \in dom(h) \end{array}$$

$$\textbf{\textit{where}}\ h(p) = (j, C_k\ \overline{b_i}^n)$$

$$\overline{t_{ki}}^{n_k} \to \rho_m \to T\ \overline{t_i}^n @ \overline{\rho_i}^m \trianglelefteq \Sigma(C_k)$$

The fact that the *build* is equal to $\emptyset$ allows us to remove some pointers from the heap without putting at risk the well-definedness of the remaining ones. Similarly, if we add fresh pointers to a heap, the result of *build* applied to the existing ones is preserved.

Definition 7. *A heap h' is said to* ***extend*** *a heap h (denoted as $h \leq h'$) if $dom(h) \subseteq dom(h')$ and $\forall p \in dom(h).\ h(p) = h'(p)$. Moreover, if no pointer in $dom(h') - dom(h)$ is reachable from any pointer in $dom(h)$, we say that h'* ***strictly extends*** *the heap h (denoted as $h < h'$).*

Lemma 3. *Let h and h' be two heaps. The following two properties hold for each pointer $p \in dom(h)$:*

1. *If $h \leq h'$, then $build(h', p, t)$ well-defined $\Rightarrow$ $build(h, p, t)$ well-defined.*
2. *If $h < h'$, then $build(h, p, t)$ well-defined $\Rightarrow$ $build(h', p, t)$ well-defined.*

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

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

Definition 8.

$$copy(h_0[p \mapsto (k, C\ \overline{v_i}^n)], p, j) = (h_n \cup [p' \mapsto (j, C\ \overline{v_i'}^n)], p')$$

$$\textbf{\textit{where}}\ fresh(p')$$

$$\forall i \in \{1..n\}.(h_i, v_i') = \begin{cases} (h_{i-1}, v_i) & \text{if } v_i = c \ \vee\ i \notin RecPos(C) \\ copy(h_{i-1}, v_i, j) & \text{otherwise} \end{cases}$$

Lemma 4. *If $\theta = build(h, p, T@\rho)$ is well-defined and $(h', p') = copy(h, j, p)$, then for all ρ' such that $[\rho' \to j]$ is consistent with θ, $build(h', p', T@\rho')$ is well-defined and consistent with θ.*

Proof. By induction on the size of the structure pointed to by p. Let us assume that $h(p) = (k, C\ \overline{v_i}^n)$ and that $\overline{t_i}^n \to \rho \to T@\rho \trianglelefteq \Sigma(C)$ and $\overline{t_i'}^n \to \rho' \to T@\rho' \trianglelefteq \Sigma(C)$. We have:

$$build(h, p, T@\rho) = [\rho \to k] \cup build(h, v_1, t_1) \cup \cdots \cup build(h, v_n, t_n)$$

Since each $build(h, v_i, t_i)$ is well-defined, by Lemma 3 (2) we prove that $\theta' = build(h_{i-1}, v_i, t_i)$ is also well-defined, where the h_i are those appearing in the

definition of *copy*. The set of its bindings is, in fact, a subset of the bindings in θ. We can apply the induction hypothesis in order to prove that $build(h_{i-1}, v'_i, t'_i)$ is well-defined and consistent with θ' and hence with θ. Moreover, by applying Lemma 3(2) we have that $build(h_n, v'_i, t_i)$ is also well-defined and consistent with θ. Therefore it follows that:

$$build(h_n, p', T@\rho') = [\rho' \to j] \cup build(h_n, v'_1, t_1) \cup \cdots \cup build(h_n, v'_n, t_n)$$

is well-defined and consistent with θ. □

Let E be a variable environment, h a heap and Γ a type environment such that $dom(E) \subseteq dom(\Gamma)$. Definition 2 specifies that E is consistent with h under environment Γ if the following conditions hold:

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

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

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

$$\theta_R = \{[\Gamma(r) \to E(r)] \mid r \text{ is a region variable and } r \in dom(E)\}$$

   Then θ_X and θ_R are consistent.

When these three conditions hold, the result of $\theta_X \cup \theta_R$ is called the **witness** of the consistency of E and h under Γ. We are particularly interested in the fact that this property remains valid as new pointers are created in the heap. The following theorem proves that consistency is preserved by evaluation.

Theorem 2. *Let us assume that $E \vdash h, k, e \Downarrow h', k, v$ and that $\Gamma \vdash e : t$. If E and h are consistent under Γ with witness θ, then $build(h', v, t)$ is well-defined and consistent with θ.*

Proof. By induction on the depth of the $\Downarrow$ derivation. We distinguish cases on the last rule applied.

$\boxed{\boldsymbol{e \equiv c}}$ Since $build(h, c, B) = \emptyset$, is trivially well-defined and consistent with θ.

$\boxed{\boldsymbol{e \equiv x}}$ Since $x \in dom(E)$, $build(h, E(x), \Gamma(x))$ is well-defined and consistent with θ and hence, $build(h, v, t)$ is also well-defined and consistent with θ.

$\boxed{\boldsymbol{e \equiv x@r}}$ We know that $build(h, p, \Gamma(x))$ is well-defined and that the region instantiation $[\Gamma(r) \to E(r)] = [\rho' \to j'] \subseteq \theta_R$ is consistent with θ. Hence, by applying Lemma 4 we get $build(h', v, t)$ well-defined and consistent with θ.

$\boxed{\boldsymbol{e \equiv x!}}$ Analogous to the case $e = x$, as the resulting structure is essentially the same as the one pointed to by p and it has the same type.

We assume that $build(h \uplus [p \mapsto (j, C\ \overline{v_i}^n)], E(x), \Gamma(x))$ is well-defined and consistent with θ. Since $h \leq h \uplus [p \mapsto C\ \overline{v_i}^n]$ we can use Lemma 3 (1) in order to have $build(h, E(x), \Gamma(x))$ well-defined and consistent with θ. We shall denote the resulting heap $h \uplus [q \mapsto (j, C\ \overline{v_i}^n)]$ by h'. By Lemma 3 (2) we have that $build(h', E(x), \Gamma(x))$ is also well-defined and consistent with θ. Moreover, using the definition of *build* we can obtain $build(h', p, \Gamma(x)) = build(h', q, \Gamma(x))$ and hence the lemma holds.

$\boxed{\boldsymbol{e \equiv f\ \overline{a_i}^n @ \overline{r_i}^m}}$ Let $E' = [\overline{x_i \mapsto E(a_i)}^n, \overline{r_i \mapsto E(r_i')}^m, self \mapsto k+1]$ and σ the type scheme corresponding to the function f. If $\overline{t_i}^n \to \overline{\rho_i}^m \to t$ is an instance of σ, then we can derive:

$$\Gamma' \vdash e_f : t \quad \textbf{where } \Gamma' = [\overline{x_i : t_i}^n, \overline{r_i : \rho_i}^m, self : \rho_{self}] \text{ and } \forall i \in \{1..n\}.\ \rho_{self} \neq \rho_i$$

In order to apply the induction hypothesis we have to show that E' and h' are consistent under Γ' with witness θ. Since for each $i \in \{1..n\}$ we have that $build(h, E'(x_i), \Gamma'(x_i)) = build(h, E(a_i), \Gamma(a_i))$ and the latter is well-defined, we can ensure that the first condition of consistency holds. It can also be seen that the θ_X and θ_R corresponding to E, h and Γ are equivalent to those corresponding to E', h and Γ'. Therefore the second and third conditions of consistency hold and hence, E' and h are consistent under Γ' with the same witness θ. We can apply the induction hypothesis in order to get the well-definedness of $build(h', v', t)$ (consistent with θ)and Lemma 3 (1) to get the well-definedness of $build(h'|_k, v', t)$ (consistent with θ as well).

$\boxed{\boldsymbol{e \equiv \textbf{let } x_1 = e_1 \textbf{ in } e_2}}$ From the fact that $\Gamma_1 \vdash e_1 : t_1$ (resp. $\Gamma_2 + [x : t_1] \vdash e_2 : t_2$) and by means of rules [EXTS] and [EXTD] we can infer $\Gamma \vdash e_1 : t_1$ (resp. $\Gamma + [x : t_1] \vdash e_2 : t_2$). Hence the induction hypothesis can be applied in order to have that $build(h', v, t_1)$ is well-defined and consistent with θ. This allows us to prove that $E[x_1 \to v]$ and h' are consistent under $\Gamma + [x : t_1]$ and therefore we can apply again the induction hypothesis so as to get $build(h'', v', t)$ well-defined and consistent with θ.

$\boxed{\boldsymbol{e \equiv \textbf{let } x_1 = C\ \overline{a_i}^n @ r \textbf{ in } e_2}}$ Let us assume that $\overline{t_i}^n \to \rho \to t' \trianglelefteq \Sigma(C)$ where $t' = T@\rho$. We define:

$$\begin{aligned} E' &= E \cup [x_1 \mapsto p] \\ h_p &= h \uplus [p \mapsto (j, C\ \overline{E(a_i)}^n)] \\ \Gamma' &= \Gamma + [x_1 : t] \end{aligned}$$

We know that $\forall x \in dom(E') - \{x_1\}.build(h, E(x), \Gamma(x))$ is well-defined and their corresponding θ's are pairwise consistent. Since $h < h_p$, we prove that the same applies to $build(h_p, E(x), \Gamma(x))$, by Lemma 3 (2). Now we shall show the well-definedness of $build(h_p, E(x_1), \Gamma(x_1)) = build(h_p, p, t')$.

$$build(h_p, p, t') = [\rho \to j] \cup build(h_p, E(a_1), t_1) \cup \cdots \cup build(h_p, E(a_n), t_n)$$

From the fact that all the $build(h_p, E(a_i), t_i)$ are pairwise consistent and they are consistent with $[\rho \to j]$ (since $[\rho \to j] = [\Gamma(r) \to E(r)] \in \theta_R$), then we prove that $build(h_p, p, t')$ is well-defined and also consistent with each $build(h_p, E(x), \Gamma(x))$, $x \in dom(E) - \{x_1\}$. Therefore E' and h_p are consistent under Γ', so the induction hypothesis can be applied in order to get $build(h', v, t)$ well-defined and consistent with θ.

$\boxed{\boldsymbol{e \equiv \textbf{case } x \textbf{ of } \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n}}}$ The last rule used is $[Case]$. Let us assume that $h(p) = (j, C_r\ {v_i}^{n_r})$ and that $\overline{t_{rj}}^{n_r} \to \rho \to T@\rho \trianglelefteq \Sigma(C_r)$. We define $E' = E \cup [\overline{x_{rj} \mapsto v_j}^{n_r}]$ and $\Gamma' = \Gamma + [x_{rj} : t_{rj}]$. By hypothesis we know that $build(h, E(x), \Gamma(x))$ is well-defined and equal to $build(h, p, T@\rho)$:

$$build(h, p, T@\rho) = [\rho \to j] \cup build(h, v_1, t_{r1}) \cup \cdots \cup build(h, v_{n_r}, t_{rn_r})$$

Since the whole $build(h, p, T@\rho)$ is well-defined, every component $build(h, v_j, t_{rj})$ is also well-defined and consistent with the whole $build$ and with the remaining $build$s coming from E. Furthermore, for every $j \in 1..n_r$, $build(h, E'(x_{rj}), \Gamma'(x_{rj}) = build(h, v_j, t_{rj})$. Hence E' and h are consistent under the type environment Γ'. Since we can obtain (via the [EXTS] and [EXTD] rules) that $\Gamma' \vdash e_r : t$, the induction hypothesis can be applied in order to get $build(h, v, t)$ well-defined and consistent with θ.

$\boxed{\boldsymbol{e \equiv \textbf{case! } x \textbf{ of } \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n}}}$ The reasoning is similar to that of the rule [CASE!]. The only difference is the fact that p now is a dangling pointer, but the Lemma 3 (1) allows us to preserve the consistence of E' and h under Γ', so we can still apply the induction hypothesis in order to get the desired result. $\square$