An Inference Algorithm for Guaranteeing Safe Destruction *

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

Universidad Complutense de Madrid, Spain

Abstract. *Safe* 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 programmer may allocate data structures. A type system is used to avoid dangling pointers arising from the inadequate usage of these facilities. In this paper we present an inference algorithm, we describe its implementation, and give a number of successfully typed examples. Also the correctness of the algorithm is proved.

1 Introduction

Many imperative languages offer low level mechanisms to allocate and free heap memory, which the programmer may use in order to dynamically create and destroy pointer based data structures. These mechanisms give the programmer complete control over memory usage but are very error prone. Well known problems that may arise when using a programmer-controlled memory management are dangling references, undesired sharing between data structures with complex side effects as a consequence, and polluting memory with garbage.

On the other hand, functional languages usually consider memory management as a low level issue. Allocation is done implicitly and usually a garbage collector takes care of the memory exhaustion situation.

A semi-explicit approach to memory control is the functional language called *Safe* [PS04], in which the programmer cooperates with the memory management system by providing some information about the intended use of data structures. For instance, the programmer may indicate that some particular data structure will not be needed in the future and that, as a consequence, it may be safely destroyed by the runtime system and its memory recovered. The language uses regions to locate data structures. It also allows controlling the degree of sharing between different data structures. A garbage collector is not needed. Allocation and destruction of data structures are done as execution proceeds.

More interesting is the definition of a type system [MPS08] guaranteeing that destruction facilities and region management can be done in a safe way. In particular, it guarantees that dangling pointers are never created in the live heap. In this paper we present an inference algorithm, we describe its implementation, and give some examples of use. We also prove the correctness of the algorithm with respect to the type system. In Section 2 we summarize language *Safe*. The type system is presented in Section 3 and the corresponding inference algorithm is explained in Section 4. In Section 5 we show some examples whose types have been successfully inferred. Finally, Section 6 compares this work with related analyses in other languages with memory management facilities.

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

2 Summary of *Safe*

Safe is a first-order polymorphic functional language similar to (first-order) Haskell or ML with some facilities to manage memory. The memory model is based on heap regions where data structures (DS) are built. However, in *Full-Safe* in which programs are written, regions are implicit. These are inferred when *Full-Safe* is desugared into *Core-Safe*. As all the analyses mentioned in this paper [PSM07,MPS08] 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 deallocate the cell corresponding to the outermost constructor. Using recursion the recursive spine of the whole data structure may be deallocated. We say that it is *condemned*. As an example, we show an append function destroying the first list's spine, while keeping its elements in order to build the result:

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

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

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

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

First, the original list `xs` is used to build a search tree by applying function `mkTreeD` (defined below). This tree is then traversed in inorder to produce the sorted list. The tree is not part of the result of the function, so it will be built in the working region and will die when the `treesortD` function returns (in *Core-Safe* this is explicit). 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 tree by calling function `insertD`, which is the destructive version of insertion in a binary search tree:

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

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

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

Fig. 1. *Core-Safe* language definition

in the example `lt` and `rt` are condemned and they must be reused in order to be part of the result.

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

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

Polymorphic algebraic data types are defined separately through **data** declarations. Region inference adds region arguments to constructors forcing the restriction that recursive substructures must live in the same region as its parent. 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` is an example with several output regions. In order to save space we show here a semi-desugared version with explicit regions. Notice that the resulting tuple and its components may live in different regions:

```
splitD :: Int -> [a]!@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
```

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

Fig. 2. Type expressions

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

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 $T\#@\overline{\rho}^m \geq T@\overline{\rho}^m$.

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 $T\ \overline{s}@\overline{\rho}^m$, and reflect that recursive sharing can happen only in the same region. As example:

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

We assume that the types of the constructors are collected in an environment Σ, easily built from the **data** type declarations. In functional types there may be several region arguments $\overline{\rho_j}^{\,l}$ where data structures may be built.

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

Fig. 3. Rule for function definitions

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 $\mathit{gen}(tf, \Gamma)$ and $tf \trianglelefteq \sigma$ to respectively denote (standard) generalization of a monomorphic type and restricted instantiation of a polymorphic type with safe types.

Several operators on environments are used in the rules. 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. 3 we present the rule [FUNB] for function definitions. Notice that the only regions in scope are the region parameters $\overline{r_j}^l$ and *self*, which gets a fresh region type ρ_{self}. The latter cannot appear in the type of the result as *self* dies when the function returns its value ($\rho_{self} \notin \mathit{regions}(s)$).

In Figure 4, the rules for typing expressions are shown. Function $\mathit{sharerec}(x, e)$ gives an upper approximation to the set of variables in scope in e which share a recursive descendant of the DS starting at x. This set is computed by the abstract interpretation based sharing analysis defined in [PSM07].

A key point to prove the correctness of the type system with respect to the semantics is an invariant of the type system 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 according to the invariant mentioned above. Notation $\mathit{type}(y)$ represents the Hindley-Milner type inferred for variable y[1].

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

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 destroyed in the main expression e_2 if desired. In both **let** rules operator $\rhd^L$ guarantees that: (1) Each variable y condemned or in-danger in e_1 may not be referenced in e_2 (i.e. $y \notin \mathit{fv}(e_2)$), as it could be a dangling reference. (2) Those variables marked as unsafe either in Γ_1 or in Γ_2 will keep those types in the combined environment.

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

Rule [APP] deals with function application. The use of the operator $\oplus$ avoids a variable to be used in two or more different positions unless they are all safe

[1] Inference implementation first infers H-M types and then destruction annotations

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

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

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

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

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

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

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

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

Fig. 4. Type rules for expressions

parameters. Otherwise undesired side-effects could happen. The set R collects all the variables sharing a recursive substructure of a condemned parameter, which are marked as in-danger in environment Γ_R. Rule [CONS] is more restrictive as only safe variables can be used to construct a DS.

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

In rule [CASE!] the discriminant is destroyed and consequently the text should not try to reference it in the alternatives. The same happens to those variables sharing a recursive substructure of x, as they may be corrupted. All those variables are added to the set R. Relation *inh*! determines the types inherited by pattern variables: recursive ones are condemned while non-recursive ones are safe. As recursive pattern variables inherit condemned types, the type environments for the alternatives contain all the variables sharing their recursive substructures as in-danger. In particular x may appear with an in-danger type. In order to type the whole expression we must change it to condemned.

4 Inference algorithm

The typing rules presented in Section 3 allow in principle several correct typings for a program. On the one hand, this is due to polymorphism and, on the other hand, to the fact that it may assign more condemned and in-danger types that those really needed. We are interested in *minimal* types in the sense of being as much polymorphic as possible and having as few unsafe types as possible.

As an example, let us consider the following definition: $f(x : xs)@r = xs@r$. The type system can give f type $[a]@\rho \to \rho' \to [a]@\rho'$ but also the type $[a]!@\rho \to \rho' \to [a]@\rho'$. Our inference algorithm will return the first one.

Also, we are not interested in having mandatory explicit type declarations. This is what the inference algorithm presented in this section achieves.

It has two different phases: a (modified) Hindley-Milner phase and an unsafety propagation phase. The first one is rather straightforward with the added complication of region inference, which is done at this stage. Its output consists of decorating each applied occurrence of a variable and each defining occurrence of a function symbol in the abstract syntax tree (AST) with its Hindley-Milner type. We will not insist further in this phase here.

The second phase propagates unsafety information from the parts of the text where condemned and in-danger types arise to the rest of the program text. As the Hindley-Milner types are already available, the only additional information needed for each variable is a *mark* telling whether it is a safe, in-danger or condemned one. Condemned and in-danger marks arise for instance in the [CASE!], [REUSE], and [APP] typing rules while mandatory safe marks arise for instance in rules for constructor applications. The algorithm generates minimal sets of these marks in the program sites where they are mandatory and propagates this information bottom-up in the AST looking for consistency of the marks. It may happen that a safe mark is inferred for a variable in a program site and a condemned mark is inferred for the same variable in another site. This sometimes is allowed by the type system —e.g. it is legal to read a variable in the auxiliary expression of a **let** and to destroy it in the main expression—, and disallowed some other times—e.g. in a **case**, it is not legal to have a safe type for a variable in one alternative and a condemned or in-danger type for it in another alternative.

So, the algorithm has two working modes. In the bottom-up working mode, it accumulates sets of marks for variables. In fact, it propagates bottom-up four sets of variables (D, R, S, N) respectively meaning condemned, in-danger, safe, and don't-know variables in the corresponding expression. The fourth set arises from the non-deterministic typing rules for [COPY] and [CASE] expressions.

The algorithm checks for consistency the information coming from two or more different branches of the AST. This happens for instance in **let** and **case** expressions. Even though the information is consistent it may be necessary to propagate some information down the AST. For instance, $x \in D_1$ and $x \in N_2$ is consistent in two different branches 1 and 2 of a **case** or a **case!**, but a D mark for x must be propagated down the branch 2.

So, the algorithm consists of a single bottom-up traversal of the AST, occassionally interrupted by top-down traversals when new information must be propagated in one or more branches. If the propagation does not raise an error, then the bottom-up phase is resumed.

In Figure 5 we show the rules that drive the bottom-up working mode. A judgement of the form $e \vdash_{inf} (D, R, S, N)$ should be read as: from expression e the 4-tuple (D, R, S, N) of marked variables is inferred. A straightforward

$$\frac{}{c \vdash_{inf} (\emptyset,\emptyset,\emptyset,\emptyset)}\ [\text{LIT}_I] \qquad \frac{}{x \vdash_{inf} (\emptyset,\emptyset,\{x\},\emptyset)}\ [\text{VAR}_I] \qquad \frac{}{x@r \vdash_{inf} (\emptyset,\emptyset,\emptyset,\{x\})}\ [\text{COPY}_I]$$

$$\frac{R = \mathit{sharerec}(x, x!) - \{x\} \quad \mathit{type}(x) = T@\,\rho}{x! \vdash_{inf} (\{x\}, R, \emptyset, \emptyset)}\ [\text{REUSE}_I] \qquad \frac{\forall i \in \{1..n\}.a_i \vdash_{inf} (\emptyset,\emptyset,S_i,\emptyset)}{C\ \overline{a_i}^n\ @r \vdash_{inf} (\emptyset,\emptyset,\bigcup_{i=1}^n S_i,\emptyset)}\ [\text{CONS}_I]$$

$$\frac{\begin{array}{lll}\forall i \in \{1..n\}.D_i = \{a_i \mid i \in I_D\} & (\bigcup_{i=1}^n D_i) \cap (\bigcup_{i=1}^n S_i) = \emptyset & R \cap (\bigcup_{i=1}^n S_i) = \emptyset \\ \forall i \in \{1..n\}.S_i = \{a_i \mid i \in I_S\} & (\bigcup_{i=1}^n D_i) \cap (\bigcup_{i=1}^n N_i) = \emptyset & R \cap (\bigcup_{i=1}^n D_i) = \emptyset \\ \forall i \in \{1..n\}.N_i = \{a_i \mid i \in I_N\} & \forall i,j \in \{1..n\}\,.\, i \neq j \Rightarrow D_i \cap D_j = \emptyset & R \cap (\bigcup_{i=1}^n N_i) = \emptyset \\ \Sigma \vdash f : (I_D,\emptyset,I_S,I_N) & R = \bigcup_{i=1}^n \{\mathit{sharerec}(a_i, f\ \overline{a_i}^n\ @\overline{r_j}^l) - \{a_i\} \mid a_i \in D_i\} & \end{array}}{f\ \overline{a_i}^n\ @\overline{r_j}^l \vdash_{inf} (\bigcup_{i=1}^n D_i, R, \bigcup_{i=1}^n S_i, (\bigcup_{i=1}^n N_i) - (\bigcup_{i=1}^n S_i))}\ [\text{APP}_I]$$

$$\frac{\begin{array}{ll} e_1 \vdash_{inf} (D_1,R_1,S_1,N_1) & x_1 \notin R_2 \qquad (D_1 \cup R_1) \cap \mathit{fv}(e_2) = \emptyset \\ e_2 \vdash_{inf} (D_2,R_2,S_2,N_2) & N = (N_1 - (D_2 \cup R_2 \cup S_2)) \cup N_2 \\ (\emptyset,\emptyset,N_1 \cap (D_2 \cup R_2 \cup S_2)) \vdash_{check} e_1 & (\emptyset,\emptyset,(S_1 \cup \{x_1\}) \cap N_2) \vdash_{check} e_2 \end{array}}{\textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2 \vdash_{inf} ((D_1 \cup D_2) - \{x_1\}, R_1 \cup (R_2 - D_1), ((S_1 - N_2) \cup S_2) - (\{x_1\} \cup D_2 \cup R_2), N - \{x_1\})}\ [\text{LET}_I]$$

$$\frac{\begin{array}{ll} \forall i \in \{1..n\}\,.\, e_i \vdash_{inf} (D_i,R_i,S_i,N_i) & \mathit{def}(\sqcup_{i=1}^n (D_i,R_i,S_i,N_i,P_i)) \\ \forall i \in \{1..n\}\,.\, P_i = \bigcup_{j=1}^{n_i} \{x_{ij}\} & \forall i \in \{1..n\}\,.\, \mathit{def}(\mathit{inh}(\mathit{type}(x), D_i, R_i, S_i, P_i, \mathit{Rec}_i)) \\ \forall i \in \{1..n\}\,.\, \mathit{Rec}_i = \bigcup_{j=1}^{n_i} \{x_{ij} \mid j \in \mathit{RecPos}(C_i)\} & \\ \mathit{type}(x) = \begin{cases} d \text{ if } x \in D \\ r \text{ if } x \in R \\ s \text{ if } x \in S \\ n \text{ e. o. c.} \end{cases} & (D,R,S,N) = \sqcup_{i=1}^n (D_i,R_i,S_i,N_i,P_i) \\ & N' = \begin{cases} N & \text{if } x \in D \cup R \cup S \\ N \cup \{x\} & \text{if } x \notin D \cup R \cup S \end{cases} \\ \multicolumn{2}{c}{\forall i \in \{1..n\}\,.\, ((D \cup D_i') \cap N_i, R \cup ((R_i' \cup R_i'' \cup R_i''') - D_i), (S \cup S_i') \cap N_i) \vdash_{check} e_i} \\ \multicolumn{2}{l}{\textbf{where}\ \ D_i' = \emptyset \qquad R_i' = \begin{cases} \mathit{Rec}_i \text{ if } \mathit{type}(x) = d \\ \emptyset \quad \text{otherwise} \end{cases} \qquad S_i' = \begin{cases} P_i - \mathit{Rec}_i & \text{if } \mathit{type}(x) = d \\ P_i - R_i'' & \text{if } \mathit{type}(x) = r \\ P_i & \text{if } \mathit{type}(x) = s \\ \emptyset & \text{otherwise} \end{cases}} \\ \multicolumn{2}{l}{\quad R_i'' = \{y \in P_i \cap \mathit{sharerec}(z, e_i) \mid z \in (D \cup D_i') \cap N_i\}} \\ \multicolumn{2}{l}{\quad R_i''' = \{y \in D \cap \mathit{sharerec}(z, e_i) \mid z \in (D \cup D_i') \cap N_i\} - (D \cap N_i)} \\ \multicolumn{2}{l}{\quad R_i'' \cap (S_i \cup S_i') = \emptyset} \end{array}}{\textbf{case}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n \vdash_{inf} (D,R,S,N')}\ [\text{CASE}_I]$$

$$\frac{\begin{array}{ll} \forall i \in \{1..n\}\,.\, e_i \vdash_{inf} (D_i,R_i,S_i,N_i) & \mathit{def}(\sqcup_{i=1}^n (D_i,R_i,S_i,N_i,P_i)) \\ \forall i \in \{1..n\}\,.\, P_i = \bigcup_{j=1}^{n_i} \{x_{ij}\} & \forall i \in \{1..n\}\,.\, \mathit{def}(\mathit{inh}!(D_i, R_i, S_i, P_i, \mathit{Rec}_i)) \\ \forall i \in \{1..n\}\,.\, \mathit{Rec}_i = \bigcup_{j=1}^{n_i} \{x_{ij} \mid j \in \mathit{RecPos}(C_i)\} & R \cap L = \emptyset\ \wedge\ \mathit{type}(x) = T@\rho \\ R = \mathit{sharerec}(x, \textbf{case!}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n) & \\ L = \bigcup_{i=1}^n \mathit{fv}(e_i) & (D,R',S,N) = \sqcup_{i=1}^n (D_i,R_i,S_i,N_i,P_i) \\ \multicolumn{2}{c}{\forall i \in \{1..n\}\,.\, ((D \cup \mathit{Rec}_i) \cap N_i, R \cup R' \cup (R_i' \cup R_i'') - D_i, (S \cup (P_i - \mathit{Rec}_i)) \cap N_i) \vdash_{check} e_i} \\ \multicolumn{2}{l}{\quad \textbf{where}\ \ R_i' = \{y \in P_i \cap \mathit{sharerec}(z, e_i) \mid z \in (D \cup \mathit{Rec}_i) \cap N_i\} - (\mathit{Rec}_i \cap N_i)} \\ \multicolumn{2}{l}{\qquad\quad R_i'' = \{y \in D \cap \mathit{sharerec}(z, e_i) \mid z \in D \cap N_i\} - (D \cap N_i)} \\ \multicolumn{2}{l}{\qquad\quad R_i' \cap (P_i - \mathit{Rec}_i) = \emptyset\ \wedge\ \{y \in \mathit{sharerec}(z, e_i) \mid z \in \mathit{Rec}_i\} \cap (P_i - \mathit{Rec}_i) = \emptyset} \end{array}}{\textbf{case!}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n \vdash_{inf} (D \cup \{x\}, (R \cup R') - \{x\}, S, N)}\ [\text{CASE!}_I]$$

Fig. 5. Bottom-up inference rules

invariant of this set of rules is that the four sets inferred for each expression e are pairwise disjoint and their union is a superset of e's free variables. The set R may contain variables in scope but not free in e. This is due to the use of the set *sharerec* consisting of *all* variables in scope satisfying the sharing property. The predicates and least upper bound appearing in the rules [CASE$_I$] and [CASE!$_I$] are defined in Figures 6 and 7.

In Figure 8 we show the top-down checking rules. A judgement $(D, R, S) \vdash_{check} e$ should be understood that the sets of marked variables D, R, S are correctly propagated down the expression e. One invariant in this case is that the three sets are pairwise disjoint and that the union of D and S is contained in the fourth set N inferred from the expression by the $\vdash_{inf}$ rules. It can be seen that the $\vdash_{inf}$ rules may invoke the $\vdash_{check}$ rules. However, the $\vdash_{check}$ rules do not invoke the $\vdash_{inf}$ ones. The ocurrences of $\vdash_{inf}$ in the $\vdash_{check}$ rules should be interpreted as a remembering of the sets that were inferred in the bottom-up mode and that the algorithm recorded in the AST. So there is no need to infer them again.

The rules [VAR$_I$], [COPY$_I$] and [REUSE$_I$] assign to the corresponding variable a safe, don't-know and condemned mark respectively. If a variable occurs as a parameter of a data constructor then it gets a safe mark, as specified by the rule [CONS$_I$]. For the case of function application (rule [APP$_I$]) we obtain from the signature Σ the positions of the parameters which are known to be condemned (I_D) and safe (I_S). The remaining ones belong to the set I_N of unknown positions. The actual parameters in the function application get the corresponding mark. The disjointness conditions in the rule [APP$_I$] prevent a variable from occurring at two condemned positions, or at a safe and a condemned position simultaneously. In rules [REUSE$_I$] and [APP$_I$] all variables belonging to the set R are returned as in-danger, in order to preserve the invariant of the type system mentioned above.

The rule [LET$_I$] correspond with the rules [LET1] and [LET2] of the type system. The results of the subexpressions e_1 and e_2 are checked by means of the assumption $(D_1 \cup R_1) \cap fv(e_2) = \emptyset$, corresponding to the $\rhd^L$ operator of the type system. Moreover, if a variable gets a condemned, in-danger or safe mark in e_2 then it can't be used destructively or become in danger in e_1, because of the operator $\rhd^L$. Hence this variable has to be propagated as safe through e_1 by means of a $\vdash_{check}$. According to the type system, the variables belonging to R_2 could also be propagated through e_1 with an unsafe mark. However, the inference algorithm resolves the non-determinism of the type system by assigning a maximal number of safe marks.

To infer the four sets for a **case**/**case!** expression (rules [CASE$_I$] and [CASE!$_I$]) we have to infer the result from each alternative. The operator $\sqcup$ ensures the consistency of the marks inferred for a variable: if a variable gets two different marks in two distinct branches then at least one of them must be a don't-know mark. On the other hand, the inherited types of the pattern variables in each branch are checked via the *inh* and *inh*! predicates. A mark may be propagated top-down through the AST (by means of $\vdash_{check}$ rules) in one of the following cases:

1. A variable gets a don't-know mark in a branch e_j and a different mark in a branch e_k. The mark obtained from e_k must be propagated through e_j.
2. A pattern variable gets a don't-know mark in a branch e_j. Its inherited type must be propagated through e_j. That is what the sets D'_i, R'_i and S'_i of the rule [CASE$_I$] achieve.

$$
\begin{array}{l}
def(inh(n, D_i, R_i, S_i, P_i, Rec_i)) \equiv true \\
def(inh(s, D_i, R_i, S_i, P_i, Rec_i)) \equiv P_i \cap (D_i \cup R_i) = \emptyset \\
def(inh(r, D_i, R_i, S_i, P_i, Rec_i)) \equiv P_i \cap D_i = \emptyset \\
def(inh(d, D_i, R_i, S_i, P_i, Rec_i)) \equiv Rec_i \cap (D_i \cup S_i) = \emptyset \ \wedge \ (P_i - Rec_i) \cap (D_i \cup R_i) = \emptyset \\
def(inh!(D_i, R_i, S_i, P_i, Rec_i)) \equiv Rec_i \cap (R_i \cup S_i) = \emptyset \ \wedge \ (P_i - Rec_i) \cap (D_i \cup R_i) = \emptyset
\end{array}
$$

Fig. 6. Predicates inh and $inh!$

$$
\begin{array}{l}
def(\sqcup_{i=1}^{n}(D_i, R_i, S_i, N_i, P_i)) \equiv \forall i, j \in \{1..n\} \,.\, i \neq j \Rightarrow \quad (D_i - P_i) \cap (R_j - P_j) = \emptyset \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (D_i - P_i) \cap (S_j - P_j) = \emptyset \wedge (R_i - P_i) \cap (S_j - P_j) = \emptyset \\
\sqcup_{i=1}^{n}(D_i, R_i, S_i, N_i, P_i) \stackrel{\text{def}}{=} (D, R, S, N) \ \textbf{where} \ \begin{cases} D = \bigcup_{i=1}^{n}(D_i - P_i) & R = \bigcup_{i=1}^{n}(R_i - P_i) \\ S = \bigcup_{i=1}^{n}(S_i - P_i) & N = \left(\bigcup_{i=1}^{n}(N_i - P_i)\right) - (D \cup R \cup S) \end{cases}
\end{array}
$$

Fig. 7. Least upper bound definitions

$$
\frac{}{(\emptyset, R, \emptyset) \vdash_{check} c} \ [\text{LIT}_C] \qquad \frac{}{(\emptyset, R, \emptyset) \vdash_{check} x} \ [\text{VAR}_C] \qquad \frac{}{(\emptyset, R, \emptyset) \vdash_{check} x!} \ [\text{REUSE}_C]
$$

$$
\frac{}{(\{x\}, R, \emptyset) \vdash_{check} x@r} \ [\text{COPY1}_C] \qquad \frac{}{(\emptyset, R, \emptyset) \vdash_{check} x@r} \ [\text{COPY2}_C] \qquad \frac{}{(\emptyset, R, \{x\}) \vdash_{check} x@r} \ [\text{COPY3}_C]
$$

$$
\frac{}{(\emptyset, R, \emptyset) \vdash_{check} C \ \overline{a_i}^n \ @r} \ [\text{CONS}_C] \qquad \frac{\begin{array}{c} f \ \overline{a_i}^n \ @\overline{r_j}^l \vdash_{inf} (D, R, S, N) \\ \forall a_i \in D_p \,.\, (\#j : 1 \leq j \leq n : a_i = a_j) = 1 \end{array}}{(D_p, R_p, S_p) \vdash_{check} f \ \overline{a_i}^n @\overline{r_j}^l} \ [\text{APP}_C]
$$

$$
\frac{\begin{array}{l} e_1 \vdash_{inf} (D_1, R_1, S_1, N_1) \ R_p \cap S_1 = \emptyset \ \wedge \ ((D_p \cap N_1) \cup R_p \cup R_p'') \cap fv(e_2) = \emptyset \\ e_2 \vdash_{inf} (D_2, R_2, S_2, N_2) \quad \exists z \in D_p \cap N_2 \,.\, x_1 \in sharerec(z, e_2) \Rightarrow x_1 \in D_2 \\ (D_p \cap N_1, R_p, S_p \cap N_1) \vdash_{check} e_1 \qquad (D_p \cap N_2, R_p \cup (R_p' - D_2), S_p \cap N_2) \vdash_{check} e_2 \\ \textbf{where} \ R_p' = \{y \in ((D_p \cap N_1) \cup D_1) \cap sharerec(z, e_2) \mid z \in D_p \cap N_2\} - (N_2 \cup \{x_1\}) \\ \qquad R_p'' = \{y \in sharerec(z, e_1) \mid z \in D_p \cap N_1\} \end{array}}{(D_p, R_p, S_p) \vdash_{check} \textbf{let} \ x_1 = e_1 \ \textbf{in} \ e_2} \ [\text{LET}_C]
$$

$$
\frac{\begin{array}{l} \forall i \in \{1..n\} \,.\, e_i \vdash_{inf} (D_i, R_i, S_i, N_i) \\ \forall i \in \{1..n\} \,.\, P_i = \bigcup_{j=1}^{n_i}\{x_{ij}\} \\ \forall i \in \{1..n\} \,.\, Rec_i = \bigcup_{j=1}^{n_i}\{x_{ij} \mid j \in RecPos(C_i)\} \\ D = \bigcup_{i=1}^{n}(D_i - P_i) \\ \quad x \in D_p \cup R_p \cup S_p \Rightarrow \forall i \in \{1..n\} \,.\, def(inh(type(x), D_i, R_i, S_i, P_i, Rec_i)) \end{array} \quad type(x) = \begin{cases} d \text{ if } x \in D_p \\ r \text{ if } x \in R_p \\ s \text{ if } x \in S_p \\ n \text{ otherwise} \end{cases}}{\begin{array}{l} \forall i \in \{1..n\} \,.\, ((D_p \cup D_{p_i}) \cap N_i, (R_p \cup R_{p_i} \cup R_{p_i}' \cup R_{p_i}'') - D_i, (S_p \cup S_{p_i}) \cap N_i) \vdash_{check} e_i \\ \textbf{where} \ D_{p_i} = \emptyset \qquad R_{p_i} = \begin{cases} Rec_i \text{ if } type(x) = d \\ \emptyset \quad \text{otherwise} \end{cases} \qquad S_{p_i} = \begin{cases} P_i - Rec_i & \text{if } type(x) = d \\ P_i - R_{p_i}' & \text{if } type(x) = r \\ P_i & \text{if } type(x) = s \\ \emptyset & \text{otherwise} \end{cases} \end{array}} \ [\text{CASE}_C]
$$

$$
\frac{\begin{array}{l} R_{p_i}' = \{y \in P_i \cap sharerec(z, e_i) \mid z \in D_p \cap N_i\} \\ R_{p_i}'' = \{y \in (D_p \cup D) \cap sharerec(z, e_i) \mid z \in D_p \cap N_i\} - (D_p \cap N_i) \\ R_{p_i}' \cap (S_i \cup S_{p_i}) = \emptyset \ \wedge \ R_p \cap S_i = \emptyset \end{array}}{(D_p, R_p, S_p) \vdash_{check} \textbf{case} \ x \ \textbf{of} \ \overline{C_i \ \overline{x_{ij}}^{n_i} \rightarrow e_i}^n} \ [\text{CASE}_C]
$$

$$
\frac{\begin{array}{l} \forall i \in \{1..n\} \,.\, e_i \vdash_{inf} (D_i, R_i, S_i, N_i) \\ \forall i \in \{1..n\} \,.\, P_i = \bigcup_{j=1}^{n_i}\{x_{ij}\} \qquad\qquad D = \bigcup_{i=1}^{n}(D_i - P_i) \\ \forall i \in \{1..n\} \,.\, Rec_i = \bigcup_{j=1}^{n_i}\{x_{ij} \mid j \in RecPos(C_i)\} \\ \quad \forall i \in \{1..n\} \,.\, \{y \in (P_i - Rec_i) \cap sharerec(z, e_i) \mid z \in D_p \cap N_i\} = \emptyset \\ \qquad \forall i \in \{1..n\} \,.\, (D_p \cap N_i, R_p \cup (R_{p_i}' - D_i), S_p \cap N_i) \vdash_{check} e_i \\ \textbf{where} \ R_{p_i}' = \{y \in (D_p \cup D) \cap sharerec(z, e_i) \mid z \in D_p \cap N_i\} - (D_p \cup N_i) \end{array}}{(D_p, R_p, S_p) \vdash_{check} \textbf{case!} \ x \ \textbf{of} \ \overline{C_i \ \overline{x_{ij}}^{n_i} \rightarrow e_i}^n} \ [\text{CASE!}_C]
$$

Fig. 8. Top-down checking rules

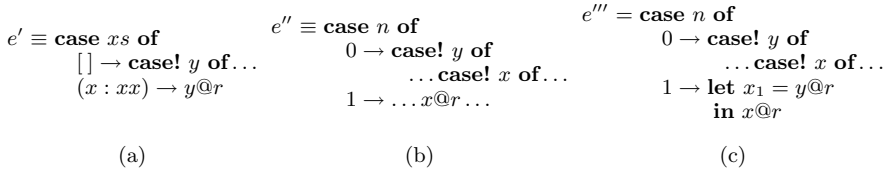

Fig. 9. Motivating examples for R_i'', R_i''' in [CASE$_I$] and R_p' in [LET$_C$].

3. A variable belongs to R_i'' or R_i'''. These cases are explained below.

There exists an invariant in the $\vdash_{check}$ rules (see below) which specifies the following: if a variable x is propagated top-down with a condemned mark, those variables sharing a recursive substructure with x either have been inferred previously as condemned (via the $\vdash_{inf}$ rules) or have been propagated with an unsafe (i.e. in-danger or condemned) mark as well. The sets R_i'' and R_i''' occur in the [CASE$_I$] and [CASE!$_I$] rules in order to preserve this invariant. The set R_i'' contains the pattern variables which may share a recursive substructure with some condemned variable being propagated top-down through the e_i. An example of this situation is shown in Fig. 9 (a). Let us assume that $x \in \mathit{sharerec}(y, y@r)$. The condemned mark of y obtained in the first branch has to be propagated through the branch guarded by $(x : xx)$. Thus we must propagate an in-danger mark for the variable x, since it shares a recursive substructure of y.

In order to justify the occurrence of R_i''', let us consider the e'' expression shown in Fig. 9 (b). Let e_0 and e_1 be the expressions corresponding to the **case** branches guarded by 0 and 1, respectively. Variables x and y are inferred as condemned in e_0. On the other hand, x is inferred as don't-know in e_1. Hence the condemned mark of x has to be propagated through e_1. We assume that y doesn't belong to any of the four sets inferred for e_1. According to the invariant mentioned above, if $y \in \mathit{sharerec}(x, e_1)$ then y should be propagated through e_1 as well. The fact that $y \in R_i'''$ guarantees this.

The $\vdash_{check}$ rules capture the same verifications as the $\vdash_{inf}$ rules, but in a top-down fashion. The occurrence of the R_p' set in [LET$_C$] is motivated by the example shown in Figure 9 (c). We shall assume that $y \in \mathit{sharerec}(x, x@r)$. In the **let** expression x and y get a don't-know mark, but because of their destructive use in the **case** branch guarded by 0, a condemned mark for x and y has to be propagated through the **let** expression. The mark corresponding to x spreads over the expression $x@r$, but y should also be included there as in-danger, in order to preserve the invariant of $\vdash_{check}$ introduced above, since y shares a recursive structure of x. This is what the set R_p' ensures.

The algorithm is modular in the sense that each function body is independently inferred. The result is reflected in the function type and this type is available for typing the remaining functions. For typing a recursive function a fixpoint computation is needed. In the initial environment a don't-know mark is assigned to each formal argument. After each iteration, some don't-know marks may have turned into condemned, in-danger or safe marks. This procedure continues until the mark for each argument stabilises. If the fixpoint assigns an in-danger mark to an argument, this is rejected as a bad typing. Otherwise, if any don't-know mark remains, this is forced to be a safe mark by the algorithm and propagated down the whole function body by using the $\vdash_{check}$ rules once more. As a consequence, if the algorithm succeeds, every variable inferred as

don't-know during the bottom-up trasversal will eventually get a d, r or s mark (see Appendix for a detailed proof).

If n is the size of the AST for a function body and m is the number of its formal arguments, the algorithm runs in $\Theta(mn^3)$ in the worst case. This corresponds to m iterations of the fixpoint and a top-down traversal at each intermediate expression. We conjecture however that the average case is near to $\Theta(n^2)$, corresponding to a single bottom-up traversal and a single fixpoint iteration.

4.1 Correctness of the inference algorithm

Lemma 1. *Let us assume that during the inference algorithm we have* $e \vdash_{inf} (D, R, S, N)$ *and* $(D', R', S') \vdash_{check} e$ *for an expression* e*. Then*

1. D, R, S *and* N *are pairwise disjoint.*
2. $D \cup S \cup N \subseteq FV(e)$, $R \subseteq scope(e)$ *and* $D \cup R \cup S \cup N \supseteq FV(e)$.
3. $\bigcup_{z \in D} sharerec(z, e) \subseteq D \cup R$.
4. D', R' *and* S' *are pairwise disjoint.*
5. $D' \cup S' \subseteq N$, $R' \subseteq scope(e)$.
6. $\bigcup_{z \in D'} sharerec(z, e) \subseteq D' \cup R' \cup D$.
7. $R' \cap S = \emptyset$, $R' \cap D = \emptyset$.

Proof. (1), (2) and (3) by structural induction on e; (4), (5), (6) and (7) hold at each initial call to $\vdash_{check}$ and it is preserved at each recursive call.

A single subexpression e may suffer more than one $\vdash_{check}$ during the inference algorithm but always with different variables. This is due to the fact, not reflected in the rules, that whenever some variables in the set N inferred for e are forced to get a mark different from n, the decoration in the AST is changed to the new marks. More precisely, if $e \vdash_{inf} (D, R, S, N)$ and $(D', R', S') \vdash_{check} e$, then the decoration is changed to $(D \cup D', R \cup R', S \cup S', N - (D' \cup R' \cup S'))$. So, the next $\vdash_{check}$ for expression e will get a smaller set $N - (D' \cup R' \cup S')$ of don't-know variables and, by Lemma 1, only those variables can be forced to change its mark. As a corollary, the mark for a variable can change during the algorithm from n to d, r or s, but no other transitions between marks are possible.

Let $(D', R', S') \vdash^*_{check} e$ denote the accumulation of all the $\vdash_{check}$ involving e during the algorithm and let D', R' and S' represent the union of respectively all the marks d, r and s forced in these calls to $\vdash_{check}$. If $e \vdash_{inf} (D, R, S, N)$ represent the sets inferred during the bottom-up mode, then $D' \cup R' \cup S' \supseteq N$ must hold, since every variable eventually gets a mark d, r or s.

The next theorem uses the convention $\Gamma(x) = s$ (respectively, r or d) to indicate that x has a safe type (respectively, an in danger or a condemned type) without worrying about which precise type it has.

Theorem 1. *Let us assume that the function declaration* $f\ \overline{x_i}^n\ @\ \overline{r_j}^l = e$ *has been successfully typed by the inference algorithm and let* e' *be any subexpression of* e *for which the algorithm has got* $e' \vdash_{inf} (D, R, S, N)$ *and* $(D', R', S') \vdash^*_{check} e'$*. Then there exists a safe type* s' *and a well-formed type environment* Γ *such that* $\Gamma \vdash e' : s'$*, and* $\forall x \in scope(e')$*:*

$$[x \in D \cup D' \leftrightarrow \Gamma(x) = d] \wedge [x \in S \cup S' \leftrightarrow \Gamma(x) = s] \wedge [x \in R \cup R' \leftrightarrow \Gamma(x) = r]$$

Proof. It can be done by structural induction on e'.

5 Small examples

In this section we show some examples. Firstly, we review the example of appending two lists (Sec. 2). The *Core-Safe* code for `concatD` is as follows:

$$\begin{aligned} \mathit{concatD}\ \mathit{zs}\ \mathit{ys}@r = \mathbf{case!}\ \mathit{zs}\ \mathbf{of} \\ [\,] &\to \mathit{ys} \\ (x : \mathit{xs}) &\to \mathbf{let}\ x_1 = \mathit{concatD}\ \mathit{xs}\ \mathit{ys}\ @r\ \mathbf{in}\ (x : x_1)@r \end{aligned}$$

We shall start with the recursive call to *concatD*. Initially all parameter positions are marked as don't-know and hence the actual arguments *xs* and *ys* belong to set N. In addition to this, variables x and x_1 get an s mark since they are used to build a DS. Joining the results of both auxiliary and main expressions in **let** we get the following sets: $D = \emptyset$, $R = \emptyset$, $S = \{x\}$, $N = \{xs, ys\}$. With respect to the **case!** branch guarded by $[\,]$, the variable *ys* gets a safe mark (rule [VAR$_I$]). Information of both alternatives in **case!** is gathered as follows:

($[\,]$ **guard**)	$D_1 = \emptyset$	$R_1 = \emptyset$	$S_1 = \{ys\}$	$N_1 = \emptyset$	$P_1 = \emptyset$	$Rec_1 = \emptyset$
($x : xs$ **guard**)	$D_2 = \emptyset$	$R_2 = \emptyset$	$S_2 = \{x\}$	$N_2 = \{xs, ys\}$	$P_2 = \{x, xs\}$	$Rec_2 = \{xs\}$

Since *ys* has a safe mark in the branch guarded by $[\,]$ and a don't-know mark in the branch guarded by $(x : xs)$, the safe mark has to be propagated through the latter by means of the $\vdash_{check}$ rules. Moreover, the pattern variable *xs* is also propagated as condemned. The first bottom-up traversal of the AST terminates with the following result: $D = \{zs\}$, $R = \emptyset$, $S = \{ys\}$ and $N = \emptyset$. Consequently the type signature of *concatD* is updated: the first position is now condemned and the second one is safe. Another bottom-up traversal is needed, as the fixpoint has not been reached yet. Now variables *xs* and *ys* belong to sets D and S respectively in the recursive call to *concatD*. Variable *zs* is also marked as in-danger, since it shares a recursive structure with *xs*. However, neither *xs* nor *zs* occur free in the main expression of **let** and hence the rule [LET$_I$] may still be applied. At the end of this iteration a fixpoint has been reached. The final type signature for *concatD* is $\forall a.[a]!@\rho_1 \to [a]@\rho_2 \to \rho_2 \to [a]@\rho_2$.

With respect to the remaining examples shown in Sec. 2, the types inferred by the algorithm are as follows:

$$\begin{aligned} \mathit{treesortD} &:: \forall a, \rho_1, \rho_2\ .\ [a]!@\rho_1 \to \rho_2 \to [a]@\rho_2 \\ \mathit{mkTreeD} &:: \forall \rho_1, \rho_2\ .\ [\mathit{Int}]!@\rho_1 \to \rho_2 \to \mathit{Tree}\ \mathit{Int}@\rho_2 \\ \mathit{insertD} &:: \forall \rho\ .\ \mathit{Int} \to \mathit{Tree}\ \mathit{Int}!@\rho \to \rho \to \mathit{Tree}\ \mathit{Int}@\rho \\ \mathit{splitD} &:: \forall a, \rho_1, \rho_2, \rho_3\ .\ \mathit{Int} \to [a]!@\rho_2 \to \rho_1 \to \rho_2 \to \rho_3 \to ([a]@\rho_1, [a]@\rho_2)@\rho_3 \end{aligned}$$

6 Related Work

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. Their main contribution is a *region inference* algorithm which introduces region annotations at the intermediate language level. An extension of their work [BTV96,TBE$^+$06] allows to *reset* all the data structures in a region whithout deallocating the whole region. The AFL system [AFL95] inserts (as a result of an analysis) allocation and deallocation commands separated from the **letregion** construct which now only brings new regions into scope. In [HMN01] a comparison of these works is done.

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* system, region allocation and deallocation are synchronized with function calls instead of being introduced by a special language construct. But the 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 $match'$ which destroys constructor cells. Its operational behaviour is similar to that of *Safe* **case!**. The main difference is the compile time analysis guaranteeing the safe use of this dangerous feature. A minor difference with [HJ03] is that they do not use the concept of nested regions where DSs are allocated. In [PSM07] a more detailed comparison with 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], where the authors propose a type system for a language which explicitly reuses heap cells. They prove that well-typed programs can be safely translated to an imperative language with an explicit deallocation/reusing mechanism. We summarise here the differences and similitudes with our work.

In the first place, 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. (3) They distinguish two different cartesian products depending on whether there is sharing or not between the tuple components.

Also, there are the following obvious similitudes: (1) They allow several accesses to the same variable, provided that only the last one is destructive. (2) They express the nature of arguments (destructive, read-only and shared, or just read-only) in the function type. (3) They need information about sharing between the variables and the final result of expressions.

But, in our view, the following more essential differences makes our language and 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). We add a third use r (in-danger) arising from a sharing analysis based on abstract interpretation. This use allows us to know more precisely which variables are in danger when some other 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 COPY) we allow the partial order $s \leq r$ and $s \leq d$. It is not clear whether that order gives more power to the system or not. In principle it will allow different 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 combinations than theirs. Let $i \in \{1, 2, 3\}$ the use assigned to x_1, be j the use of a variable z in e_1 and be k the use of the same 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.

Examples of *Safe* programs using respectively the combinations $(1, 2, 3)$ and $(1, 2, 2)$ are the following, where x and z get an s-type in our type system:

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

Both programs take profit from the fact that variable z is not a recursive descendant of x.

References

[AFL95] A. Aiken, M. Fähndrich, and R. Levien. Better static memory management: improving region-based analysis of higher-order languages. In *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 *POPL'96*, pages 171–183. ACM Press, 1996.

[HJ03] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *POPL'03*, 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 *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 *ICFP'99*, pages 70–81. ACM Press, 1999.

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

[MPS08] M. Montenegro, R. Peña, and C. Segura. A type system for safe memory management and its proof of correctness. Technical report, SIC-5-08. Universidad Complutense de Madrid, 2008.

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

[PS04] R. Peña and C. Segura. A First-Order Functional Language for Reasoning about Heap Consumption. In *Proc. of the 16th International Workshop on Implementation of Functional Languages, IFL'04*, pages 64–80, 2004.

[PSM07] R. Peña, C. Segura, and M. Montenegro. A Sharing Analysis for Safe. In *Trends in Functional Programming (Vol. 7)*, pages 109–128. Intellect, 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 Correctness of the inference algorithm (Proof)

A.1 Properties of the inference algorithm

Lemma 1. *Let us assume that during the inference algorithm we have* $e \vdash_{inf} (D, R, S, N)$ *and* $(D', R', S') \vdash_{check} e$ *for an expression* e*. Then the following seven properties hold:*

1. D, R, S *and* N *are pairwise disjoint.*
2. $D \cup S \cup N \subseteq fv(e)$, $R \subseteq scope(e)$ *and* $D \cup R \cup S \cup N \supseteq fv(e)$.
3. $\bigcup_{z \in D} sharerec(z, e) \subseteq D \cup R$
4. D', R' *and* S' *are pairwise disjoint.*
5. $D' \cup S' \subseteq N$, $R' \subseteq scope(e)$.
6. $\bigcup_{z \in D'} sharerec(z, e) \subseteq D' \cup R' \cup D$.
7. $R' \cap S = \emptyset$, $R' \cap D = \emptyset$.

Proof.

Properties (1), (2) and (3)

These properties can be proven by induction on the structure of e.

$\boxed{e \equiv c}$ $\boxed{e \equiv x}$ $\boxed{e \equiv x@r}$ $\boxed{e \equiv x!}$ $\boxed{e \equiv C\ \overline{a_i}^n\ @r}$

Trivial, by simple inspection of the corresponding $\vdash_{inf}$ rules. For the case $e \equiv x!$ the property $R \subseteq scope(e)$ holds due to the fact that *sharerec* function only returns variables belonging to $scope(e)$. In addition, we have:

$$\begin{aligned} \bigcup_{z \in D} sharerec(z, e) &= sharerec(x, e) \\ &= (sharerec(x, e) - \{x\}) \cup \{x\} \\ &= R \cup D \end{aligned}$$

$\boxed{e \equiv f\ \overline{a_i}^n\ @\ \overline{r_j}^l}$

We have got the following sets:

$$\begin{aligned} D &= \bigcup_{i=1}^{n} D_i \\ R &= \bigcup_{i=1}^{n} \{sharerec(a_i, f\ \overline{a_i}^n\ @\ \overline{r_j}^l) - \{a_i\} \mid a_i \in D_i\} \\ S &= \bigcup_{i=1}^{h} S_i \\ N &= \bigcup_{i=1}^{h} N_i - \bigcup_{i=1}^{n} S_i \end{aligned}$$

It follows that $D \cap R = \emptyset$, $D \cap S = \emptyset$, $D \cap N = \emptyset$, $R \cap S = \emptyset$, $R \cap N = \emptyset$ because of the hypothesis in the [APP$_I$] rule and due to the fact that I_D, I_S and I_N are pairwise disjoint. Besides that, the property $S \cap N = \emptyset$ holds trivially.

With respect to the property (2), it can be easily proven from the definition of *sharerec* that $R \subseteq scope(e)$. The remaining set inclusions hold, since we have $D \cup S \cup N = fv(e)$: let us assume that $x \in D$. It follows that $x \in D_i$ for some i and therefore, x is a parameter in the function call. Cases $x \in S$ and $x \in N$ are similar. On the other hand, if $x \in fv(e)$ then x belongs to D_i, S_i or N_i, since $I_D \cup I_S \cup I_N = \{1..n\}$.

To prove the property (3), let $y \in sharerec(z, e)$ for some variable $z \in \bigcup_{i=1}^{n} D_i$. Then there exists an $i \in \{1..n\}$ so that $y \in sharerec(a_i, e)$ and $a_i \in D_i$ hold:

- If $y = a_i$, then $y \in D$.
- If $y \neq a_i$, then $y \in R$.

$\boxed{e \equiv \textbf{let } x_1 = e_1 \textbf{ in } e_2}$

The inferred sets are:

$$\begin{aligned} D &= (D_1 \cup D_2) - \{x_1\} \\ R &= R_1 \cup (R_2 - D_1) \\ S &= ((S_1 - N_2) \cup S_2) - (\{x_1\} \cup D_2 \cup R_2) \\ N &= ((N_1 - (D_2 \cup R_2 \cup S_2)) \cup N_2) - \{x_1\} \end{aligned}$$

The induction hypothesis establishes that D_1, R_1, S_1 and N_1 are pairwise disjoint, and D_2, R_2, S_2, N_2 as well.

It can be shown that $D \cap R = \emptyset$ from the induction hypothesis and from the fact that $R_1 \cap D_2 = \emptyset$. The latter holds due to the hypothesis $(D_1 \cup R_1) \cap fv(e_2) = \emptyset$ in $[\text{LET}_I]$. Indeed, we have $D_2 \subseteq fv(e_2)$ by application of the induction hypothesis for the property (2).

The proof for $D \cap S = \emptyset$ is similar: it has to be shown that $D_1 \cap S_2 = \emptyset$, which can be deduced again from the the fact that $(D_1 \cup R_1) \cap fv(e_2) = \emptyset$ and $S_2 \subseteq fv(e)$. The disjointness of the remaining set pairs can be proven in a similar way.

To prove the property (2) it can be easily seen that $D \cup S \cup N \subseteq fv(e)$:

$$\begin{aligned} fv(e) &= (fv(e_1) \cup fv(e_2)) - \{x_1\} \\ &= (fv(e_1) - \{x_1\}) \cup (fv(e_2) - \{x_1\}) \\ &\supseteq ((D_1 \cup S_1 \cup N_1) - \{x_1\}) \cup ((D_2 \cup S_2 \cup N_2) - \{x_1\}) \\ &= ((D_1 \cup D_2) - \{x_1\}) \cup ((S_1 \cup S_2) - \{x_1\}) \cup ((N_1 \cup N_2) - \{x_1\}) \\ &\supseteq D \cup S \cup N \end{aligned}$$

On the other hand, let $x \in R$. According to the definition of the set R we distinguish two cases:

- $x \in R_1$: By the induction hypothesis we have that $R_1 \subseteq \mathit{scope}(e_1)$ and hence, $x \in \mathit{scope}(e_1) = \mathit{scope}(e)$.
- $x \in R_2$: It holds that $x \in \mathit{scope}(e_2) = \mathit{scope}(e) \cup \{x_1\}$. However, we have $x \neq x_1$; otherwise we would have $x_1 \in R_2$ and the rule $[\text{LET}_I]$ could not be applied. Therefore, $x \in \mathit{scope}(e)$.

So far it has been proven that $R \subseteq \mathit{scope}(e)$. It has to be shown that $D \cup R \cup S \cup N \supseteq fv(e)$. Let us assume that $x \in fv(e)$, therefore $x \neq x_1$.

- $x \in fv(e_1)$: The induction hypothesis establishes that $x \in D_1 \cup R_1 \cup S_1 \cup N_1$.
  - If $x \in D_1$ then $x \in D$.
  - If $x \in R_1$ then $x \in R$.
  - If $x \in S_1$, $x \notin N_2$ and $x \notin D_2 \cup R_2$ then $x \in S$.
  - If $x \in S_1$ and $x \in N_2$ then $x \in N$.
  - If $x \in S_1$ but $x \in D_2 \cup R_2$ then we have $x \in R$ or $x \in D$. The proof is similar to the one shown for the case $x \in fv(e_2)$.
  - If $x \in N_1$ and $x \notin (D_2 \cup R_2 \cup S_2)$ then $x \in N$.

- If $x \in N_1$ and $x \in (D_2 \cup R_2 \cup S_2)$, although is it not necessarily true that $x \in \mathit{fv}(e_2)$ (this case is shown below), the proof is similar to the case $\mathit{fv}(e_2)$, because $x \neq x_1$.

- $x \in \mathit{fv}(e_2)$: By induction hypothesis we have $x \in D_2 \cup R_2 \cup S_2 \cup N_2$.
  - If $x \in D_2$ then $x \in D$.
  - If $x \in R_2$ and $x \notin D_1$ then $x \in R$.
  - If $x \in R_2$ and $x \in D_1$ then $x \in D$.
  - If $x \in S_2$ then $x \in S$.
  - If $x \in N_2$ then $x \in N$.

In any case it holds that $x \in D \ \vee \ x \in R \ \vee \ x \in S \ \vee \ x \in N$ and hence $x \in D \cup R \cup S \cup N$.

To prove the property (3), let $y \in \mathit{sharerec}(z, e)$ for some $z \in (D_1 \cup D_2) - \{x_1\}$. It holds that $y \neq x_1$, since $x_1 \notin \mathit{scope}(e)$. According to the set to which z belongs, the following cases can be distinguished:

- $z \in D_1$
  Since $\mathit{scope}(e_1) = \mathit{scope}(e)$ we have $y \in \mathit{sharerec}(z, e_1)$ for some $z \in D_1$. The induction hypothesis establishes that

$$\bigcup_{z \in D_1} \mathit{sharerec}(z, e_1) \subseteq D_1 \cup R_1$$

  Therefore, $y \in D_1 \cup R_1$. If $y \in D_1$ then $y \in D$. If $y \in R_1$ then $y \in R$.
- $z \in D_2$
  In that case we have $\mathit{scope}(e_2) = \mathit{scope}(e) \cup \{x_1\}$. However, we have $y \in \mathit{sharerec}(z, e_2)$ for some $z \in D_2$ as well, since $y \neq x_1$. Again the induction hypothesis can be applied:

$$\bigcup_{z \in D_2} \mathit{sharerec}(z, e_2) \subseteq D_2 \cup R_2$$

  So that we obtain $y \in D_2 \cup R_2$. If $y \in D_2$ then $y \in D$. If $y \in R_2$ then it holds that $y \in R$, or $y \in D$.

$\boxed{e \equiv \textbf{case } x \textbf{ of } \overline{C\ \overline{x_{ij}}^{n_i} \rightarrow e_i}^{n}}$

The following sets are inferred:

$$\begin{aligned} D &= \textstyle\bigcup_{i=1}^{n}(D_i - P_i) \\ R &= \textstyle\bigcup_{i=1}^{n}(R_i - P_i) \\ S &= \textstyle\bigcup_{i=1}^{n}(S_i - P_i) \\ N' &= (\textstyle\bigcup_{i=1}^{n}(N_i - P_i)) - (D \cup R \cup S) \\ N &= \begin{cases} N' \cup \{x\} & \text{if } x \notin D \cup R \cup S \\ N' & \text{otherwise} \end{cases} \end{aligned}$$

In order to prove (1) we begin proving that $D \cap R = \emptyset$ by contradiction. Let us assume that there exists a variable z such that $z \in D$ and $z \in R$. Hence, there exist $i, j \in \{1..n\}$ such that $z \in (D_i - P_i)$ and $z \in (R_j - P_j)$.

- If $i = j$ then we have that $z \in D_i$ and $z \in R_i$ contradicting the induction hypothesis, which establishes the disjointness of D_i and R_i.
- If $i \neq j$ then we have $(D_i - P_i) \cap (R_j - P_j) \neq \emptyset$, contradicting the fact that $\sqcup$ operator is well-defined.

The proof for $D \cap S = \emptyset$ and $R \cap S = \emptyset$ is similar. On the other hand, it holds trivially that N' is disjoint from D, R and S. It can also be proved by case distinction ($z = x$ or $z \neq x$) that N is disjoint from D, R and S.

Let us prove property (2). First:

$$\begin{array}{l} fv(e) = \bigcup_{i=1}^{n} (fv(e_i) - \{x_{ij} \mid j \in \{1..n_i\}\}) \cup \{x\} \\ \quad = \bigcup_{i=1}^{n} (fv(e_i) - P_i) \cup \{x\} \\ \quad \supseteq \bigcup_{i=1}^{n} ((D_i \cup S_i \cup N_i) - P_i) \cup \{x\} \\ \quad = \bigcup_{i=1}^{n} (D_i - P_i) \cup \bigcup_{i=1}^{n} (S_i - P_i) \cup \bigcup_{i=1}^{n} (N_i - P_i) \cup \{x\} \\ \quad \supseteq \bigcup_{i=1}^{n} (D_i - P_i) \cup \bigcup_{i=1}^{n} (S_i - P_i) \cup (\bigcup_{i=1}^{n} (N_i - P_i) - (D \cup R \cup S)) \cup \{x\} \\ \quad = D \cup S \cup N' \cup \{x\} \\ \quad \supseteq D \cup S \cup N \end{array}$$

Second, let us assume that $z \in R$. Then there exists an $i \in \{1..n\}$ such that $z \in R_i$ and $z \notin P_i$ and hence:

$$z \in R_i \Rightarrow z \in \mathit{scope}(e_i) \Rightarrow z \in \mathit{scope}(e)$$

The first step follows from the induction hypothesis and the last one follows from the fact that $\mathit{scope}(e_i) = \mathit{scope}(e) \cup P_i$ and that $z \notin P_i$.

In order to prove $D \cup R \cup S \cup N \supseteq fv(e)$, let $z \in fv(e)$.

- If $z = x$ then either $z \in N$ or $z \in D \cup R \cup S$.
- If $z \neq x$ then $z \in fv(e_i)$ and $z \notin P_i$ for some $i \in \{1..n\}$. From the induction hypothesis we have $z \in D_i \cup R_i \cup S_i \cup N_i$.
  - If $z \in D_i$ (resp. R_i, S_i), then we have $z \in D$ (resp. R, S), since $z \notin P_i$.
  - If $z \in N_i$ then either $z \in N$ or $z \in D \cup R \cup S$.

With respect to the property (3) let $y \in \mathit{sharerec}(z, e)$ for some $z \in D$. Since no variable in P_i belongs to $\mathit{scope}(e)$ we have $y \notin P_i$ for all $i \in \{1..n\}$. On the other hand, $z \in D_j$ and $z \notin P_j$ for some $j \in \{1..n\}$. The induction hypothesis establishes that:

$$\forall i \in \{1..n\} \,.\, \bigcup_{z \in D_i} \mathit{sharerec}(z, e_i) \subseteq D_i \cup R_i$$

Since $\mathit{scope}(e) \subseteq \mathit{scope}(e_j)$ holds, it can be seen that $y \in \mathit{sharerec}(z, e_j)$ for some $z \in D_j$. This implies that $y \in D_j \cup R_j$. If $y \in D_j$ then $y \in D$, since $y \notin P_j$. In a similar way, if $y \in R_j$ then $y \in R$.

$$\boxed{e \equiv \textbf{case!}\ x\ \textbf{of}\ \overline{C\ \overline{x_{ij}}^{n_i} \rightarrow e_i}^{n}}$$

The following sets are inferred:

$$\begin{array}{l} D = D^* \cup \{x\} \\ R = (\mathit{sharerec}(x, e) \cup R^*) - \{x\} \\ S = S^* \\ N = N^* \\ \qquad \textbf{where}\ (D^*, R^*, S^*, N^*) = \sqcup_{i=1}^{n} (D_i, R_i, S_i, N_i, P_i) \end{array}$$

The proof of the disjointness of D^*, R^*, S^* and N^* is similar to the one seen for the **case**. It has to be proven that:

$$\begin{array}{cccc} x \notin S & x \notin N & \mathit{sharerec}(x,e) \cap D^* = \emptyset \\ \mathit{sharerec}(x,e) \cap S = \emptyset & \mathit{sharerec}(x,e) \cap N = \emptyset \end{array}$$

All of these can be obtained from the hypothesis $R \cap L = \emptyset$ in rule [CASE!$_I$]. If a variable belongs to D^*, S or to N then it also belongs to $fv(e)$ by property (2) (see below).

Using the same reasoning as the one seen for **case** we have $fv(e) \supseteq D^* \cup S^* \cup N^* \cup \{x\}$ and hence $fv(e) \supseteq D \cup S \cup N$.

On the other hand, let z be a variable such that $z \in R$:

- If $z \in \mathit{sharerec}(x,e)$ then $z \in \mathit{scope}(e)$ by the definition of the function *sharerec*, which only returns variables in scope.
- If $z \in R'$ then $z \in \mathit{scope}(e)$ (see **case** above).

Then we have $R \subseteq \mathit{scope}(e)$. The proof for $D \cup R \cup S \cup N \supseteq fv(e)$ is similar to the one seen in **case** expressions, but in this case the variable x always belongs to D.

In order to prove (3), let $y \in \mathit{sharerec}(z,e)$ for some $z \in D$. We shall prove that $y \in D \cup R$: If $y = x$ then it holds that $y \in D$, so let us assume in the following that $y \neq x$. We distinguish cases:

- If $z = x$ then $y \in \mathit{sharerec}(x,e)$ and hence $y \in R$.
- If $z \neq x$ then it can be proven that $y \in D^* \cup R^*$ (see **case**). Hence $y \in D \cup R$.

Properties (4), (5), (6) and (7)

These four properties are related with the $\vdash_{check}$ rules. In order to prove them we will show that they hold in each initial call to $\vdash_{check}$ and that they are preserved in each recursive call.

Initial calls in [LET$_I$]

Properties (4), (6) and (7) hold trivially, since $R' = D' = \emptyset$. With respect to (5) we have that $R' = \emptyset \subseteq \mathit{scope}(e)$. In addition, since $D' = \emptyset$ and because of the intersection with N_1 in the *check* on e_1 and the intersection with N_2 in the *check* on e_2 we have $D' \cup S' \subseteq N$ in each case.

Initial calls in [CASE$_I$] Let $e \equiv \textbf{case}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \rightarrow e_i}^{n}$

A check on the $i-$th branch is done with the following sets:

$$\begin{array}{l} D' = (D \cup D'_i) \cap N_i \\ R' = R \cup ((R'_i \cup R''_i \cup R'''_i) - D_i) \\ S' = (S \cup S'_i) \cap N_i \end{array}$$

Property (4) can be proven considering each pair separately:

- $D' \cap R' = \emptyset$.
  From (1) it follows that $D \cap R = \emptyset$. It also holds that $D \cap (R'_i \cup R''_i) = \emptyset$, since R'_i can only have variables from $Rec_i \subseteq P_i$, the set R''_i can only have variables from P_i, and D does not contain any variable from P_i. Moreover, R'''_i does not contain variables from $D \cap N_i$ and $D' \subseteq D \cap N_i$. Thus $R'''_i \cap D' = \emptyset$ holds. Finally, we have that $D'_i = \emptyset$ and hence disjoint from R'.

- $D' \cap S' = \emptyset$
  From (1) it follows that $D \cap S = \emptyset$. Moreover, D does not have variables from P_i and S'_i only contains variables from P_i, so we have $D \cap S'_i = \emptyset$. Since $D'_i = \emptyset$, it holds that $D'_i \cap S' = \emptyset$.
- $R' \cap S' = \emptyset$
  Using the same reasoning as the one seen above the property $R \cap S' = \emptyset$ can be proven. In addition, it holds that $S \cap (R'_i \cup R''_i) = \emptyset$, since R'_i and R''_i only contain variables from P_i, which is disjoint from S. By case distinction on $\mathit{type}(x)$ and using the fact that $\mathit{Rec}_i \cap (P_i - \mathit{Rec}_i) = \emptyset$ it can be proven that $R'_i \cap S'_i = \emptyset$. Moreover, since S and D are disjoint and R'''_i does not contain any element from $P_i \supseteq S'_i$, we have that S' and R'''_i are disjoint. The property $R''_i \cap S'_i = \emptyset$ can be obtained from one of the assumptions in rule [CASE$_I$].

The property (5) can be easily proven from the definition of D' and S', in which there is an intersection which $N_i \subseteq N$. On the other hand we obtain $R \subseteq \mathit{scope}(e)$ from the property (2) and hence $R \subseteq \mathit{scope}(e) \subseteq \mathit{scope}(e_i)$. Moreover:

$$R'_i \subseteq \mathit{Rec}_i \subseteq \mathit{scope}(e_i)$$
$$R''_i \subseteq P_i \subseteq \mathit{scope}(e_i)$$
$$R'''_i \subseteq D \subseteq \mathit{scope}(e) \subseteq \mathit{scope}(e_i)$$

Thus it follows that $R' \subseteq \mathit{scope}(e_i)$.

In order to prove (6), let us assume, for some $i \in \{1..n\} : y \in \mathit{sharerec}(z, e_i)$, where $z \in (D \cup D'_i) \cap N_i = D \cap N_i$. Whe shall prove $y \in D' \cup R' \cup D_i$ by case distinction:

- $y \in P_i$
  It holds that $y \in R''_i$ and hence, either $y \in D_i$ or $y \in R'$.
- $y \notin P_i$.
  It holds that $y \in \mathit{sharerec}(z, e)$ for some $z \in D$. The property (3) allows us to establish the following:

  $$\bigcup_{z \in D} \mathit{sharerec}(z, e) \subseteq D \cup R$$

  Hence, it follows that $y \in D \cup R$ and the following case distinction is made:
  - If $y \in D$ and $y \in N_i$ then $y \in D'$.
  - If $y \in D$ but $y \notin N_i$ then $y \in R'''_i$ from which $y \in R' \ \vee \ y \in D_i$ can be proven.
  - If $y \in R$ then $y \in R'$.

  In order to prove (7) we consider each pair of sets separately:

- $R' \cap D_i = \emptyset$
  We prove that $R \cap D_i = \emptyset$ by contradiction: Let $x \in R \wedge x \in D_i$. Then there exists a $j \in \{1..n\}$ such that $x \in R_j$. Obviously $x \notin P_j \ \wedge \ x \notin P_i$. If $j = i$ then $R_i \cap D_i \neq \emptyset$, which contradicts (1), since we have $e_i \vdash_{\mathit{inf}} (D_i, R_i, S_i, N_i)$. On the other hand, if $j \neq i$ then it holds that $x \in (R_j - P_j) \cap (D_i - P_i)$, contradicting the fact that $\sqcup$ is well-defined.

- $R' \cap S_i = \emptyset$
  In the same way as $R \cap D_i = \emptyset$, it can be proven that $S \cap D_i = \emptyset$.
  It can be proven by contradiction that $R'_i \cap S_i = \emptyset$. Let $x \in R'_i \ \wedge \ x \in S_i$. This implies that $x \in Rec_i$ and that the **case** discriminant has d type. Under these conditions we have $Rec_i \cap S_i \neq \emptyset$ contradicting the well-definedness of inh.
  From the premises of [CASE$_I$] rule we can obtain $R''_i \cap S_i = \emptyset$. In addition, R'''_i only contains variables from D. With the same reasoning as the one seen for $R \cap D_i = \emptyset$ it can be proven that $D \cap S_i = \emptyset$. Hence it holds that $R'''_i \cap S_i = \emptyset$.

Initial calls in [CASE!$_I$] Let $e \equiv$ **case!** x **of** $\overline{C_i \ \overline{x_{ij}}^{n_i} \rightarrow e_i}^{n}$

A $\vdash_{check}$ on the i-th branch is done with the following sets:

$$\begin{aligned} D' &= (D^* \cup Rec_i) \cap N_i \\ R' &= R^* \cup sharerec(x, e) \cup ((R'_i \cup R''_i) - D_i) \\ S' &= (S^* \cup (P_i - Rec_i)) \cap N_i \\ &\quad \textbf{where } (D^*, R^*, S^*, N^*) = \sqcup_{i=1}^{n}(D_i, R_i, S_i, N_i, P_i) \end{aligned}$$

We prove (4):

- $D' \cap R' = \emptyset$
  As we have seen in (1), it holds that $D^* \cap R^* = \emptyset$. Moreover, since $D^* \subseteq L$ and one of the assumptions in [CASE!$_I$] states that $R \cap L = \emptyset$, we prove that $sharerec(x, e)$ is disjoint from D^*. We have also that $R'_i \cap D^* = \emptyset$, since the latter does not contain any element from P_i. The disjointness of R''_i and $D^* \cap N_i$ follows from the definition of R''_i.
  On the other hand, since the sets R^*, $sharerec(x, e)$ and R''_i do not contain variables belonging to P_i, it follows that they are disjoint from $Rec_i \cap N_i$. We have also $R'_i \cap (Rec_i \cap N_i) = \emptyset$ from the definition of R'_i.
- $D' \cap S' = \emptyset$
  It can be easily proven from the fact that $D^* \cap S^* = \emptyset$ and $Rec_i \cap (P_i - Rec_i) = \emptyset$.
- $R' \cap S' = \emptyset$
  The proof is similar to that corresponding to $D' \cap R' = \emptyset$. The only difference lies in the fact that R'_i may not be disjoint from $(P_i - Rec_i)$. However, this is forbidden due to the assumption $R'_i \cap (P_i - Rec_i) = \emptyset$ in [CASE!$_I$] rule.

Property (5) holds trivially, since there is an intersection with N_i in the definition of D' y S' of each **case!** branch. Moreover:

$$\begin{aligned} R^* \subseteq R \subseteq scope(e) &\subseteq scope(e_i) \\ sharerec(x, e) - \{x\} \subseteq scope(e) &\subseteq scope(e_i) && \text{(definition of } sharerec) \\ R'_i &\subseteq scope(e_i) && \text{(definition of } sharerec) \\ R''_i &\subseteq scope(e_i) && \text{(definition of } sharerec) \end{aligned}$$

Hence we have $R' \subseteq scope(e_i)$.

In order to prove (6), let $y \in sharerec(z, e_i)$ where $z \in (D^* \cup Rec_i) \cap N_i$ for some $i \in \{1..n\}$. We have to prove $y \in D' \cup R' \cup D_i$. If $y \in D_i$ then the membership holds trivially. If $y \in (D^* \cup Rec_i) \cap N_i$, then it also holds that $y \in D'$. Thus, let us assume in the following that $y \notin ((D^* \cup Rec_i) \cap N_i) \cup D_i$.

We proceed by case distinction:

- $y \in P_i$
  It holds that $y \in R'_i$ and hence $y \in R'$.
- $y \notin P_i$.
  We have $y \in \mathit{sharerec}(z, e)$ for some $z \in (D^* \cup \mathit{Rec}_i) \cap N_i$. If $z \in (\mathit{Rec}_i \cap N_i)$ then we have that $y \in \mathit{sharerec}(x, e)$. In that case we can prove that $y \in R \subseteq R'$ holds. In the following we shall assume that $z \in (D^* \cap N_i)$. Since the property (3) holds for each e_i, the following membership can be proven from the definition of $\sqcup$:

  $$\bigcup_{z \in D^*} \mathit{sharerec}(z, e) \subseteq D^* \cup R^*$$

  Thus $y \in D^* \cup R^*$. We distinguish cases:
  - If $y \in R^*$ then $y \in R'$.
  - If $y \in D^*$ and $y \in N_i$ then $y \in D'$.
  - If $y \in D^*$ and $y \notin N_i$ we have $y \in R'''_i$ and hence $y \in R'$.

  We prove (7):

- $R' \cap D_i = \emptyset$
  We only have to show that $R^* \cap D_i = \emptyset$. The reasoning of the **case** can be applied here.
- $R' \cap S_i = \emptyset$
  Using a similar reasoning to the one seen in the nondestructive **case**, it can be proven that $R^* \cap S_i = \emptyset$. Moreover, we have $\mathit{sharerec}(x, e) \cap S_i = \emptyset$ because of the assumption $R \cap L = \emptyset$ in [CASE!$_I$] and the fact that $\mathit{sharerec}(x, e)$ does not have variables from P_i. On the other hand the disjointness of R'_i and S_i holds, as the *inh*! rules forces an hypothetic common element of both sets to belong to $P_i - \mathit{Rec}_i$, but one of the assumptions in [CASE!$_I$] establishes that $R'_i \cap (P_i - \mathit{Rec}_i) = \emptyset$. With respect to $R''_i \cap S_i = \emptyset$, it is guaranteed by the well-definedness of the $\sqcup$.

Recursive calls in [LET$_C$] Let $e \equiv \textbf{let } x_1 = e_1 \textbf{ in } e_2$

Let us assume that the call $(D_p, R_p, S_p) \vdash_{\mathit{check}} e$ satisfies the properties (4), (5), (6) and (7). There are two $\vdash_{\mathit{check}}$ calls. The first one is done on e_1 with the following sets:

$$\begin{aligned} D' &= D_p \cap N_1 \\ R' &= R_p \\ S' &= S_p \cap N_1 \end{aligned}$$

From the fact that D_p, R_p and S_p are pairwise disjoint it can be seen that $D_p \cap N_1$, R_p and $S_p \cap N_1$ are pairwise disjoint as well. Thus, property (4) holds.

It also holds that (5), since in D' and S' there is an intersection with N_1. In addition we have $R' = R_p \subseteq \mathit{scope}(e) \subseteq \mathit{scope}(e_1)$.

With respect to (6), let $y \in \mathit{sharerec}(z, e_1)$ for some $z \in D_p \cap N_1$. It has to be proven:

$$y \in (D_p \cap N_1) \cup R_p \cup D_1 = D' \cup R' \cup D_1$$

Obviously it holds that $y \neq x_1$, since $x_1 \notin \mathit{scope}(e_1)$. Moreover, since $\mathit{scope}(e_1) = \mathit{scope}(e)$, we have $y \in \mathit{sharerec}(z, e)$ for some $z \in D_p$, and from the induction hypothesis:

$$\bigcup_{z \in D_p} \mathit{sharerec}(z, e) \subseteq D_p \cup R_p \cup ((D_1 \cup D_2) - \{x_1\})$$

Hence, $y \in D_p \cup R_p \cup ((D_1 \cup D_2) - \{x_1\})$. We proceed by case distinction:

- $y \in D_p$
  By property (5) it holds that $y \in N \subseteq N_1 \cup N_2$.
  - If $y \in N_1$ then $y \in D_p \cap N_1$ and hence $y \in D'$.
  - If we had $y \in N_2$, it would hold that $y \in fv(e_2)$ and $y \in R''_p$, which the assumption $((D_p \cap N_1) \cup R_p \cup R''_p) \cap fv(e_2) = \emptyset$ in [LET$_C$] forbids.
- $y \in R_p$
  In that case we have $y \in R'$.
- $y \in (D_1 \cup D_2) - \{x_1\}$
  We have seen that $y \neq x_1$ and hence $y \in D_1 \cup D_2$. If $y \in D_1$ the property holds trivially. If $y \in D_2$ then we would have $y \in fv(e_2)$ and $y \in R''_p$, which is not possible because of the assumption $((D_p \cap N_1) \cup R_p \cup R''_p) \cap fv(e_2) = \emptyset$ in [LET$_C$].

In order to prove (7) it holds that $R_p \cap D_1 = \emptyset$, since $D_1 \subseteq D$, which is disjoint from R_p by induction hypothesis. In addition $R_p \cap S_1 = \emptyset$ because of the assumptions of [LET$_C$].

Let us consider the recursive call to $\vdash_{check}$ on e_2:

$$\begin{array}{l} D' = D_p \cap N_2 \\ R' = R_p \cup (R'_p - D_2) \\ S' = S_p \cap N_2 \end{array}$$

To prove (4) the reasoning is similar to the one seen previously in the $\vdash_{check}$ call on e_1. Two additional properties have to be proven: $R'_p \cap D' = \emptyset$ and $R'_p \cap S' = \emptyset$. They can be obtained from the fact that R'_p does not contain variables from N_2 and that the sets D' and S' only have variables from N_2.

In the same way as the $\vdash_{check}$ call on e_1, the property (5) holds.

In order to prove (6) let us assume $y \in \mathit{sharerec}(z, e_2)$ for some $z \in D_p \cap N_2$. We have to prove:

$$y \in (D_p \cap N_2) \cup (R_p \cup (R'_p - D_2)) \cup D_2 = D' \cup R' \cup D_2$$

If $y = x_1$, then from one of the premises in [LET$_C$] we have $x \in D_2$. Now we shall assume that $y \neq x_1$. In this case it holds that $y \in \mathit{sharerec}(z, e)$ and hence the induction hypothesis can be applied in the same way as the $\vdash_{check}$ call on e_1. It follows that $y \in D_p \cup R_p \cup ((D_1 \cup D_2) - \{x_1\})$. We distinguish cases:

- $y \in D_p$
  - If $y \in N_2$ then $y \in D'$.
  - If $y \notin N_2$ then we have $y \in N_1$, since $D_p \subseteq N \subseteq N_1 \cup N_2$. In this case $y \in R'_p$ and hence $y \in D_2 \ \vee \ y \in R'$
- $y \in R_p$
  It holds that $y \in R'$.
- $y \in (D_1 \cup D_2) - \{x_1\}$
  If $y \in D_2$ then the property holds. If $y \in D_1 - D_2$ then y cannot occur in $fv(e_2)$, by the assumption in [LET$_I$] and hence $y \notin N_2$. Therefore we have $y \in R'_p$ and hence $y \in R'$.

We shall prove now (7):

- $(R_p \cup (R'_p - D_2)) \cap D_2 = \emptyset$
  We have that $R_p \cap D_2 = \emptyset$ since $D_2 \subseteq D$ and, by induction hypothesis, $R_p \cap D = \emptyset$.
- $(R_p \cup (R'_p - D_2)) \cap S_2 = \emptyset$
  It follows that $R_p \cap S_2 = \emptyset$ because of the property $S_2 \subseteq fv(e_2)$ and the assumption $R_p \cap fv(e_2) = \emptyset$ in [LET$_C$]. We prove $R'_p \cap S_2 = \emptyset$ by contradiction: Let us assume that $x \in R'_p \ \wedge \ x \in S_2$. Then $x \in fv(e_2)$ by (2). Moreover, by definition of R'_p, we have either $x \in (D_p \cap N_1)$, or $x \in D_1$. The former case contradicts the assumption $(D_p \cap N_1) \cap fv(e_2) = \emptyset$ in [LET$_C$]. The latter contradicts the assumption $D_1 \cap fv(e_2) = \emptyset$ in [LET$_I$].

Recursive calls in [CASE$_C$] Let $e \equiv \textbf{case}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n}$

Let us assume that the call $(D_p, R_p, S_p) \vdash_{check} e$ satisfies the four properties. Each i-th recursive call is done with the following sets:

$$\begin{array}{l} D' = (D_p \cup D_{p_i}) \cap N_i \\ R' = R_p \cup (R_{p_i} \cup R'_{p_i} \cup R''_{p_i}) - D_i \\ S' = (S_p \cup S_{p_i}) \cap N_i \end{array}$$

From the fact that D_p and S_p are disjoint and that $D_{p_i} = \emptyset$, it follows that D' and S' are disjoint as well. In order to prove (4) we only have to show that R' is disjoint from D' and S'. We shall prove that $R' \cap S' = \emptyset$ (the property $R' \cap D' = \emptyset$ holds in a similar way). By induction hypothesis we have $R_p \cap S_p = \emptyset$. It also holds that $(R_{p_i} \cup R'_{p_i}) \cap S_p = \emptyset$, since S_p only contains free variables from e and $R_{p_i} \cup R'_{p_i}$ only has variables from P_i. By case distinction on the set where the **case** discriminant belongs to, it can be proven that $R_{p_i} \cap S_{p_i} = \emptyset$ in any case. Moreover $R'_{p_i} \cap S_{p_i} = \emptyset$ holds by the assumption in [CASE$_C$]. Finally, the property $R''_{p_i} \cap S' = \emptyset$ also holds: on one hand, S_{p_i} only contains elements from P_i, while R''_{p_i} has no pattern variables. On the other hand, $S_p \subseteq N$ and hence S_p is disjoint from D, while also remaining disjoint from D_p by induction hypothesis.

Gathering all these equalities together we have:

$$(R_p \cup (R_{p_i} \cup R'_{p_i} \cup R''_{p_i}) - D_i) \cap ((S_p \cup S_{p_i}) \cap N_i) = \emptyset$$

Hence R' and S' are disjoint.

The proof for (5) is quite simple: It follows that $D' \cup S' \subseteq N_i$ from the fact that in the definitions of D' and S' an intersection is made with the corresponding N_i. On the other hand we have $R_p \subseteq \mathit{scope}(e) \subseteq \mathit{scope}(e_i)$. R_{p_i} only has variables from Rec_i, which also belong to $\mathit{scope}(e_i)$. In addition it holds that $R'_{p_i} \cup R''_{p_i} \subseteq \mathit{scope}(e_i)$ since both R'_{p_i} and R''_{p_i} only contain variables from $\mathit{sharerec}(e_i)$.

In order to prove (6) let us assume $y \in \mathit{sharerec}(z, e_i)$ for some variable $z \in (D_p \cup D_{p_i}) \cap N_i$. Since $D_{p_i} = \emptyset$, it holds that $z \in D_p$. We have to prove:

$$y \in (((D_p \cup D_{p_i}) \cap N_i) \cup (R_p \cup (R_{p_i} \cup R'_{p_i} \cup R''_{p_i}) - D_i) \cup D_i = D' \cup R' \cup D_i$$

If $y \in P_i$ then we have $y \in R'_{p_i}$ and hence it holds that $y \in D_i \cup R'$.

If $y \notin P_i$ then $y \in \mathit{sharerec}(z, e)$ for some $z \in D_p$. From the induction hypothesis we have:

$$\bigcup_{z \in D_p} \mathit{sharerec}(z, e) \subseteq D_p \cup R_p \cup D$$

Hence $y \in D_p \cup R_p \cup D$. We proceed by case distinction:

- $y \in D_p$
  If $y \in N_i$ then $y \in D'$. If $y \notin N_i$ then we have $y \in R''_{p_i}$, from which it follows that $y \in D_i \cup R'$.
- $y \in R_p$
  In this case $y \in D_i \cup R'$ always holds.
- $y \in D$
  If $y \notin (D_p \cap N_i)$ then $y \in R''_{p_i}$ and hence $y \in R' \ \vee \ y \in D_i$. Otherwise it belongs to D'.

Now we shall prove (7). The equality $R' \cap D_i = \emptyset$ follows from the definition of R' and the disjointness of R_p and $D \supseteq (D_i - P_i)$. Let us prove $R' \cap S_i = \emptyset$. From the induction hypothesis we can show that $R_p \cap S_i = \emptyset$. We have $R'_{p_i} \cap S_i = \emptyset$ by hypothesis in [CASE$_C$] rule. On the other hand, let z be a variable such that $z \in R_{p_i}$ and $z \in S_i$. The first membership forces x to belong to D_p, where x is the **case** discriminant. Furthermore we have $z \in \mathit{Rec}_i \cap S_i$, which contradicts the well-definedness of *inh* in the [CASE$_C$] rule. Hence we have $R_{p_i} \cap S_i = \emptyset$. Finally it holds that $R''_{p_i} \cap S_i = \emptyset$, since R''_{p_i} only has variables from D and N and it follows that $D \cap S_i = N \cap S_i = \emptyset$ by the definition of $\sqcup$.

Recursive call in [CASE!$_C$] Let $e \equiv \textbf{case!}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n}$

In this case the proof is a simplified version of the one seen in the recursive call in [CASE$_C$]. Thus it will not be described here.

Lemma 2. *Let $f\ \overline{x_i}^n\ @\ \overline{r_j}^l = e$ be a function declaration. If the algorithm eventually succeeds with $(D, \emptyset, S, \emptyset)$ for e, including the fixpoint computation and the final $\vdash_{check}$ forcing all don't-know variables to be safe, then for all $x \in \mathit{var}(e)$, x has got a mark d, s or r.*

Proof. By induction on the depth in which x comes into scope:

- **Base case:** If $x \in \mathit{scope}(e)$ then x is one of the parameters x_i, that is, one of the free variables in e. Every variable belonging to N gets an s mark, due to the final $\vdash_{check}$ done:
  $$(\emptyset, \emptyset, N) \vdash_{check} e$$
- **Inductive step:** If $x \in \mathit{var}(e)$ but $x \notin \mathit{scope}(e)$ then x is a bound variable. If x comes into scope in a expression **let** $x_1 = e_1$ **in** e_2 and it is inferred with an n mark (that is, $x_1 \in N_2$), the following $\vdash_{check}$ of [LET$_I$] rule forces it to get an s mark:
  $$(\emptyset, \emptyset, (S_1 \cup \{x_1\}) \cap N_2) \vdash_{check} e_2$$
  If x comes into scope as a pattern variable of a branch of a **case!** x' **of** $\overline{C\ \overline{x_{ij}}^{n_i} \to e_i}^{n}$ then it is one of the x_{ij} for some $i \in \{1..n\}$ and some $j \in \{1..n_i\}$. In that case the following $\vdash_{check}$ on e_i is done:
  $$((D \cup \mathit{Rec}_i) \cap N_i, R \cup R' \cup (R'_i \cup R''_i) - D_i, (S \cup (P_i - \mathit{Rec}_i)) \cap N_i) \vdash_{check} e_i$$

It holds that if $x = x_{ij}$ for some $i \in \{1..n\}$ and some $j \in \{1..n_i\}$, then either $x \in Rec_i$ or $x \in P_i - Rec_i$. In the former case x gets a d type and in the latter case it gets an s type.
If x comes into scope in a **case** x' **of** $\overline{C\ \overline{x_{ij}}^{n_i} \to e_i}^{n}$, then x is one of the x_{ij} of some **case** branch e_i. Since x' comes into scope in a wider context, from the induction hypothesis it follows that it already has a mark d, r or s. If x' has an s mark, then the following *check* on e_i is done:

$$(D \cap N_i, R \cup ((R'_i \cup R''_i \cup R'''_i) - D_i), (S \cup P_i) \cap N_i) \vdash_{check} e_i$$

and hence x gets an s mark. If x' has a d mark, the situation is similar to the one in **case**!: if x is in a condemned position of the corresponding C_i then it gets type r, otherwise it gets s type. Finally, if x' has an r mark, then x gets an s mark.

A.2 Algorithm correctness

Let $f\ \overline{x_i}^n\ @\ \overline{r_j}^l = e$ a function definition accepted by the inference algorithm. For each subexpression e' of e four sets have been inferred by means of $\vdash_{inf}$ rules. This has been done *once* for each subexpression.

$$e' \vdash_{inf} (D, R, S, N)$$

Furthermore, an expression can suffer several $\vdash_{check}$, each with a different set of variables:

$$\begin{array}{c} (D_1, R_1, S_1) \vdash_{check} e' \\ \vdots \\ (D_m, R_m, S_m) \vdash_{check} e' \end{array}$$

We shall use the notation $(D', R', S') \vdash^*_{check} e$ denote the accumulation of all the $\vdash_{check}$ suffered by e during the algorithm and let D', R' and S' represent the union of respectively all the marks d, r and s forced in these calls to $\vdash_{check}$. That is:

$$(D', R', S') \vdash^*_{check} e \textbf{ where } D' = \bigcup_{i=1}^{m} D_i \qquad R' = \bigcup_{i=1}^{m} R_i \qquad S' = \bigcup_{i=1}^{m} S_i$$

By Lemma 2 we can safely assume that for all subexpression e' of e it holds that $N \subseteq D' \cup R' \cup S'$. The following theorem will prove the correctness of the algorithm for nonrecursive functions by establishing a correspondence between the $\vdash_{inf}$ and $\vdash_{check}$ rules of the algorithm and the rules of the type system. We shall extend it later (to which corresponds to Theorem 1 in the paper) in order to include recursive definitions

Theorem 2. *Let us assume that the* nonrecursive *function declaration* $f\ \overline{x_i}^n\ @\ \overline{r_j}^l = e$ *has been successfully typed by the inference algorithm and let* e' *be any subexpression of* e *for which the algorithm has got* $e' \vdash_{inf} (D, R, S, N)$ *and* $(D', R', S') \vdash^*_{check} e'$*. Then there exists a safe type* s' *and a well-formed type environment* Γ *such that:*

1. $\Gamma \vdash e' : s$

2. $\forall x \in scope(e') :$
   (a) $\Gamma(x) = d \Leftrightarrow x \in D \cup D'$
   (b) $\Gamma(x) = s \Leftrightarrow x \in S \cup S'$
   (c) $\Gamma(x) = r \Leftrightarrow x \in R \cup R'$

Proof. By structural induction on e'.

$\boxed{e' \equiv c}$

$$\frac{}{\emptyset \vdash c : B}\ [\text{LIT}] \qquad \frac{}{c \vdash_{inf} (\emptyset, \emptyset, \emptyset, \emptyset)}\ [\text{LIT}_I] \qquad \frac{}{(\emptyset, R, \emptyset) \vdash_{check} c}\ [\text{LIT}_C]$$

Let $\Gamma = [y : r \mid y \in R']$. The environment $\emptyset$ satisfies $\emptyset \vdash e' : s$. By means of the [EXTS] rule $\Gamma \vdash e' : s$ can be inferred, and hence (1) holds. On the other hand property (2) also holds, since $D \cup D' = S \cup S' = \emptyset$ and $R \cup R' = R'$.

$\boxed{e' \equiv x}$

$$\frac{}{[x : s] \vdash x : s}\ [\text{VAR}] \qquad \frac{}{x \vdash_{inf} (\emptyset, \emptyset, \{x\}, \emptyset)}\ [\text{VAR}_I] \qquad \frac{}{(\emptyset, R, \emptyset) \vdash_{check} x}\ [\text{VAR}_C]$$

Let $\Gamma = [x : s] + [y : r \mid y \in R']$, which is well-defined, since $S = \{x\}$ and by Lemma 1 (7), the sets S and R' are disjoint. The environment $[x : s]$ satisfies $[x : s] \vdash e' : s$. By means of the [EXTS] rule it can be proven that $\Gamma \vdash e' : s$. The property (2) also holds in this case, since $D \cup D' = \emptyset$, $S \cup S' = \{x\}$ and $R \cup R' = R'$.

$\boxed{e' \equiv x!}$

$$\frac{\begin{array}{c} R = sharerec(x, x!) - \{x\} \\ \Gamma_R = \{y : danger(type(y)) \mid y \in R\} \end{array}}{\Gamma_R + [x : T!@\rho] \vdash x! : T@\rho}\ [\text{REUSE}] \qquad \frac{}{(\emptyset, R, \emptyset) \vdash_{check} x!}\ [\text{REUSE}_C]$$

$$\frac{R = sharerec(x, x!) - \{x\} \quad type(x) = T@\ \rho}{x! \vdash_{inf} (\{x\}, R, \emptyset, \emptyset)}\ [\text{REUSE}_I]$$

Let $\Gamma = [x : d] + [y : r \mid y \in R \cup R']$. It is well-defined, since $D = \{x\}$ and by Lemma 1 (1,7) it follows that $R \cap D = \emptyset$ and $R' \cap D = \emptyset$, since $D = \{x\}$.

Let $\Gamma' = [x : d] + [y : r \mid y \in R]$. Since $R = sharerec(x, e') - \{x\}$ we have $\Gamma' \vdash e' : s$. Through successive applications of [EXTS] rule it holds that $\Gamma \vdash e' : s$.

Property (2) holds trivially, since $D \cup D' = \{x\}$ and $S \cup S' = \emptyset$.

$\boxed{e' \equiv x@r}$

$$\frac{\Gamma_1 \geq_{x@r} [x : T@\rho', r : \rho]}{\Gamma_1 \vdash x@r : T\ @\rho}\ [\text{COPY}] \qquad \frac{}{x@r \vdash_{inf} (\emptyset, \emptyset, \emptyset, \{x\})}\ [\text{COPY}_I]$$

We have $x \in N$, and since $N \subseteq D' \cup R' \cup S'$ we proceed by case distinction:

- $x \in D'$
  A $\vdash_{check}$ has been done on this expression using [COPY1$_C$]. Let $\Gamma = [x : d] + [y : r \mid y \in R'] + [r : \rho]$, which is well-defined by Lemma 1 (7). It can be proven (1) from the fact that $\Gamma \geq_{e'} [x : s, r : \rho]$. Indeed, the only variable with a d type in Γ is x, which belongs to D'. From Lemma 1 (6) it follows that every variable belonging to $sharerec(x, e')$ also belongs to $D' \cup R' \cup D = \{x\} \cup R'$. Since the variables from $\{x\} \cup R'$ have unsafe types in Γ, one of the conditions imposed by $\geq_{e'}$ holds. The two remaining conditions hold trivially and hence we have $\Gamma \vdash e' : s$.
  In addition, we have $D \cup D' = \{x\}$, $R \cup R' = R'$ and $S \cup S' = \emptyset$, so that property (2) also holds.
- $x \in R'$
  In this case the $\vdash_{check}$ has been done via the rule [COPY2$_C$]. Let $\Gamma = [y : r \mid y \in R'] + [r : \rho]$. It holds that $\Gamma \geq_{e'} [x : s, r : \rho]$, since $x \in dom(\Gamma)$ and $\Gamma(x) = r \geq s$. Moreover the third property of $\geq_{e'}$ holds, since there are no variables with d type in Γ. Hence $\Gamma \vdash e' : s$ can be inferred.
  Property (2) can be proven from the fact that that $D \cup D' = S \cup S' = \emptyset$ and that in Γ there are no variables whose type is s or d. In addition $R \cup R' = R'$ and the variables from Γ with r type are exactly the ones belonging to R'.
- $x \in S'$
  Let $\Gamma = [x : s] + [y : r \mid y \in R'] + [r : \rho]$. It is well-defined because of Lemma 1 (7). The corresponding $\vdash_{check}$ rule is [COPY3$_C$]. Again, it holds that $\Gamma \geq_{e'} [x : s, r : \rho]$, since $\Gamma(x) = s \geq s$. Hence, $\Gamma \vdash e' : s$.
  Moreover the property (2) also holds, since $D \cup D' = \emptyset$, $R \cup R' = R'$ and $S \cup S' = \{x\}$.

$\boxed{e' \equiv C\ \overline{a_i}^n\ @r}$

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

$$\frac{\forall i \in \{1..n\}. a_i \vdash_{inf} (\emptyset, \emptyset, S_i, \emptyset)}{C\overline{a_i}^n @r \vdash_{inf} (\emptyset, \emptyset, \bigcup_{i=1}^n S_i, \emptyset)}\ [\text{CONS}_I] \qquad \frac{}{(\emptyset, R, \emptyset) \vdash_{check} C\ \overline{a_i}^n\ @r}\ [\text{CONS}_C]$$

Let $\Gamma = \bigoplus_{i=1}^n [a_i : s] + [y : r \mid y \in R'] + [r : \rho]$. The result of $\bigoplus$ operator is well-defined, since it is only applied to environments with safe variables. Furthermore, $S = \{a_i \mid var(a_i) \wedge i \in \{1..n\}\}$ and from Lemma 1 (7) it follows that $R' \cap S = \emptyset$. Hence Γ is well defined.

On the other hand, let $\Gamma' = \bigoplus_{i=1}^n [a_i : s] + [r : \rho]$. Trivially it holds that $\Gamma' \vdash e' : s$. Using the [EXTS] rule of the type system this environment can be extended with variables from R' in order to prove $\Gamma \vdash e' : s$.

Moreover $R \cup R' = R'$ and $D \cup D' = \emptyset$ and hence properties (2a) and (2c) hold. In addition, for any variable x:

$$\Gamma(x) = s \Leftrightarrow x \in var(e') \Leftrightarrow x \in S \cup S'$$

Thus property (2b) also holds.

$\boxed{e' \equiv g\ \overline{a_i}^n\ @\ \overline{r_j}^l}$

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

$$\frac{\begin{array}{c}\forall i \in \{1..n\}.D_i = \{a_i \mid i \in I_D\} \quad (\bigcup_{i=1}^{n} D_i) \cap (\bigcup_{i=1}^{n} S_i) = \emptyset \qquad R \cap (\bigcup_{i=1}^{n} S_i) = \emptyset \\ \forall i \in \{1..n\}.S_i = \{a_i \mid i \in I_S\} \quad (\bigcup_{i=1}^{n} D_i) \cap (\bigcup_{i=1}^{n} N_i) = \emptyset \qquad R \cap (\bigcup_{i=1}^{n} D_i) = \emptyset \\ \forall i \in \{1..n\}.N_i = \{a_i \mid i \in I_N\} \quad \forall i,j \in \{1..n\}\,.\, i \neq j \Rightarrow D_i \cap D_j = \emptyset \quad R \cap (\bigcup_{i=1}^{n} N_i) = \emptyset \\ \Sigma \vdash g : (I_D, \emptyset, I_S, I_N) \qquad R = \bigcup_{i=1}^{n} \left\{\mathit{sharerec}(a_i, g\ \overline{a_i}^n\ @\ \overline{r_j}^l) - \{a_i\} \mid a_i \in D_i\right\}\end{array}}{g\ \overline{a_i}^n\ @\ \overline{r_j}^l \vdash_{\mathit{inf}} (\bigcup_{i=1}^{n} D_i, R, \bigcup_{i=1}^{n} S_i, (\bigcup_{i=1}^{n} N_i) - (\bigcup_{i=1}^{n} S_i))}\ [\mathrm{APP}_I]$$

$$\frac{g\ \overline{a_i}^n\ @\ \overline{r_j}^l \vdash_{\mathit{inf}} (D, R, S, N) \qquad \forall a_i \in D_p\,.\,(\#j : 1 \leq j \leq n : a_i = a_j) = 1}{(D_p, R_p, S_p) \vdash_{\mathit{check}} g\ \overline{a_i}^n\ @\ \overline{r_j}^l}\ [\mathrm{APP}_C]$$

Let us assume that f is a nonrecursive call, that is, $f \neq g$. This allows us to establish $N = \emptyset$ and hence $D' = S' = \emptyset$.

Let $(I_D, \emptyset, I_S, \emptyset)$ the signature of the symbol g. For each $i \in \{1..n\}$, Γ_i is defined as follows:

$$\Gamma_i = \begin{cases} [a_i : d] \text{ if } a_i \text{ is a variable and } i \in I_D \\ [a_i : s] \text{ if } a_i \text{ is a variable and } i \in I_S \end{cases} \qquad (i > 0)$$

We define Γ' as follows:

$$\Gamma' = [\overline{r_j : \rho_j}^l, g : \sigma] + \bigoplus_{i=0}^{n} \Gamma_i$$

We shall prove by contradiction that Γ is well-defined. If it were not well-defined, it would fall into at least one of the following cases:

1. There are two environments Γ_i and Γ_j, (with $i, j \in \{1..n\}$, $i \neq j$) such that $x \in \mathit{dom}(\Gamma_i) \cap \mathit{dom}(\Gamma_j)$ and $\Gamma_i(x) = \Gamma_j(x) = d$. Hence we have $i, j \in I_D$ and by hypothesis in $[\mathrm{APP}_I]$ rule, $x \in D_i$ and $x \in D_j$. This would lead to $D_i \cap D_j \neq \emptyset$, which contradicts one of the assumptions in rule $[\mathrm{APP}_I]$.
2. There are two environments Γ_i and Γ_j, (with $i, j \in \{1..n\}$, $i \neq j$) such that $x \in \mathit{dom}(\Gamma_i) \cap \mathit{dom}(\Gamma_j)$, $\Gamma_i(x) = d$ and $\Gamma_j(x) = s$. In this case we have $x \in D_i$ and $x \in S_j$, which contradicts the assumption $(\bigcup_{i=1}^{n} D_i) \cap (\bigcup_{i=1}^{n} S_i) = \emptyset$.
3. There are two environments Γ_i and Γ_j, (with $i, j \in \{1..n\}$, $i \neq j$) such that $x \in \mathit{dom}(\Gamma_i) \cap \mathit{dom}(\Gamma_j)$, $\Gamma_i(x) = s$ and $\Gamma_j(x) = d$. This case is similar to the previous one.

On the other hand, we define two environments Γ_R and $\Gamma_{R'}$ as follows:

$$\begin{aligned} \Gamma_R &= [y : r \mid y \in \mathit{sharerec}(a_i, e') - \{a_i\}, i \in \{1..n\}, \mathit{cdm?}(t_i)] \\ \Gamma_{R'} &= [y : r \mid y \in R'] \end{aligned}$$

and let $\Gamma'' = \Gamma_R + \Gamma'$. There are no common variables in the domains of Γ_R and Γ' and hence Γ'' is well-defined. Indeed, the definition of Γ_R does not include the a_i variables and the identifiers g and r which belong to the domain of Γ'.

From the way in which the Γ'' environment has been built, similar to the rule of the type system [APP], it follows that $\Gamma'' \vdash e' : s$. We now define

$\Gamma = \Gamma_{R'} \otimes \Gamma''$. It is well-defined by Lemma 1. From the fact that $\Gamma'' \vdash e' : s$ and via the [EXTS] rule we can infer $\Gamma \vdash e' : s$ and hence property (1) holds. In the same way, property (2) is quite straightforward to prove:

$$\begin{aligned}\Gamma(x) = d &\Leftrightarrow \exists i \in \{1..n\}.\ x \in dom(\Gamma_i)\ \wedge\ \Gamma_i(x) = d \\ &\Leftrightarrow \exists i \in I_D.\ x = a_i \\ &\Leftrightarrow \exists i \in \{1..n\}\,.\, x \in D_i \\ &\Leftrightarrow x \in D\end{aligned}$$

$$\begin{aligned}\Gamma(x) = s &\Leftrightarrow \exists i \in \{1..n\}.\ x \in dom(\Gamma_i)\ \wedge\ \Gamma_i(x) = s \\ &\Leftrightarrow \exists i \in I_S.\ x = a_i \\ &\Leftrightarrow \exists i \in \{1..n\}\,.\, x \in S_i \\ &\Leftrightarrow x \in S\end{aligned}$$

$$\begin{aligned}\Gamma(x) = r &\Leftrightarrow x \in dom(\Gamma_R)\ \vee\ x \in dom(\Gamma_{R'}) \\ &\Leftrightarrow (\exists i \in \{1..n\}.\ x \in sharerec(a_i, e') - \{a_i\}\ \wedge\ a_i \in D_i)\ \vee\ x \in R' \\ &\Leftrightarrow x \in R \cup R'\end{aligned}$$

$\boxed{e' \equiv \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2}$

Let us assume that $x_1 \notin D_2$, which corresponds to the use of rule [LET1] of the type system.

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

$$\frac{\begin{array}{c} e_1 \vdash_{inf} (D_1, R_1, S_1, N_1) \qquad x_1 \notin R_2 \qquad (D_1 \cup R_1) \cap fv(e_2) = \emptyset \\ e_2 \vdash_{inf} (D_2, R_2, S_2, N_2) \qquad N = (N_1 - (D_2 \cup R_2 \cup S_2)) \cup N_2 \\ (\emptyset, \emptyset, N_1 \cap (D_2 \cup R_2 \cup S_2)) \vdash_{check} e_1 \quad (\emptyset, \emptyset, (S_1 \cup \{x_1\}) \cap N_2) \vdash_{check} e_2 \end{array}}{\textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2 \vdash_{inf} ((D_1 \cup D_2) - \{x_1\}, R_1 \cup (R_2 - D_1), ((S_1 - N_2) \cup S_2) - (\{x_1\} \cup D_2 \cup R_2), N - \{x_1\})}\ [\text{LET}_I]$$

$$\frac{\begin{array}{c} e_1 \vdash_{inf} (D_1, R_1, S_1, N_1)\ R_p \cap S_1 = \emptyset\ \wedge\ ((D_p \cap N_1) \cup R_p \cup R''_p) \cap fv(e_2) = \emptyset \\ e_2 \vdash_{inf} (D_2, R_2, S_2, N_2) \quad \exists z \in D_p \cap N_2\,.\, x_1 \in sharerec(z, e_2) \Rightarrow x_1 \in D_2 \\ (D_p \cap N_1, R_p, S_p \cap N_1) \vdash_{check} e_1 \qquad (D_p \cap N_2, R_p \cup (R'_p - D_2), S_p \cap N_2) \vdash_{check} e_2 \\ \textbf{where}\ R'_p = \{y \in ((D_p \cap N_1) \cup D_1) \cap sharerec(z, e_2) \mid z \in D_p \cap N_2\} - N_2 \\ R''_p = \{y \in sharerec(z, e_1) \mid z \in D_p \cap N_1\} \end{array}}{(D_p, R_p, S_p) \vdash_{check} \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2}\ [\text{LET}_C]$$

We have:

$$\begin{array}{ll} e_1 \vdash_{inf} (D_1, R_1, S_1, N_1) & e_2 \vdash_{inf} (D_2, R_2, S_2, N_2) \\ (D'_1, R'_1, S'_1) \vdash^*_{check} e_1 & (D'_2, R'_2, S'_2) \vdash^*_{check} e_2 \end{array}$$

By induction hypothesis there are two environments Γ_1 and Γ_2 that satisfy:

$$\Gamma_1 \vdash e_1 : s_1 \qquad \Gamma_2 \vdash e_2 : s_2$$

Let $\Gamma'_2 = \Gamma_2 \upharpoonright (dom(\Gamma_2) - \{x_1\})$ and $\Gamma = \Gamma_1 \rhd^L \Gamma'_2$, where $L = fv(e_2)$. The environment Γ is well-defined if $\forall x \in dom(\Gamma_1).\ unsafe?(\Gamma_1(x)) \Rightarrow x \notin L$ holds. We shall show this: Let x a variable from the domain of Γ_1 such that $x \in L$. The assumption $(D_1 \cup R_1) \cap fv(e_2)$ in [LET$_I$] prevents x from occurring in $D_1 \cup R_1$. Moreover, the assumption $((D_p \cap N_1) \cup R_p \cup R''_p) \cap fv(e_2) = \emptyset$ in [LET$_C$] prevents x from occurring in $D'_1 \cup R'_1$. From the fact that $x \notin D_1 \cup D'_1$ and by induction hypothesis we have $\Gamma_1(x) \neq d$. Similarly, from the fact that $x \notin R_1 \cup R'_1$ and by induction hypothesis we have $\Gamma_1(x) \neq r$. Gathering these two results together, we have $\neg unsafe?(\Gamma_1(x))$.

Thus Γ is well-defined. Now it has to be proven that $\Gamma_2' + [x_1 : s_1] \vdash e_2 : s$. If $x_1 \notin dom(\Gamma_2)$ then $\Gamma_2' = \Gamma_2$. In this case we obtain $\Gamma_2' \vdash e_2 : s$ and it is only necessary the [EXTS] rule in order to get $\Gamma_2' + [x_1 : s_1] \vdash e_2 : s$. If $x_1 \in dom(\Gamma_2')$ we have to prove that $\Gamma_2(x_1) = s$. We proceed by contradiction: Let us assume that $\Gamma_2(x_1) = d$. It follows that $x_1 \in D_2 \cup D_2'$. Since we assume that $x_1 \notin D_2$ ([LET1] in the type system), we have $x_1 \in D_2'$. That implies $x_1 \in N_2$ (by Lemma 1 (5)). Hence the $\vdash_{check}$ on e_2 in [LET$_I$] rule forces x_1 to belong to S_2'. This contradicts the property $D_2' \cap S_2' = \emptyset$ (Lemma 1 (4)). Now we shall assume that $\Gamma_2(x_1) = r$: this would imply $x_1 \in R_2 \cup R_2'$. By the assumption in [LET$_I$] we establish that $x_1 \notin R_2$, and by simple inspection of the $\vdash_{check}$ calls in [LET$_I$] and [LET$_C$], it can be proven that $x_1 \notin R_2'$, so $\Gamma_2(x_1) \neq r$. The remaining case is that $\Gamma_2(x_1) = s$.

We have got $\Gamma \vdash e' : s$ and hence the property (1) holds. We shall now prove the property (2a) considering each implication separately. Let $x \in scope(e')$. This implies that $x \neq x_1$:

- $x \in D \cup D' \Rightarrow \Gamma(x) = d$
  If $x \in D$, then by definition of D we have $x \in D_1 \ \vee \ x \in D_2$.
  If $x \in D'$, then $x \in N$. From the definition of N it follows that $x \in N_1 \vee x \in N_2$ and hence $x \in D_1' \ \vee \ x \in D_2'$.
  If $x \in D_1 \cup D_1'$ we have $\Gamma_1(x) = d$ by induction hypothesis and hence $\Gamma(x) = (\Gamma_1 \rhd^L \Gamma_2)(x) = \Gamma_1(x) = d$
  On the other hand, if $x \in D_2 \cap D_2'$ we have $\Gamma_2(x) = d$ by induction hypothesis and hence $\Gamma(x) = (\Gamma_1 \rhd^L \Gamma_2)(x) = \Gamma_2(x) = d$ (since it is not possible $unsafe?(\Gamma_1(x))$, as $\Gamma_1 \rhd^L \Gamma_2$ would not be defined).
- $\Gamma(x) = d \Rightarrow x \in D \cup D'$
  If $\Gamma(x) = \Gamma_1(x) = d$ then $x \in D_1 \cup D_1'$ and hence $x \in D \cup D'$.
  If $\Gamma(x) = \Gamma_2(x) = d$ then $x \in D_2 \cup D_2'$ and hence $x \in D \cup D'$.

We have completed the proof for (2a). We shall prove now (2b):

- $x \in S \cup S' \Rightarrow \Gamma(x) = s$
  First we assume that $x \in S$. We have $x \in (S_1 - N_2) \cup S_2$ and $x \notin D_2 \cup R_2$. If $x \in S_1$ but $x \notin N_2$ from the induction hypothesis it follows that $\Gamma_1(x) = s$. In order to prove $(\Gamma_1 \rhd^L \Gamma_2)(x) = \Gamma_1(x) = s$ it is enough to prove that x cannot occur in Γ_2 with an unsafe type. Indeed:
  - If $\Gamma_2(x) = d$ then $x \in D_2 \cup D_2'$, which is not possible, since $x \notin D_2 \cup N_2$.
  - If $\Gamma_2(x) = r$ then we have $x \in R_2 \cup R_2'$. Since $x \notin R_2$, it holds that $x \in R_2'$. The occurrence of x in R_2' has its source in a $\vdash_{check}$ call on e_2 in [LET$_C$]. Thus we have $x \in R' \cup R_p'$. It is not possible that $x \in R'$, since $R' \cap S = \emptyset$ holds by Lemma 1 (7). It is not possible that $x \in R_p'$, since this would imply that $x \in N_1 \cup D_1$, which contradicts Lemma 1 (1).

  If $x \in S_2$ then we have $\Gamma_2(x) = s$ by induction hypothesis. This implies that $x \in fv(e_2)$. Since the Γ environment is well defined, it is not possible under these conditions that x has an unsafe type in Γ_1. Thus $\Gamma(x) = (\Gamma_1 \rhd^L \Gamma_2) = s$.
  Now we shall assume that $x \in S'$. By Lemma 1 (5) we have $x \in N \subseteq N_1 \cup N_2$. If $x \in N_1$ then $x \in S_1'$. By induction hypothesis we have $\Gamma_1(x) = s$. To prove $\Gamma(x) = (\Gamma_1 \rhd^L \Gamma_2)(x) = s$ we have to ensure that x does not have an unsafe type in Γ_2:
  - If $\Gamma_2(x) = d$ then $x \in D_2 \cup D_2'$. However, it is not possible that $x \in D_2$ since we would have $x \in D$, which contradicts the disjointness of D and N. It is not possible that $x \in D_2'$, since it would imply that $x \in D'$, contradicting the disjointness of D' and S'.

- If $\Gamma_2(x) = r$ then $x \in R_2 \cup R_2'$. It is not possible that $x \in R_2$ since that would imply $x \in R \cup D$ and the sets D and R are disjoint from N. The case $x \in R_2'$ is not possible: it would lead to $x \in R' \cup R_p'$. If $x \in R'$ then R' and S' would not be disjoint. If $x \in R_p'$ we have $x \in D' \cup D_1 \subseteq D' \cup D$. However, it holds that $D' \cap S' = \emptyset$ and $D \cap N = \emptyset$, which lead to a contradiction.

  If $x \in N_2$ then $x \in S_2'$ and, by the induction hypothesis, $\Gamma_2(x) = s$. Since $x \in fv(e_2)$ and Γ is well-defined, the variable x cannot occur in Γ_1 with an unsafe type. Hence $\Gamma(x) = (\Gamma_1 \rhd^L \Gamma_2)(x) = s$.
- $\Gamma(x) = s \Rightarrow x \in S \cup S'$

  If $\Gamma(x) = \Gamma_1(x) = s$ then $x \in S_1 \cup S_1'$. It can be proven that $x \notin D_2$ and $x \notin R_2$, since $\Gamma_2(x) \neq d, r$. If $x \in S_1'$ then we have $x \in S'$. On the other hand, if $x \in S_1$ a case distinction is made:
  - If $x \notin N_2$ then $x \in S$.
  - If $x \in N_2$ then $x \in N \subseteq D' \cup R' \cup S'$. It is not possible that $x \in D'$, since that would imply $\Gamma(x) = d$. It is not possible that $x \in R'$ since it implies that $\Gamma(x) = r$ (see below). Hence we have $x \in S'$. (call to $\vdash_{check}$ in [LET$_I$]).

  If $\Gamma(x) = \Gamma_2(x) = s$ then $x \in S_2 \cup S_2'$. Moreover it holds that $x \notin D_2$ and $x \notin R_2$, since $\Gamma_2(x) \neq d, r$. Under these conditions, if $x \in S_2$ then $x \in S$. In addition, if S_2' then $x \in S'$.

Finally we shall prove (2c):

- $x \in R \cup R' \Rightarrow \Gamma(x) = r$

  Let $x \in R \subseteq R_1 \cup R_2$. If $x \in R_1$ then $\Gamma_1(x) = r$ and hence, $\Gamma(x) = (\Gamma_1 \rhd^L \Gamma_2)(x) = r$. If $x \in R_2$ then $\Gamma_2(x) = r$. In this case it holds that $\Gamma(x) = r$ only if x does not have d type in Γ_1. However, this is not the case, since Γ is well-defined.

  On the other hand, let $x \in R'$. Hence $x \notin D_1$; otherwise we would have $x \in D$, and a contradiction with $R' \cap D = \emptyset$ (Lemma 1 (7)). Thus we have $x \in R_1'$ and by induction hypothesis, $\Gamma_1(x) = r$. Hence we have $\Gamma(x) = r$.
- $\Gamma(x) = r \Rightarrow x \in R \cup R'$

  If $\Gamma(x) = \Gamma_1(x) = r$ then $x \in R_1 \cup R_1'$ and hence $x \in R \cup R'$.

  If $\Gamma(x) = \Gamma_2(x) = r$ then $\Gamma_1(x) \neq d$, which implies that $x \notin D_1$. On the other hand, the induction hypothesis establishes that $x \in R_2 \cup R_2'$. If $x \in R_2'$ then $x \in R' \cup R_p'$. It is not possible that $x \in R_p'$, as it would imply that $x \in D' \cup D_1 \subseteq D' \cup D$ and hence we would have $\Gamma(x) = d$. Thus $x \in R'$.

For the case $x \in D_2$ ([LET2] in the type system) the reasoning is similar. Now we have to prove the assumption $\Gamma_2' + [x_1 : d_1] \vdash e_2 : s$ of [LET2]. However, since we have $x \in D_2$ it holds that $x \in dom(\Gamma_2)$ and $\Gamma_2(x) = d$. Thus the assumption holds.

$$\boxed{e' \equiv \textbf{case } z \textbf{ of } \overline{C\ \overline{x_{ij}}^{n_i} \to e_i}^{n}}$$

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

$$\frac{\begin{array}{l}\forall i \in \{1..n\} \,.\, e_i \vdash_{inf} (D_i, R_i, S_i, N_i) \qquad\qquad def(\sqcup_{i=1}^{n}(D_i, R_i, S_i, N_i, P_i)) \\ \forall i \in \{1..n\} \,.\, P_i = \bigcup_{j=1}^{n_i}\{x_{ij}\} \qquad\qquad \forall i \in \{1..n\} \,.\, def(inh(type(x), D_i, R_i, S_i, P_i, Rec_i)) \\ \forall i \in \{1..n\} \,.\, Rec_i = \bigcup_{j=1}^{n_i}\{x_{ij} \mid j \in RecPos(C_i)\} \\ type(x) = \begin{cases} d \text{ if } x \in D \\ r \text{ if } x \in R \\ s \text{ if } x \in S \\ n \text{ e. o. c.} \end{cases} \qquad\qquad (D, R, S, N) = \sqcup_{i=1}^{n}(D_i, R_i, S_i, N_i, P_i) \\ \qquad\qquad\qquad\qquad\qquad\qquad N' = \begin{cases} N & \text{if } x \in D \cup R \cup S \\ N \cup \{x\} & \text{if } x \notin D \cup R \cup S \end{cases} \\ \forall i \in \{1..n\} \,.\, ((D \cup D'_i) \cap N_i, R \cup ((R'_i \cup R''_i \cup R'''_i) - D_i), (S \cup S'_i) \cap N_i) \vdash_{check} e_i \\ \textbf{where} \;\; D'_i = \emptyset \qquad R'_i = \begin{cases} Rec_i \text{ if } type(x) = d \\ \emptyset \quad \text{otherwise} \end{cases} \qquad S'_i = \begin{cases} P_i - Rec_i & \text{if } type(x) = d \\ P_i - R''_i & \text{if } type(x) = r \\ P_i & \text{if } type(x) = s \\ \emptyset & \text{otherwise} \end{cases} \\ \qquad R''_i = \{y \in P_i \cap sharerec(z, e_i) \mid z \in (D \cup D'_i) \cap N_i\} \\ \qquad R'''_i = \{y \in D \cap sharerec(z, e_i) \mid z \in (D \cup D'_i) \cap N_i\} - (D \cap N_i) \\ \qquad R''_i \cap (S_i \cup S'_i) = \emptyset \end{array}}{\textbf{case } x \textbf{ of } \overline{C_i \; \overline{x_{ij}}^{n_i} \to e_i}^{\,n} \vdash_{inf} (D, R, S, N')} \; [\text{CASE}_I]$$

$$\frac{\begin{array}{l}\forall i \in \{1..n\} \,.\, e_i \vdash_{inf} (D_i, R_i, S_i, N_i) \\ \forall i \in \{1..n\} \,.\, P_i = \bigcup_{j=1}^{n_i}\{x_{ij}\} \\ \forall i \in \{1..n\} \,.\, Rec_i = \bigcup_{j=1}^{n_i}\{x_{ij} \mid j \in RecPos(C_i)\} \\ D = \bigcup_{i=1}^{n}(D_i - P_i) \\ \qquad x \in D_p \cup R_p \cup S_p \Rightarrow \forall i \in \{1..n\} \,.\, def(inh(type(x), D_i, R_i, S_i, P_i, Rec_i)) \end{array} \quad type(x) = \begin{cases} d \text{ if } x \in D_p \\ r \text{ if } x \in R_p \\ s \text{ if } x \in S_p \\ n \text{ otherwise} \end{cases}}{\begin{array}{l} \forall i \in \{1..n\} \,.\, ((D_p \cup D_{p_i}) \cap N_i, (R_p \cup R_{p_i} \cup R'_{p_i} \cup R''_{p_i}) - D_i, (S_p \cup S_{p_i}) \cap N_i) \vdash_{check} e_i \\ \textbf{where } D_{p_i} = \emptyset \qquad R_{p_i} = \begin{cases} Rec_i \text{ if } type(x) = d \\ \emptyset \quad \text{otherwise} \end{cases} \qquad S_{p_i} = \begin{cases} P_i - Rec_i & \text{if } type(x) = d \\ P_i - R'_{p_i} & \text{if } type(x) = r \\ P_i & \text{if } type(x) = s \\ \emptyset & \text{otherwise} \end{cases} \\ \qquad R'_{p_i} = \{y \in P_i \cap sharerec(z, e_i) \mid z \in D_p \cap N_i\} \\ \qquad R''_{p_i} = \{y \in (D_p \cup D) \cap sharerec(z, e_i) \mid z \in D_p \cap N_i\} - (D_p \cap N_i) \\ \qquad R'_{p_i} \cap (S_i \cup S_{p_i}) = \emptyset \;\wedge\; R_p \cap S_i = \emptyset \end{array}}{(D_p, R_p, S_p) \vdash_{check} \textbf{case } x \textbf{ of } \overline{C_i \; \overline{x_{ij}}^{n_i} \to e_i}^{\,n}} \; [\text{CASE}_C]$$

Let us define Γ as follows:

$$\begin{array}{rl} \Gamma = & [x : d \mid x \in D \cup D'] \\ & + [x : r \mid x \in R \cup R'] \\ & + [x : s \mid x \in S \cup S'] \end{array}$$

First we shall prove that Γ is well-defined. We will show that the sets $(D \cup D')$, $(R \cup R')$ and $(S \cup S')$ are pairwise disjoint. We show that $(D \cup D') \cap (R \cup R') = \emptyset$. Indeed, by Lemma 1 (1) we have $D \cap R = \emptyset$. On the other hand, as $D' \subseteq N$ and $N \cap R = \emptyset$, we have $D' \cap R = \emptyset$. Moreover $D \cap R' = \emptyset$ and $D' \cap R' = \emptyset$ follow from the properties (7) and (4) from Lemma 1. The equality $(R' \cup R) \cap (S \cup S') = \emptyset$ is proven in a similar way. With respect to $(D \cup D') \cap (S \cup S') = \emptyset$, this follows from Lemma 1 (1, 6).

Now we prove that $\Gamma \geq_{e'} [z : s]$. On one hand we have $z \in dom(\Gamma)$, since z occurs free in e' and by Lemma 1 (5) we have either $z \in D$ or $z \in R$ or $z \in S$ or $z \in N \subseteq D' \cup R' \cup S'$. Moreover, in each case it holds that $\Gamma(z) \geq s$. Moreover, putting the properties (3) and (6) of Lemma 1 together we have:

$$\bigcup_{z \in D \cup D'} sharerec(z, e') \subseteq D \cup R \cup D' \cup R'$$

Thus every variable sharing a recursive child of another with d type in Γ has an unsafe type in this environment. The three conditions for $\geq_{e'}$ are satisfied and hence $\Gamma \geq_{e'} [z : s]$.

Now we shall prove that $\Gamma'_i \vdash e_i : s$, where $\Gamma'_i = \Gamma + \overline{[x_{ij} : \tau_{ij}]}^{n_i}$ and we shall find the corresponding τ_{ij} in order to satisfy the *inh* predicates in rule [CASE]. We know by the induction hypothesis that there exists a Γ_i which satisfies $\Gamma_i \vdash e_i : s$. If we assign $\forall x \in dom(\Gamma_i).\, \Gamma'_i(x) = \Gamma_i(x)$ we will be able to infer, by means of the rules [EXTS] and [EXTD] that $\Gamma'_i \vdash e_i : s$.

Let $x \in dom(\Gamma_i) \cap dom(\Gamma'_i)$ and let us assume that $x \in P_i$, that is, $x = x_{ij}$ for some $j = \{1..n_i\}$. According to the type of the **case** discriminant we have the following cases:

- $z \in D$
  First we assume that $x \in P_i - Rec_i$. From the *inh* predicate in rule [CASE$_I$] it follows that $x \notin D_i$ and $x \notin R_i$. Thus we have either $x \in S_i$ or $x \in N_i$. In the latter case the only possibility is $x \in S'_i$ since $D'_i = \emptyset$ and R'_i only contains variables from Rec_i (otherwise, the *inh* predicate in [CASE$_C$] would not be defined). Since $x \in S_i \cup S'_i$, we have $\Gamma_i(x) = s$ by induction hypothesis. Thus we are forced to assign $\tau_{ij} = s$ in Γ'_i, so the $inh(s, s, \Gamma(z))$ in rule [CASE] holds, since $\Gamma(z)$ has d type and it holds that $\neg utype?(\Gamma'(x), \Gamma(z))$, since $x \notin Rec_i$.
  Now we assume $x \in Rec_i$. The *inh* predicate specifies that $x \notin D_i$ and $x \notin S_i$. The remaining possibility is that $x \in R_i$ or $x \in N_i$. In the latter case we have $x \in R'_i$, since D'_i is empty and S'_i does not contain variables from Rec_i. Since $x \in R_i \cup R'_i$ and hence $\Gamma_i(x) = r$ we can assign $\tau_{ij} = r$ so as to have $\Gamma'(x) = r$. Thus the *inh* predicate of rule [CASE] holds for the variable x.
- $z \in R$
  The use of *inh* in [CASE$_I$] forces x not to belong to D_i. On the other hand we have $D'_i = \emptyset$ and $S'_i = P_i - R'_{p_i}$. The remaining possibilities are $x \in R_i \cup R'_i$ and $x \in S_i \cup S'_i$.
  If $x \in R_i \cup R'_i$ then $\Gamma_i(x) = r$ and we can assign $\tau_{ij} = r$. Thus $\Gamma'_i(x) = r$ allows the *inh* predicate to verify.
  If $x \in S_i \cup S'_i$ then $\Gamma_i(x) = s$ and we can assign $\tau_{ij} = s$. Hence the *inh* predicate also holds for x.
- $z \in S$
  In this case we have $x \notin D_i$ and $x \notin R_i$, and if $x \in N_i$ then $x \in S'_i$. The membership $x \in R'_i$ cannot hold, since it implies $x \in R''_i$ and z would share a recursive child of a variable belonging to $D \cup D'_i$, which contradicts $z \in S$. Thus we have $x \in S_i \cup S'_i$ and by induction hypothesis, $\Gamma_i(x) = s$. We can assign $\tau_{ij} = s$ and hence the *inh* predicate of [CASE] rule is satisfied.
- $z \in N$
  We do a case distinction according to the membership of z: $z \in D'$, $z \in R'$ or $z \in S'$. The proof for each case is similar to the ones above. In this case the *inh* of rule [CASE$_C$] and the definitions of D_{p_i}, R_{p_i} and S_{p_i} force the variable x to have an specific type.

Now we assume that $x \notin P_i$. If $x \in D_i$ then $\Gamma_i(x) = d$ by induction hypothesis. In this case, from the definition of $\sqcup$ it follows that $x \in D$ and hence $\Gamma'_i(x) = \Gamma(x) = d$. The cases $x \in R_i$ and $x \in S_i$ are similar. If $x \in N_i$ then another case distinction is made: $x \in D'_i$, $x \in S'_i$ or $x \in R'_i$. In the first case, $\Gamma_i(x) = d$ and since $x \in D'_i \cap N_i$ we obtain $x \in D'$ and hence $\Gamma'_i(x) = \Gamma_i(x) = d$. Again, the two remaining cases are similar.

As we have seen, $\Gamma \vdash e' : s$ can be inferred by the [CASE] rule of the type system and hence (1) holds. Finally, (2a), (2b) and (2c) hold trivially by the definition of Γ.

$$\boxed{e' \equiv \textbf{case!}\ z\ \textbf{of}\ \overline{C\ \overline{x_{ij}}^{n_i} \to e_i}^{n}}$$

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

$$\frac{\begin{array}{ll}\forall i \in \{1..n\}\ .\ e_i \vdash_{\mathit{inf}} (D_i, R_i, S_i, N_i) & \mathit{def}(\sqcup_{i=1}^{n}(D_i, R_i, S_i, N_i, P_i)) \\ \forall i \in \{1..n\}\ .\ P_i = \bigcup_{j=1}^{n_i}\{x_{ij}\} & \forall i \in \{1..n\}\ .\ \mathit{def}(\mathit{inh}!(D_i, R_i, S_i, P_i, \mathit{Rec}_i)) \\ \forall i \in \{1..n\}\ .\ \mathit{Rec}_i = \bigcup_{j=1}^{n_i}\{x_{ij} \mid j \in \mathit{RecPos}(C_i)\} & R \cap L = \emptyset\ \wedge\ \mathit{type}(z) = T@\rho \\ R = \mathit{sharerec}(z, \textbf{case!}\ z\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n}) & \\ L = \bigcup_{i=1}^{n} \mathit{fv}(e_i) & (D, R', S, N) = \sqcup_{i=1}^{n}(D_i, R_i, S_i, N_i, P_i) \\ \multicolumn{2}{l}{\forall i \in \{1..n\}\ .\ ((D \cup \mathit{Rec}_i) \cap N_i, R \cup R' \cup (R_i' \cup R_i'') - D_i, (S \cup (P_i - \mathit{Rec}_i)) \cap N_i) \vdash_{\mathit{check}} e_i} \\ \multicolumn{2}{l}{\quad \textbf{where}\ \ R_i' = \{y \in P_i \cap \mathit{sharerec}(z, e_i) \mid z \in (D \cup \mathit{Rec}_i) \cap N_i\} - (\mathit{Rec}_i \cap N_i)} \\ \multicolumn{2}{l}{\qquad\quad R_i'' = \{y \in D \cap \mathit{sharerec}(z, e_i) \mid z \in D \cap N_i\} - (D \cap N_i)} \\ \multicolumn{2}{l}{\qquad\quad R_i' \cap (P_i - \mathit{Rec}_i) = \emptyset\ \wedge\ \{y \in \mathit{sharerec}(z, e_i) \mid z \in \mathit{Rec}_i\} \cap (P_i - \mathit{Rec}_i) = \emptyset}\end{array}}{\textbf{case!}\ z\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n} \vdash_{\mathit{inf}} (D \cup \{z\}, (R \cup R' - \{z\}), S, N)}\ [\text{CASE!}_I]$$

$$\frac{\begin{array}{l}\forall i \in \{1..n\}\ .\ e_i \vdash_{\mathit{inf}} (D_i, R_i, S_i, N_i) \\ \forall i \in \{1..n\}\ .\ P_i = \bigcup_{j=1}^{n_i}\{x_{ij}\} \qquad D = \bigcup_{i=1}^{n}(D_i - P_i) \\ \forall i \in \{1..n\}\ .\ \mathit{Rec}_i = \bigcup_{j=1}^{n_i}\{x_{ij} \mid j \in \mathit{RecPos}(C_i)\} \\ \forall i \in \{1..n\}\ .\ \{y \in (P_i - \mathit{Rec}_i) \cap \mathit{sharerec}(z, e_i) \mid z \in D_p \cap N_i\} = \emptyset \\ \forall i \in \{1..n\}\ .\ (D_p \cap N_i, R_p \cup (R'_{p_i} - D_i), S_p \cap N_i) \vdash_{\mathit{check}} e_i \\ \textbf{where}\ R'_{p_i} = \{y \in (D_p \cup D) \cap \mathit{sharerec}(z, e_i) \mid z \in D_p \cap N_i\} - (D_p \cup N_i)\end{array}}{(D_p, R_p, S_p) \vdash_{\mathit{check}} \textbf{case!}\ z\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n}}\ [\text{CASE!}_C]$$

We shall define Γ as follows:

$$\begin{array}{rl}\Gamma = & [x : d \mid x \in D \cup D'] \\ & + [x : r \mid x \in R \cup R'] \\ & + [x : s \mid x \in S \cup S']\end{array}$$

In this case it holds that $z \in D$ and hence $\Gamma(z) = d$. Let Γ_0 be the environment resulting from the removal of the binding $[z : d]$ from Γ:

$$\Gamma_0 = \Gamma \upharpoonright (\mathit{dom}(\Gamma) - \{z\})$$

First we shall prove that $\forall y \in \mathit{sharerec}(z, e')$ the property $\forall i \in \{1..n\}.\ y \notin \mathit{fv}(e_i)$ holds. This is enforced by the assumption $R \cap L = \emptyset$ in [CASE!$_I$] rule.

Now we shall prove that $\Gamma_i' \vdash e_i : s$, where $\Gamma_i' = \Gamma_0 + [z : r] + \overline{[x_{ij} : t_{ij}]}^{n_i}$. We assume that $\Gamma_i \vdash e_i : s$ for some Γ_i. We have to prove that $\Gamma_i'(x) = \Gamma_i(x)$ for all $x \in \mathit{dom}(\Gamma_i)$:

Let $x \in \mathit{dom}(\Gamma_i)$. If $x \in P_i$ and x occurs free in e_i the following case distinction is made:

- $x \in \mathit{Rec}_i$
  By the *inh*! in [CASE!$_I$] we know that $x \notin R_i$, $x \notin S_i$. If $x \in N_i$ then $x \in D_i'$ holds, by the $\vdash_{\mathit{check}}$ in [CASE$_I$].

Hence we have $x \in D_i \cup D'_i$. The induction hypothesis allows us to establish $\Gamma_i(x) = d$. From the assignment $t_{ij} = d$ it follows that $\Gamma_i(x) = \Gamma'_i(x) = d$ and that the conditions enforced by the *inh*! of [CASE!] rule in the type system hold.

- $x \in P_i - Rec_i$
  The proof is similar to the case above. In this case we have $x \in S_i \cup S'_i$ and $\Gamma_i(x) = \Gamma'_i(x) = s$.

On the other hand, if $x \in P_i$ but $x \notin fv(e_i)$, the environment Γ_i can be extended via de [EXTS] and [EXTD] rules in order to include the x_{ij} occurring in Γ'_i. The addition of a condemned variable by means of [EXTD] involves the addition of all variables sharing a recursive descendant of it. However, the condition $\{y \in sharerec(z, e_i) \mid z \in Rec_i\} \cap (P_i - Rec_i) = \emptyset$ in [CASE$_I$] avoids a conflict with nonrecursive patterns whose type is safe.

The case in which $x \notin P_i$ the reasoning is the same to the one corresponding one to the nondestructive **case**. Properties (2a), (2b) and (2c) hold trivially by the definition of Γ.

A.3 Recursive definitions. Fixed point

The Theorem 2 above excludes recursive function definitions. Now we shall extend it to include them. The main problem to deal with is the existence of a fixed point in the successive algorithm iterations.

Firstly, a partial order over signatures is defined. Let $f\ \overline{x_i}^n\ @\ \overline{r_j}^l = e$ a function definition. A signature corresponding to that definition is a tuple (I_D, I_R, I_S, I_N), where $I_R = \emptyset$, $I_D \cup I_S \cup I_N = \{1..n\}$, and I_D, I_S and I_N are pairwise disjoint.

Definition 1. *Let $\sqsubseteq$ be a partial order defined over signatures as follows:*

$$(I_D, \emptyset, I_S, I_N) \sqsubseteq (I'_D, \emptyset, I'_S, I'_N) \Leftrightarrow_{def} \begin{array}{l} I_D \subseteq I'_D \wedge \\ I_D \cup I_S \subseteq I'_D \cup I'_S \wedge \\ I_N \supseteq I'_N \end{array}$$

Trivially it holds that $\sqsubseteq$ is reflexive and transitive. The antisymmetry property can be proven from the fact that I_D, I_S and I_N are pairwise disjoint. Given a function definition $f\ \overline{x_i}^n\ @\ \overline{r_j}^l = e$, the set of possible signatures with the partial order $\sqsubseteq$ constitute a finite lattice whose bottom element is $\bot = (\emptyset, \emptyset, \emptyset, \{1..n\})$

The $\sqsubseteq$ order can be extended to function environments Σ as follows: We have $\Sigma \sqsubseteq \Sigma'$ iff $dom(\Sigma) = dom(\Sigma')$ and for all function symbol $g \in dom(\Sigma) \cap dom(\Sigma')$:

$$\Sigma \vdash g : S \ \wedge\ \Sigma' \vdash g : S' \Rightarrow S \sqsubseteq S'$$

We shall use the $e \vdash^{\Sigma}_{inf} (D, R, S, N)$ notation where necessary in order to specify that the inference algorithm returns (D, R, S, N) for the expression e under a function environment Σ.

Lemma 3. *Let $f\ \overline{x_i}^n\ @\ \overline{r_j}^l = e$ be a function definition and Σ, Σ' two function environments such that $\Sigma \sqsubseteq \Sigma'$. If the algorithm has $e \vdash^{\Sigma}_{inf} (D, R, S, N)$ with the function environment Σ and it has $e \vdash^{\Sigma'}_{inf} (D', R', S', N')$ with the function environment Σ', then:*

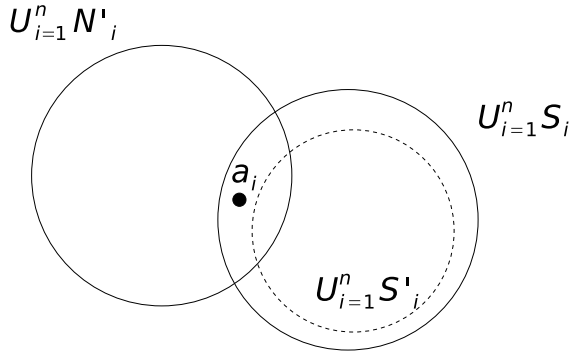

Fig. 10. Situation in which $N \subset N'$

1. $D \cup R \subseteq D' \cup R'$
2. $D \cup R \cup S \subseteq D' \cup R' \cup S'$
3. $N \supseteq N'$

Proof. By structural induction on e.

$\boxed{e \equiv c}$ $\boxed{e \equiv x}$ $\boxed{e \equiv x@r}$ $\boxed{e \equiv x!}$ $\boxed{e \equiv C\ \overline{a_i}^n\ @r}$

It holds trivially, since the result (D, R, S, N) does not depend on Σ.

$\boxed{e \equiv g\ \overline{a_i}^n\ @\overline{r_j}^l}$

Let $\Sigma \vdash g : (I_D, \emptyset, I_S, I_N)$ and $\Sigma' \vdash g : (I'_D, \emptyset, I'_S, I'_N)$. From $I_D \subseteq I'_D$ it follows that $D \subseteq D'$ and $R \subseteq R'$. Hence we have $D \cup D' \subseteq R \cup R'$.

Moreover, since $I_D \cup I_S \subseteq I'_D \cup I'_S$ it holds that $D \cup S \subseteq D' \cup S'$. Hence $D \cup R \cup S \subseteq D' \cup R' \cup S'$.

We shall prove by contradiction that $N \supseteq N'$. Let us assume that there exists an a_i such that $a_i \notin N$ but $a_i \in N'$. From the set inclusion $I_N \supseteq I'_N$ it follows that $\bigcup_{i=1}^{n} N_i \supseteq \bigcup_{i=1}^{n} N'_i$ and hence the only possibility is that $\bigcup_{i=1}^{n} S_i \supset \bigcup_{i=1}^{n} S'_i$. This is shown in Figure 10.

Let $a_i \in \bigcup_{i=1}^{n} N'_i$, that is, $i \in I'_N$. If $a_i \in \bigcup_{i=1}^{n} S_i$ but $a_i \notin \bigcup_{i=1}^{n} S'_i$ we have $i \in I_S$, but $i \notin I'_S$, from which it follows that $a_i \in I'_D$ by the property $I_D \cup I_S \subseteq I'_D \cup I'_S$. Since I'_D and I'_N are disjoint, it holds that $i \notin I'_N$.

$\boxed{e \equiv \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2}$

Let us assume the following results:

$$e_1 \vdash^{\Sigma}_{inf} (D_1, R_1, S_1, N_1)\ e_2 \vdash^{\Sigma}_{inf} (D_2, R_2, S_2, N_2)$$

$$e_1 \vdash^{\Sigma'}_{inf} (D'_1, R'_1, S'_1, N'_1)\ e_2 \vdash^{\Sigma'}_{inf} (D'_2, R'_2, S'_2, N'_2)$$

Property (1) is proven as follows:

$$\begin{array}{ll} & D \cup R \\ = & ((D_1 \cup D_2) - \{x_1\}) \cup (R_1 \cup (R_2 - D_1)) \\ = & (D_1 \cup D_2 \cup R_1 \cup R_2) - \{x_1\} & \text{since } x_1 \notin R_1 \text{ and } x_1 \notin R_2 \\ \subseteq & (D_1' \cup D_2' \cup R_1' \cup R_2') - \{x_1\} & \text{by I.H.} \\ = & ((D_1' \cup D_2') - \{x_1\}) \cup (R_1' \cup (R_2' - D_1')) \\ = & D' \cup R' \end{array}$$

In a similar way we show the property (2):

$$\begin{array}{l} D \cup R \cup S \\ = ((D_1 \cup D_2) - \{x_1\}) \cup (R_1 \cup (R_2 - D_1)) \cup (((S_1 - N_2) \cup S_2) - (\{x_1\} \cup D_2 \cup R_2)) \\ = ((D_1 \cup D_2) \cup (R_1 \cup (R_2 - D_1)) \cup (((S_1 - N_2) \cup S_2) - (D_2 \cup R_2))) - \{x_1\} \\ = (D_1 \cup D_2 \cup R_1 \cup R_2 \cup (S_1 - N_2) \cup S_2) - \{x_1\} \\ \subseteq (D_1' \cup D_2' \cup R_1' \cup R_2' \cup (S_1' - N_2') \cup S_2') - \{x_1\} \\ = ((D_1' \cup D_2') - \{x_1\}) \cup (R_1' \cup (R_2' - D_1')) \cup (((S_1' - N_2') \cup S_2') - (\{x_1\} \cup D_2' \cup R_2')) \\ = D' \cup R' \cup S' \end{array}$$

In order to prove the property (3), it is only necessary the induction hypothesis and the property (2).

$$\boxed{e \equiv \textbf{case } x \textbf{ of } \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n}}$$

We shall assume that the algorithm has for every $i \in \{1..n\}$:

$$e_i \vdash^{\Sigma}_{inf} (D_i, R_i, S_i, N_i) \qquad e_i \vdash^{\Sigma'}_{inf} (D_i', R_i', S_i', N_i')$$

Applying the induction hypothesis:

$$D \cup R = \bigcup_{i=1}^{n}(D_i - P_i) \cup \bigcup_{i=1}^{n}(R_i - P_i) \subseteq \bigcup_{i=1}^{n}(D_i' - P_i) \cup \bigcup_{i=1}^{n}(R_i' - P_i) = D' \cup R'$$

And hence property (1) holds. For the remaining properties we follow a similar reasoning.

$$\boxed{e \equiv \textbf{case!} \ x \textbf{ of } \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n}}$$

The proof is similar to the one corresponding to the nondestructive **case**. The R' set occurring in [CASE!$_I$] does not affect the proof, since this set does not depend on the function environment.

Definition 2. *For any fixed function declaration $f\ \overline{x_i}^n\ @\ \overline{r_j}^l = e$ the function F is defined over environments as follows:*

$$F_{f\ \overline{x_i}^n\ @\ \overline{r_j}^l = e}(\Sigma) = \Sigma\,[g \mapsto \mathit{extract}(\overline{x_i}^n, D, R, S, N)] \ \ \textbf{\textit{where}}\ e \vdash^{\Sigma}_{inf} (D, R, S, N)$$

where extract obtains the signature of the function from the corresponding sets (D, R, S, N) and it is defined as follows:

$$\begin{array}{rl} \mathit{extract}(\overline{x_i}^n, D, \emptyset, S, N) = & (\{i \in \{1..n\} \mid x_i \in D\}, \\ & \emptyset, \\ & \{i \in \{1..n\} \mid x_i \in S\}, \\ & \{i \in \{1..n\} \mid x_i \in N\} \cup \{i \in \{1..n\} \mid x_i \notin D \cup S \cup N\}) \end{array}$$

Lemma 4. *Given a definition* $f\ \overline{x_i}^n\ @\ \overline{r_j}^l = e$*, the function* $F_{f\ \overline{x_i}^n\ @\ \overline{r_j}^l = e}$ *is monotonic w.r.t.* $\sqsubseteq$*. That is:*

$$\Sigma \sqsubseteq \Sigma' \Longrightarrow F_{f\ \overline{x_i}^n\ @\ \overline{r_j}^l = e}(\Sigma) \sqsubseteq F_{f\ \overline{x_i}^n\ @ \overline{r_j}^l = e}(\Sigma')$$

Proof. We shall assume the following results from the inference algorithm:

$$\begin{array}{ll} e \vdash^{\Sigma}_{inf} (D, R, S, N) & (I_D, \emptyset, I_S, I_N) = extract(\overline{x_i}^n, D, R, S, N) \\ e \vdash^{\Sigma'}_{inf} (D', R', S', N') & (I'_D, \emptyset, I'_S, I'_N) = extract(\overline{x_i}^n, D', R', S', N') \end{array}$$

It has to be proven that $(I_D, \emptyset, I_S, I_N) \sqsubseteq (I'_D, \emptyset, I'_S, I'_N)$. By Lemma 3 it holds that $D \cup R \subseteq D' \cup R'$. Since $R = R' = \emptyset$, we have $D \subseteq D'$ and hence $I_D \subseteq I'_D$. The remaining conditions for $\sqsubseteq$ can be easily obtained from Lemma 3.

The lemma above proves that we get a "greater" signature in each algorithm iteration and hence, a fixpoint can be reached by Kleene's ascending chain. This allow us to modify the Theorem 2 so as to include recursive definitions:

Theorem 1. *Let us assume that the function declaration* $f\ \overline{x_i}^n\ @\ \overline{r_j}^l = e$ *has been successfully typed by the inference algorithm and let* e' *be any subexpression of* e *for which the algorithm has got* $e' \vdash_{inf} (D, R, S, N)$ *and* $(D', R', S') \vdash^*_{check} e'$*. Then there exists a safe type* s' *and a well-formed type environment* Γ *such that:*

1. $\Gamma \vdash e' : s$
2. $\forall x \in scope(e')$:
   (a) $\Gamma(x) = d \Leftrightarrow x \in D \cup D'$
   (b) $\Gamma(x) = s \Leftrightarrow x \in S \cup S'$
   (c) $\Gamma(x) = r \Leftrightarrow x \in R \cup R'$

Proof. The proof is as seen in Theorem 2. The only additional case to consider is the one corresponding to a recursive call, that is, $e' = f\ \overline{x_i}^n\ @\ \overline{r_j}^l$. Let $(I_D, \emptyset, I_S, I_N)$ the signature obtained for f. The set I_N may be nonempty. For all $i \in \{1..n\}$ an environment Γ_i is defined as follows:

$$\Gamma_i = \begin{cases} [a_i : d] \text{ if } a_i \text{ is a variable and } a_i \in D \cup D' \\ [a_i : s] \text{ if } a_i \text{ is a variable and } a_i \in S \cup S' \end{cases}$$

In the same way as Theorem 2, Γ' is defined as follows:

$$\Gamma' = [\overline{r_j : \rho_j}^l, f : \sigma] + \bigoplus_{i=0}^{n} \Gamma_i$$

We shall show that Γ' is well-defined. Indeed, let us assume that there exists an $x \in dom(\Gamma_i) \cap dom(\Gamma_j)$ such that $\Gamma_i(x) = \Gamma_j(x) = d$ for some $i, j \in \{1..n\}$ such that $i \neq j$. If $x \in D$ the same reasoning as in Theorem 2 may be applied in order to get a contradiction. Moreover, it is not possible that $x \in D'$, since the following condition of [APP$_C$] rule prevents a parameter from occurring in two condemned positions:

$$\forall a_i \in D_p.\ (\#j : 1 \leq j \leq n : a_i = a_j) = 1$$

Now properties (1) and (2) can be proven in the same way as Theorem 2.