

An Inference Algorithm for Guaranteeing Safe Destruction

Manuel Montenegro, Ricardo Peña, and Clara Segura

Dpto. Sistemas Informáticos y Computación, Univ. Complutense de Madrid, Spain
manuelmont@gmail.com, ricardo@sip.ucm.es, csegura@sip.ucm.es*

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 reasoned about.

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.

In previous papers [1, 2] we proposed a semi-explicit approach to memory control by defining a functional language, called *Safe*, 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 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 reason about 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. Finally, in Section 5 we show some examples whose types have been successfully inferred.

*Work partially supported by the Spanish project TIN2004-07943-C04.

2 SUMMARY OF SAFE

2.1 Syntax

We start by reproducing some crucial definitions which underlie the language. In [2] we compared our language design with other approaches using regions and memory management facilities.

Definition 1. A **region** is a contiguous memory area in the heap where data structures can be constructed, read, or destroyed. It is allocated and freed as a whole, in constant time.

Definition 2. A **cell** is a small memory space, big enough to hold a data constructor. In implementation terms, a cell contains the mark (or code pointer) of the constructor, and a representation of the free variables to which the constructor is applied. These may consist, either of basic values, or of pointers to non-basic values.

Definition 3. A **data structure**, in the following a *DS*, is the set of cells obtained by starting at one cell considered as the root, and taking the transitive closure of the relation $C_1 \rightarrow C_2$, where C_1 and C_2 are cells of the same type T , and in C_1 there is a pointer to C_2 .

That means that, for instance in a list of type $[[a]]$, we are considering as a DS the cells of the cons-nil spine of the *outermost* list, but not those belonging to the individual innermost lists. Each one of the latter constitute a separate DS.

The following decisions were taken:

1. A DS completely resides in one region.
2. One DS can be part of another DS, or two DSs can share a third one.
3. The basic values —integers, booleans, etc.— do not allocate cells in regions. They live inside the cells of DSs, or in the stack.
4. A function of n parameters can access:
 - Its n parameters, each one residing in a possibly different region.
 - Its **output region**, whenever it builds a DS as a result. There is at most one output region per function. Delivering this region identifier as a parameter is the responsibility of the call. We force functions to leave their result in an output region belonging to the calling context in order to safely delete the intermediate results computed by the function.
 - Its (optional) **working region**, referred to through the reserved identifier *self*, where it may create intermediate DSs. The working region has the same lifetime as the function call: It is allocated at each invocation and freed at function termination.
5. If a parameter of a function is a DS, it can be destroyed by the function. We will say that the parameter is **condemned** because this capability depends on the function definition, not on its use.

$prog$	$\rightarrow dec_1; \dots; dec_n; expr$	
dec	$\rightarrow f \overline{x_i^n} r = expr$	{recursive, polymorphic function}
	$ f \overline{x_i^n} = expr$	
$expr$	$\rightarrow a$	{atom: literal c or variable x }
	$ x@r$	{copy}
	$ x!$	{reuse}
	$ (f \overline{a_i^n})@r$	{function application}
	$ (f \overline{a_i^n})$	{function application}
	$ (C \overline{a_i^n})@r$	{constructor application}
	$ \mathbf{let} x_1 = \overline{expr_1} \mathbf{in} expr$	{non-recursive, monomorphic}
	$ \mathbf{case} x \mathbf{of} \overline{alt_i^n}$	{read-only case}
	$ \mathbf{case!} x \mathbf{of} \overline{alt_i^n}$	{destructive case}
alt	$\rightarrow C \overline{x_i^n} \rightarrow expr$	

FIGURE 1. First-order functional language *Core-Safe*

6. The capabilities a function has on its accessible DSs and regions are: a function may only read a DS which is a **safe** (not condemned) parameter; a function may read (before destroying it), and must destroy, a DS which is a condemned parameter; a function may construct, read, or destroy DSs, in either its output or its working region.

The syntax of *Core-Safe* is shown in Figure 1. This is a first-order eager functional language where sharing is expressed using variables in function and constructor applications. This is obtained after the desugaring of a higher-level language similar to Haskell or ML, called *Full-Safe*. However, several analyses and type inference are done at core level. This is a usual approach in many compilers.

A program $prog$ in *Core-Safe* is a sequence of possibly recursive polymorphic function definitions¹ followed by a main expression $expr$, calling them, whose value is the program result. Function definitions building and returning a new DS will have an additional parameter r , which is the output region, where the resulting DS is to be constructed. In the right hand side expression only r and its own working region $self$ may be used. Polymorphic algebraic data types definitions are also allowed. We will assume they are defined separately through **data** declarations.

The program expressions include variables, literals, function and constructor applications, and also **let** and **case** expressions, but there are some additional expressions:

If x is a DS, the expression $x@r$ represents a copy in region r of the DS accessed from x . The DS x must live in a region $r' \neq r$. Both x and $x@r$ have the same recursive structure and they share their non-recursive substructures.

The expression $x!$ means the reusing of the destroyable DS to which x points. This is useful when we do not want to destroy completely a condemned parameter but instead to reuse part of it. In semantic terms, x and $x!$ point to the same physical structure but, in language terms, once $x!$ is used, the name x becomes inaccessible in the subsequent text.

¹The extension to mutual recursion would pose no special problems, but we restrict ourselves to non-mutual recursion in order to ease the presentation.

$$\begin{aligned}
revD &:: \forall a, \rho_1, \rho_2. [a]!@ \rho_1 \rightarrow \rho_2 \rightarrow [a]@ \rho_2 \\
revD \ x \ r &= (revauxD \ x \ []@r)@r \\
\\
revauxD &:: \forall a, \rho_1, \rho_2. [a]!@ \rho_1 \rightarrow [a]@ \rho_2 \rightarrow \rho_2 \rightarrow [a]@ \rho_2 \\
revauxD \ x \ y \ r &= \mathbf{case!} \ x \ \mathbf{of} \\
&\quad [] \rightarrow y \\
&\quad x : xx \rightarrow \mathbf{let} \ x_1 = (x : y)@r \\
&\quad \mathbf{in} \ (revauxD \ xx \ x_1)@r
\end{aligned}$$

FIGURE 2. Destructive list inversion

In function application we have a special syntax $@r$ to express the inclusion of the additional output region parameter. Using the same syntax, we express that a constructor application is to be allocated in region r .

The **case!** expression indicates that the outer constructor of x is disposed after the pattern matching so that x is not accessible anymore. The recursive substructures may be explicitly destroyed in the subsequent code via another **case!** or reused via $x!$. A condemned variable may be read but, once its content has been destroyed or reused in another structure, it may not be accessed again. This is what the type system guarantees. It annotates the type of such variable with a $!$.

We show now with an example how to use the language facilities. The example is the function that reverses a list and, at the same time, destroys it. Its Core-Safe code is shown in Figure 2. We show also the types that the functions receive from the type inference algorithm presented in Section 3. We use the usual auxiliary function with an accumulator parameter. Notice that the differences with the usual functional version are, on the one hand, the use of the region parameter r and, on the other, that a **case!** is used over the original list. The recursive application of the function destroys it completely. Those who call $revD$ should know that the argument is lost in the inversion process, and should not try to use it anymore. This is reflected in the type of the first argument with a $!$ annotation.

3 TYPE SYSTEM

In this section we describe a polymorphic type system with data types for programming in a safe way when using the destruction facilities offered by the language. This type system has been proven correct with respect to a small-step operational semantics of the language, and it is the main topic of a journal paper currently in preparation. Here we describe it briefly.

In Figure 3 the types syntax is defined. 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 at the outermost level. In-danger types arise as an intermediate step during typing but notice that functional types (i.e. the types of functions) can only include either safe or condemned types.

The predicate $utype?(t, t')$ says whether two types have the same underlying (i.e. safe) type. The predicate $safe?(\tau)$ tells us whether τ is a safe type, while $unsafe?(\tau)$ tells us

$\tau \rightarrow t$	{outermost}	$r \rightarrow T \bar{\rho} \bar{s} \# @ \rho'$	
	r	{in danger}	$b \rightarrow a$ {variable}
	σ	{polym. function}	B {basic}
	ρ	{region}	$tf \rightarrow \bar{T}_i^n \rightarrow \rho' \rightarrow T \bar{\rho} \bar{s} @ \rho'$
$t \rightarrow s$	{safe}	$\bar{T}_i^n \rightarrow s$	
	d	{condemned}	$\bar{s}_i^n \rightarrow \rho' \rightarrow T \bar{\rho} \bar{s} @ \rho'$ {constructor}
$s \rightarrow T \bar{\rho} \bar{s} @ \rho'$		$\sigma \rightarrow \forall a. \sigma$	
	b	$\forall \rho. \sigma$	
$d \rightarrow T \bar{\rho} \bar{s} ! @ \rho'$		tf	

FIGURE 3. Type expressions

whether τ is either a condemned or in-danger type. Predicates $region?(\tau)$ and $function?(\tau)$ respectively indicate that τ is a region type or a functional type.

The intended semantics of these types is the following:

1. Safe types s represent pointers to closures in the heap that may be read, copied or used to build other closures, but that cannot be reused or destroyed.
2. Condemned types d are given to those pointers directly involved in the destructive action of a **case!** expression or a reuse somewhere in the program. More specifically, not only the discriminant of a **case!** expression is condemned but also the recursive pattern variables in its alternatives. The closure pointed to by a condemned pointer may be read using a **case** or copied before the **case!** destroys it, but it may not be used to build another closure or returned as a result. Once the closure is destroyed, such pointer may not be accessed any more.
3. In-danger types r are given to those pointers that are not directly condemned in the program but that point to a closure that is a recursive child of a condemned one, i.e. it potentially may be a dangling pointer. Consequently, they have the same access constraints as the condemned ones.

Consistently, condemned and in-danger types appear only at the outermost level: It is not reasonable to build a closure with potentially dangling pointers.

Functional types returning algebraic types may have an additional region parameter ρ' where the result must be allocated. Only if the function returns a basic value or does not build a DS (e.g. it returns a part of one of its arguments) then the region parameter is not necessary. However data constructors always need a region parameter where to build a DS. As recursive sharing of DSs may happen only inside the same region, the constructors are given types indicating that the recursive substructure and the structure itself must live in the same region. For example, in the case of lists:

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

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

A partial order between types is defined allowing a condemned or in danger DS to be read or copied while they are not destroyed: $\tau \geq \tau, T! @ \rho \geq T @ \rho$, and $T \# @ \rho \geq T @ \rho$.

This partial order is extended below to type environments in the context of the expression being typed.

In the type environments, Γ , we can find region type assignments $r : \rho$, variable type assignments $x : t$, and polymorphic type assignments to functions $f : \sigma$. Additionally we attach to the environments the set P of safe input DS of the function that is being defined, and we write Γ^P . In the rules we will also use $gen(tf, \Gamma)$ and $tf \trianglelefteq \sigma$ to denote respectively (standard) generalization of a monomorphic type and (restricted to safe types) instantiation of a polymorphic type.

Several operators over environments are used:

- Usual operator $+$ demands disjoint domains in order to join them.
- Operator \otimes is defined only if common variables have the same type.
- Operator \oplus demands that variables may have an unsafe type at most in one of the environments.

In Figure 4 we show the type rules for the expressions (there are also rules for function definitions but we do not show them here). We explain here in detail some of them.

In some rules we use the sets $shareall(x_1, e)$ and $sharerrec(x_1, e)$, obtained from the sharing analysis defined in [2]:

- $shareall(x, e)$ returns the set of all the variables in scope in e which, at runtime, may share any substructure of the structure pointed to by x .
- $sharerrec(x, e)$ returns the set of all the variables in scope in e which, at runtime, may share any recursive substructure of the structure pointed to by x .

There are rules for typing literals ([LIT]), and variables of several kinds ([VAR], [REGION] and [FUNCTION]). Notice that these are given a type under the smallest typing environment. Rules [EXTS] and [EXTD] allow to extend the typing environments in a controlled way:

1. Only fresh variables in scope may be added.
2. Rule [EXTS] allows the addition of variables with safe types, in-danger types, region types or functional types.
3. If a variable with a condemned type is added, all those variables sharing its recursive substructure but itself (i.e. $sharerrec(x, e) - \{x\}$) must be also added to the environment with its corresponding in-danger type. This is controlled by rule [EXTD]. Notation $danger(type(y))$ represents the in-danger version of the Hindley-Milner type inferred for variable y (at this point they have already been inferred, see the following section).

Rule [COPY] allows any variable to be copied. Even condemned variables may be copied before being destroyed. This is expressed by extending the previously defined partial order between types to environments:

$$\begin{aligned} \Gamma_1 \geq_e \Gamma_2 \equiv & \quad dom(\Gamma_2) \subseteq dom(\Gamma_1) \wedge \\ & \quad \forall x \in dom(\Gamma_2). \Gamma_1(x) \geq \Gamma_2(x) \wedge \\ & \quad \forall z \in dom(\Gamma_1). \Gamma_1(z) = d \rightarrow \forall z \in sharerrec(x, e) - \{x\}. unsafe?(\Gamma_1(z)) \end{aligned}$$

$$\begin{array}{c}
\frac{\Gamma^P \vdash e : s \quad x \notin \text{dom}(\Gamma^P) \quad \text{safe?}(\tau) \vee \text{danger?}(\tau) \vee \text{region?}(\tau) \vee \text{function?}(\tau)}{\Gamma^P + [x : \tau] \vdash e : s} \text{ [EXTS]} \quad \frac{\Gamma^P \vdash e : s \quad x \notin \text{dom}(\Gamma^P) \quad R = \text{sharerec}(x, e) - \{x\} \quad \Gamma_R = \{y : \text{danger}(\text{type}(y)) \mid y \in R\}}{\Gamma^P \otimes \Gamma_R + [x : d] \vdash e : s} \text{ [EXTD]} \\
\\
\frac{}{\emptyset \vdash c : B} \text{ [LIT]} \quad \frac{}{[x : s]^P \vdash x : s} \text{ [VAR]} \quad \frac{}{[r : \rho]^P \vdash r : \rho} \text{ [REGION]} \quad \frac{tf \trianglelefteq \sigma}{[f : \sigma]^P \vdash f : tf} \text{ [FUNCTION]} \\
\\
\frac{R = \text{sharerec}(x, x!) - \{x\} \quad \Gamma_R = \{y : \text{danger}(\text{type}(y)) \mid y \in R\}}{\Gamma_R + [x : T!@ \rho]^P \vdash x! : T @ \rho} \text{ [REUSE]} \quad \frac{\Gamma_1^P \geq_{x@r} [x : T @ \rho', r : \rho] \quad \rho \neq \rho'}{\Gamma_1^P \vdash x@r : T @ \rho} \text{ [COPY]} \\
\\
\frac{\Gamma_1^P \vdash e_1 : s_1 \quad \Gamma_2^P + [x_1 : s_1] \vdash e : s \quad C = \text{shareall}(x_1, e)}{\Gamma_1^P \triangleright_C^{fv(e)} \Gamma_2^P \vdash \text{let } x_1 = e_1 \text{ in } e : s} \text{ [LET1]} \\
\\
\frac{\Gamma_1^P \vdash e_1 : s_1 \quad \Gamma_2^P + [x_1 : d_1] \vdash e : s \quad d_1 = \bar{s}_1 \quad C = \text{shareall}(x_1, e) \quad R = \text{sharerec}(x_1, e) \quad P \cap R = \emptyset}{\Gamma_1^P \triangleright_{C-R}^{fv(e)} \Gamma_2^P \vdash \text{let } x_1 = e_1 \text{ in } e : s} \text{ [LET2]} \\
\\
\frac{\Gamma_0^P \vdash f : \bar{t}_i^n \rightarrow \rho \rightarrow T @ \rho \quad \Gamma_0^P \vdash r : \rho \quad \Gamma^P = \Gamma_0^P + \bigoplus_{i=1}^n [a_i : t_i] \quad R = \bigcup_{i=1}^n \{\text{sharerec}(a_i, (f \bar{a}_i^n) @ r) - \{a_i\} \mid \text{cdm?}(t_i)\} \quad \Gamma_R = \{y : \text{danger}(\text{type}(y)) \mid y \in R\}}{\Gamma_R + \Gamma^P \vdash (f \bar{a}_i^n) @ r : T @ \rho} \text{ [APP1]} \\
\\
\frac{\Sigma(C) = \sigma \quad \bar{s}_i^n \rightarrow \rho \rightarrow T \bar{s}_j^k @ \rho \trianglelefteq \sigma \quad \Gamma^P = \bigoplus_{i=1}^n [a_i : s_i] + [r : \rho]}{\Gamma^P \vdash (C \bar{a}_i^n) @ r : T \bar{s}_j^k @ \rho} \text{ [CONS]} \\
\\
\frac{(\forall i \in \{1..n\}). \Sigma(C_i) = \sigma_i \quad (\forall i \in \{1..n\}). \bar{s}_{ij}^{n_i} \rightarrow \rho \rightarrow T @ \rho \trianglelefteq \sigma_i \quad \Gamma^P \geq_{\text{case } x \text{ of } \bar{C}_i \bar{x}_{ij}^{n_i} \rightarrow e_i^n} [x : T @ \rho] \quad (\forall i \in \{1..n\}). \forall j \in \{1..n_i\}. \text{inh}(t_{ij}, s_{ij}, \Gamma^P(x)) \quad (\forall i \in \{1..n\}). \Gamma^P + [x_{ij} : \tau_{ij}]^{n_i} \vdash e_i : s}{\Gamma^P \vdash \text{case } x \text{ of } \bar{C}_i \bar{x}_{ij}^{n_i} \rightarrow e_i^n : s} \text{ [CASE]} \\
\\
\frac{(\forall i \in \{1..n\}). \Sigma(C_i) = \sigma_i \quad (\forall i \in \{1..n\}). \bar{s}_{ij}^{n_i} \rightarrow \rho \rightarrow T @ \rho \trianglelefteq \sigma_i \quad R = \text{sharerec}(x, \text{case! } x \text{ of } \bar{C}_i \bar{x}_{ij}^{n_i} \rightarrow e_i^n) \quad (\forall i \in \{1..n\}). \forall j \in \{1..n_i\}. \text{inh!}(t_{ij}, s_{ij}, T!@ \rho) \quad \forall z \in R, i \in \{1..n\}. z \notin \text{fv}(e_i) \quad (\forall i \in \{1..n\}). \Gamma^P + [x : T \# @ \rho] + [x_{ij} : t_{ij}]^{n_i} \vdash e_i : s}{\Gamma^P + [x : T!@ \rho] \vdash \text{case! } x \text{ of } \bar{C}_i \bar{x}_{ij}^{n_i} \rightarrow e_i^n : s} \text{ [CASE!]}
\end{array}$$

FIGURE 4. Rules for expressions

$$\begin{aligned}
\text{def } (\Gamma_1^P \triangleright_C^L \Gamma_2^P) &\equiv (\forall x \in \text{dom}(\Gamma_1^P) \cap \text{dom}(\Gamma_2^P). \text{utype?}(\Gamma_1^P(x), \Gamma_2^P(x))) \wedge \\
&(\forall x \in \text{dom}(\Gamma_1^P). \text{unsafe?}(\Gamma_1^P(x)) \rightarrow x \notin L) \wedge \\
&(\forall x \in C \cap L \cap \text{dom}(\Gamma_2^P). \neg \text{unsafe?}(\Gamma_2^P(x))) \\
\forall x \in \text{dom}(\Gamma_1^P) \cup \text{dom}(\Gamma_2^P). \Gamma_1^P \triangleright_C^L \Gamma_2^P(x) &= \begin{cases} \Gamma_2^P(x) & \text{if } x \notin \text{dom}(\Gamma_1^P) \vee \\ & (x \in \text{dom}(\Gamma_1^P) \cap \text{dom}(\Gamma_2^P) \wedge \text{safe}(\Gamma_1^P(x))) \\ \Gamma_1^P(x) & \text{otherwise} \end{cases}
\end{aligned}$$

FIGURE 5. Operator over environments

This operator is later used also in rule [CASE] indicating that it is possible to read any variable, even a condemned one, before it is destroyed.

Rules [LET1] and [LET2] control the intermediate results. We use a family of operators \triangleright_C^L , being C and L sets of variables, formally defined in Figure 5. If the corresponding operator is not well defined then the rule cannot be applied.

An invariant of the type system (see below) tells that if a variable is 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 from a let-bound expression to the main continuation expression.

Rule [LET2] allows the intermediate result x_1 to be used destructively in the main expression e if desired. But this only may be done if the safe input parameters do not share a recursive substructure of x_1 ($P \cap \text{sharerec}(x_1, e) = \emptyset$). In this rule operator \triangleright guarantees that:

1. Each variable y condemned or in-danger in e_1 may not be referenced in e (i.e. $y \notin \text{fv}(e)$), as it could be a dangling reference.
2. Those variables sharing non-recursive descendants of x_1 (i.e. those in $\text{shareall}(x_1, e) - \text{sharerec}(x_1, e)$) but itself which are referenced in e may not have condemned or in-danger types in e : Otherwise x_1 would be corrupted.
3. Those variables marked as condemned or in-danger either in Γ_1^P or in Γ_2^P will keep those types in the combined environment.

Rule [LET1] is applied when the intermediate result is safely used in the main expression. The main difference with the previous one is that the second constraint must be even more restrictive: None of the variables sharing any (recursive or non-recursive) descendant of x_1 may have unsafe types.

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. The rest of the rules ensure that none of them can be used anymore.

Rule [APP1] deal with function application. The use of the operator \oplus avoids a variable to be used in two or more different positions unless they are all read-only parameters. Otherwise undesired side-effects could happen. There is also a rule [APP2] for functions without the additional region parameter.

$$\begin{array}{ll}
inh(s, s, \tau) & \longleftrightarrow ro?(\tau) \vee dgr?(\tau) \vee (\neg utype?(s, \tau) \wedge cmd?(\tau)) \\
inh(danger(s), s, \tau) & \longleftrightarrow dgr?(\tau) \vee (utype?(s, \tau) \wedge cmd?(\tau)) \\
\\
inh!(s, s, d) & \longleftrightarrow \neg utype?(s, d) \\
inh!(d, s, d) & \longleftrightarrow utype?(s, d)
\end{array}$$

FIGURE 6. Definitions of inheritance compatibility

The set R collects all the variables sharing a recursive substructure of a condemned parameter. These are marked as in-danger in environment Γ_R . The condemned actual parameters are marked as such in Γ^P . The combined environment 'informs' the rest of the text of these type assignments. Combining this rule with [LET1] and [LET2], whenever a variable is used in a condemned position the subsequent text could access neither to it nor to the variables sharing a recursive part of it.

Rule [CONS] is more restrictive as only read-only variables can be used to construct a DS.

In rule [CASE!] the discriminant x 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 now. Such those variables are added to the set R . This means that access to recursive substructures of x can only be achieved through the pattern variables, as it is safe to access recursive substructures that have not been destroyed yet.

Relation $inh!$, defined in Figure 6, determines the types inherited by pattern variables, according to the types semantics previously explained: Non-recursive pattern variables are safe while recursive pattern variables are condemned. In fact it can be considered a function, and sometimes we will write $t = inh!(s, d)$ instead of $inh!(t, s, d)$. As recursive pattern variables inherit condemned types, the type environments for the alternatives must contain all the variables sharing their recursive substructures with in-danger types. In particular x appears with an in-danger type. In order to type the whole expression we must change such type to a condemned one because x is being destroyed.

In rule [CASE] a different inheritance relation is used because the discriminant may be of any type according to the partial order previously defined. Relation inh , defined in Figure 6, determines which types are acceptable for pattern variables, according to the types semantics previously explained:

- If the discriminant is safe, so must be all the pattern variables.
- If the discriminant is in-danger, the pattern variables may be safe or in-danger.
- If the discriminant is condemned, recursive pattern variables ($utype?(s, t)$) are in-danger while non-recursive ones are safe.

One of the key points to prove the correctness of the type system with respect to the semantics is the following invariant property that may be proved by structural induction over the expression being typed.

Lemma 4. *If $\Gamma^P \vdash e : s$ and $\Gamma^P(x) = d$ then*

$$\forall y \in \text{sharerec}(x, e) - \{x\}. y \in \text{dom}(\Gamma^P) \wedge \text{unsafe?}(\Gamma^P(y)).$$

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 than those really needed. We are interested in *minimal* types in the sense of being as much polymorphic as possible and having as less unsafe types as possible.

As an example, let consider the following function:

$$f(x : xs)@r = xs@r$$

The type system can give f type $[a]@rho - > rho' - > [a]@rho'$ but also the type $[a]!@rho - > rho' - > [a]@rho'$. The 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 assigning polymorphic types to region variables and handling inequalities of the form $\rho_1 \neq \rho_2$ arising from the typing rule of the COPY expression. 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. Eventhough the information is consistent it may be

$$\begin{array}{c}
\frac{}{c \vdash_{inf} (\mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0})} \text{[LIT}_I\text{]} \quad \frac{}{x \vdash_{inf} (\mathbf{0}, \mathbf{0}, \{x\}, \mathbf{0})} \text{[VAR}_I\text{]} \quad \frac{}{x @ r \vdash_{inf} (\mathbf{0}, \mathbf{0}, \mathbf{0}, \{x\})} \text{[COPY}_I\text{]} \\
\\
\frac{R = \text{sharerec}(x, x!) - \{x\} \quad \text{type}(x) = T @ \rho}{x! \vdash_{inf} (\{x\}, R, \mathbf{0}, \mathbf{0})} \text{[REUSE}_I\text{]} \quad \frac{\forall i \in \{1..n\}. a_i \vdash_{inf} (\mathbf{0}, \mathbf{0}, S_i, \mathbf{0})}{C \bar{a}_i^n @ r \vdash_{inf} (\mathbf{0}, \mathbf{0}, \bigcup_{i=1}^n S_i, \mathbf{0})} \text{[CONS}_I\text{]} \\
\\
\frac{\begin{array}{l}
\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) = \mathbf{0} \quad R \cap (\bigcup_{i=1}^n S_i) = \mathbf{0} \\
\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) = \mathbf{0} \quad R \cap (\bigcup_{i=1}^n D_i) = \mathbf{0} \\
\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 = \mathbf{0} \quad R \cap (\bigcup_{i=1}^n N_i) = \mathbf{0} \\
\Sigma \vdash f : (I_D, \mathbf{0}, I_S, I_N) \quad R = \bigcup_{i=1}^n \{\text{sharerec}(a_i, f \bar{a}_i^n @ r) - \{a_i\} \mid a_i \in D_i\}
\end{array}}{f \bar{a}_i^n @ r \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\text{]} \\
\\
\frac{\begin{array}{l}
C = \begin{cases} \text{shareall}(x_1, e_2) & \text{if } x_1 \notin D_2 \cup R_2 \\ \text{shareall}(x_1, e_2) - \text{sharerec}(x_1, e_2) & \text{if } x_1 \in D_2 \end{cases} \\
e_1 \vdash_{inf} (D_1, R_1, S_1, N_1) \quad N = N_1 - (D_2 \cup R_2 \cup S_2) \cup N_2 \\
e_2 \vdash_{inf} (D_2, R_2, S_2, N_2) \quad \text{def}((D_1 \cup R_1) \triangleright_C^{FV(e_2)} (D_2 \cup R_2)) \\
(\mathbf{0}, \mathbf{0}, N_1 \cap (D_2 \cup R_2 \cup S_2)) \vdash_{check} e_1 \quad (\mathbf{0}, \mathbf{0}, (S_1 \cup \{x_1\}) \cap N_2) \vdash_{check} e_2
\end{array}}{\text{let } x_1 = e_1 \text{ 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\text{]} \\
\\
\frac{\begin{array}{l}
\forall i \in \{1..n\}. e_i \vdash_{inf} (D_i, R_i, S_i, N_i) \quad \text{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}\} \quad (D, R, S, N) = \sqcup_{i=1}^n (D_i, R_i, S_i, N_i, P_i) \\
\forall i \in \{1..n\}. \text{Rec}_i = \bigcup_{j=1}^{n_i} \{x_{ij} \mid j \in \text{RecPos}(C_i)\} \quad \forall i \in \{1..n\}. \text{def}(\text{inh}(\text{type}(x), D_i, R_i, S_i, P_i, \text{Rec}_i))
\end{array}}{\begin{array}{l}
\text{type}(x) = \begin{cases} d & \text{if } x \in D \\ r & \text{if } x \in R \\ s & \text{if } x \in S \\ n & \text{otherwise} \end{cases} \quad 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) \cap N_i, (S \cup S'_i) \cap N_i) \vdash_{check} e_i \\
\text{where } D'_i = \mathbf{0} \quad R'_i = \begin{cases} \mathbf{0} & \text{if } \text{type}(x) = s \\ \mathbf{0} & \text{if } \text{type}(x) = r \\ \text{Rec}_i & \text{if } \text{type}(x) = d \end{cases} \quad S'_i = \begin{cases} P_i & \text{if } \text{type}(x) = s \\ \mathbf{0} & \text{if } \text{type}(x) = r \\ P_i - \text{Rec}_i & \text{if } \text{type}(x) = d \end{cases}
\end{array}}{\text{case } x \text{ of } \overline{C_i \bar{x}_{ij}^{n_i}} \rightarrow e_i^n \vdash_{inf} (D, R, S, N')} \text{[CASE}_I\text{]} \\
\\
\frac{\begin{array}{l}
\forall i \in \{1..n\}. e_i \vdash_{inf} (D_i, R_i, S_i, N_i) \quad \text{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}\} \quad (D, R', S, N) = \sqcup_{i=1}^n (D_i, R_i, S_i, N_i, P_i) \\
\forall i \in \{1..n\}. \text{Rec}_i = \bigcup_{j=1}^{n_i} \{x_{ij} \mid j \in \text{RecPos}(C_i)\} \quad \forall i \in \{1..n\}. \text{def}(\text{inh}!(D_i, R_i, S_i, P_i, \text{Rec}_i)) \\
R = \text{sharerec}(x, \text{case! } x \text{ of } \overline{C_i \bar{x}_{ij}^{n_i}} \rightarrow e_i^n) - \{x\} \quad \text{def}((R \cup \{x\}) \triangleright_{\mathbf{0}}^L (D \cup R')) \\
L = FV(\text{case! } x \text{ of } \overline{C_i \bar{x}_{ij}^{n_i}} \rightarrow e_i^n) - \{x\} \quad \text{type}(x) = T @ \rho \\
\forall i \in \{1..n\}. ((D \cup \text{Rec}_i) \cap N_i, R' \cap N_i, (S \cup (P_i - \text{Rec}_i)) \cap N_i) \vdash_{check} e_i
\end{array}}{\text{case! } x \text{ of } \overline{C_i \bar{x}_{ij}^{n_i}} \rightarrow e_i^n \vdash_{inf} (D \cup \{x\}, R \cup (R' - \{x\}), S, N)} \text{[CASE!}_I\text{]}
\end{array}$$

FIGURE 7. Bottom-up inference rules

$$\begin{aligned}
\text{def}(\text{inh}(n, D_i, R_i, S_i, P_i, \text{Rec}_i)) &\equiv \text{true} \\
\text{def}(\text{inh}(s, D_i, R_i, S_i, P_i, \text{Rec}_i)) &\equiv P_i \cap (D_i \cup R_i) = \emptyset \\
\text{def}(\text{inh}(r, D_i, R_i, S_i, P_i, \text{Rec}_i)) &\equiv P_i \cap D_i = \emptyset \\
\text{def}(\text{inh}(d, D_i, R_i, S_i, P_i, \text{Rec}_i)) &\equiv \text{Rec}_i \cap (D_i \cup S_i) = \emptyset \wedge (P_i - \text{Rec}_i) \cap (D_i \cup R_i) = \emptyset \\
\text{def}(\text{inh}!(D_i, R_i, S_i, P_i, \text{Rec}_i)) &\equiv \text{Rec}_i \cap (R_i \cup S_i) = \emptyset \wedge (P_i - \text{Rec}_i) \cap (D_i \cup R_i) = \emptyset \\
\text{def}(\sqcup_{i=1}^n (D_i, R_i, S_i, N_i, P_i)) &\equiv \forall i, j \in \{1..n\}. i \neq j \Rightarrow \begin{aligned} &(D_i - P_i) \cap (R_j - P_j) = \emptyset \wedge \\ &(D_i - P_i) \cap (S_j - P_j) = \emptyset \wedge \\ &(R_i - P_i) \cap (S_j - P_j) = \emptyset \end{aligned}
\end{aligned}$$

$$\sqcup_{i=1}^n (D_i, R_i, S_i, N_i, P_i) \stackrel{\text{def}}{=} (D, R, S, N) \text{ where } \begin{cases} D = \bigcup_{i=1}^n (D_i - P_i) \\ R = \bigcup_{i=1}^n (R_i - P_i) \\ S = \bigcup_{i=1}^n (S_i - P_i) \\ N = (\bigcup_{i=1}^n (N_i - P_i)) - (D \cup R \cup S) \end{cases}$$

FIGURE 8. Predicates and least upper bounds

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, occasionally interrupted by top-down traversals when new information must be propagated in one or more branches. If the propagation does not rise an error, then the bottom-up phase is resumed.

In Figure 7 we show the rules that drive the bottom-up working mode. A judgement of the form $e \vdash_{\text{inf}} (D, R, S, N)$ should be read as: from expression e the 4-tuple (D, R, S, N) of marked variables is inferred. A straightforward 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 sets *sharerec* and *shareall* consisting of *all* variables in scope satisfying the respective sharing property. The predicates and least upper bound appearing in the rules [LET_l] and [CASE!_l] are defined in Figure 8.

In Figure 9 we show the top-down checking rules. A judgement of the form $(D, R, S) \vdash_{\text{check}} e$ should be understood that the sets of marked variables D, R, S are correctly propagated down the expression e . The invariant in this case is that the three sets are pairwise disjoint and that their union 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 occurrences 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 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. Oth-

$$\begin{array}{c}
\frac{}{(\mathbf{0}, \mathbf{0}, \mathbf{0}) \vdash_{check} e} \text{ [EMPTY}_C\text{]} \quad \frac{f\bar{a}_i^n @ r \vdash_{inf} (D, R, S, N) \quad R_p = \mathbf{0} \quad \forall a_i \in D_p. (\#j : 1 \leq j \leq n : a_i = a_j) = 1}{(D_p, R_p, S_p) \vdash_{check} f\bar{a}_i^n @ r} \text{ [APP}_C\text{]} \\
\\
\frac{}{(\{x\}, \mathbf{0}, \mathbf{0}) \vdash_{check} x @ r} \text{ [COPY1}_C\text{]} \quad \frac{}{(\mathbf{0}, \{x\}, \mathbf{0}) \vdash_{check} x @ r} \text{ [COPY2}_C\text{]} \quad \frac{}{(\mathbf{0}, \mathbf{0}, \{x\}) \vdash_{check} x @ r} \text{ [COPY3}_C\text{]} \\
\\
\frac{e_1 \vdash_{inf} (D_1, R_1, S_1, N_1) \quad (D_p \cap N_1, R_p \cap N_1, S_p \cap N_1) \vdash_{check} e_1 \quad e_2 \vdash_{inf} (D_2, R_2, S_2, N_2) \quad (D_p \cap N_2, R_p \cap N_2, S_p \cap N_2) \vdash_{check} e_2 \quad L = FV(e_2) \quad C = \begin{cases} \text{shareall}(x_1, e_2) & \text{if } x_1 \in S_2 \\ \text{shareall}(x_1, e_2) - \text{sharerec}(x_1, e_2) & \text{if } x_1 \in D_2 \end{cases} \quad \text{def}((D_p \cup R_p) \cap N_1 \triangleright_C^L (D_p \cup R_p) \cap N_2)}{(D_p, R_p, S_p) \vdash_{check} \mathbf{let } x_1 = e_1 \mathbf{ in } e_2} \text{ [LET}_C\text{]} \\
\\
\frac{\forall i \in \{1..n\}. e_i \vdash_{inf} (D_i, R_i, S_i, N_i) \quad \forall i \in \{1..n\}. P_i = \bigcup_{j=1}^{m_i} \{x_{ij}\} \quad \forall i \in \{1..n\}. Rec_i = \bigcup_{j=1}^{m_i} \{x_{ij} \mid j \in RecPos(C_i)\} \quad \text{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} \quad \forall i \in \{1..n\}. D_{p_i} = \mathbf{0} \quad \forall i \in \{1..n\}. R_{p_i} = \begin{cases} Rec_i & \text{if } x \in D_p \\ \mathbf{0} & \text{otherwise} \end{cases} \quad \forall i \in \{1..n\}. S_{p_i} = \begin{cases} P_i - Rec_i & \text{if } x \in D_p \\ P_i & \text{if } x \in S_p \\ P_i - (R_i \cup S_i) & \text{if } x \in R_p \\ \mathbf{0} & \text{otherwise} \end{cases} \quad \forall i \in \{1..n\}. ((D_p \cup D_{p_i}) \cap N_i, (R_p \cup R_{p_i}) \cap N_i, (S_p \cup S_{p_i}) \cap N_i) \vdash_{check} e_i \quad x \in D_p \cup R_p \cup S_p \Rightarrow \forall i \in \{1..n\}. \text{def}(\text{inh}(\text{type}(x), D_i, R_i, S_i, P_i, Rec_i))}{(D_p, R_p, S_p) \vdash_{check} \mathbf{case } x \mathbf{ of } \overline{C_i \bar{x}_{ij}^{m_i}} \rightarrow e_i^n} \text{ [CASE}_C\text{]} \\
\\
\frac{\forall i \in \{1..n\}. e_i \vdash_{inf} (D_i, R_i, S_i, N_i) \quad \forall i \in \{1..n\}. (D_p \cap N_i, R_p \cap N_i, S_p \cap N_i) \vdash_{check} e_i}{(D_p, R_p, S_p) \vdash_{check} \mathbf{case! } x \mathbf{ of } \overline{C_i \bar{x}_{ij}^{m_i}} \rightarrow e_i^n} \text{ [CASE!}_C\text{]}
\end{array}$$

FIGURE 9. Top-down checking rules

erwise, 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.

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 5. *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. D', R' and S' are pairwise disjoint.

4. $D' \cup R' \cup S' \subseteq N$.

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

Lemma 6. *Let $f \overline{x}_i^n = e$ 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 $x \in \text{var}(e)$, x has got a mark d, s or r .*

Proof. This is an invariant of the \vdash_{check} rules. The final \vdash_{check} forces all don't-know arguments to have an s mark. Then, assuming that every free variable of e has a mark different from n , and by cases on e , it can be easily seen that each recursive call to \vdash_{check} preserves the property that every variable free in a subexpression gets a mark different from n .

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 5, 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} 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} . If $e \vdash_{inf} (D, R, S, N)$ represent the sets inferred during the bottom-up mode, then by Lemma 6, $D' \cup R' \cup S' = N$ must hold.

The next proposition uses these lemmas to establish the correctness of the inference algorithm. As Hindley-Milner types are correctly inferred and our concern now is the correctness of the marks, we use 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.

Proposition 7. *Let us assume that the function declaration $f \overline{x}_i^n = e$ 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. $\Gamma \vdash e' : s'$, and
2. $\forall x \in \text{scope}(e'). \quad x \in D \cup D' \leftrightarrow \Gamma(x) = d \wedge$
 $x \in S \cup S' \leftrightarrow \Gamma(x) = s \wedge$
 $(x \in R \cup R' \vee \exists y \in D \cup D'. x \in \text{sharerec}(y, e') \wedge x \notin D \cup D') \leftrightarrow \Gamma(x) = r$

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

```

-- Binary Tree data type
data Tree a @ rho = Empty @ rho
                  | Node (Tree a @ rho) a (Tree a @ rho) @ rho

-- Insertion in a binary search tree
insertD x Empty! @ r = Node (Empty @ r) x (Empty @ r) @r
insertD x (Node i y d)! @ r
    | x < y = Node (insertD x i @r) y d! @r
    | x == y = Node i! y d! @r
    | x > y = Node i! y (insertD x d @r) @r

-- List concatenation
concatD []! xs @ r = xs
concatD (x:xs)! ys @ r = (x:(concatD xs ys @ r))@r

-- List split
splitD 0 zs! @r = ([@r, zs!}@r
splitD n []! @r = ([@r, []@r}@r
splitD n (y:ys)! @ r = ((y:ys1}@r, ys2}@r
    where (ys1, ys2) = splitD (n-1) ys @r

```

FIGURE 10. Example function definitions related to lists and binary search trees

5 SMALL EXAMPLES

In this section we show some examples. First, we review the example of reversing a list. Given the definition of *revauxD* (Figure 2), initially all parameter positions of *revauxD* are marked as dont-know. Therefore x_1 and xx belong to set N . Besides that, x and ys are marked as safe in the auxiliary expression of the **let** binding since they are used to construct a DS. Combining the results of auxiliary and principal expressions in **let** we obtain that x_1 is not used destructively in function application and hence, a top-down traversal is needed to check that x_1 may be marked as safe. Information from both alternatives in **case!** is gathered in the following sets:

$$\begin{array}{lllllll}
 D_1 = \emptyset & R_1 = \emptyset & S_1 = \{ys\} & N_1 = \emptyset & P_1 = \emptyset & Rec_1 = \emptyset & ([\text{ guard}) \\
 D_2 = \emptyset & R_2 = \emptyset & S_2 = \{x,ys\} & N_2 = \{xx\} & P_2 = \{x,xx\} & Rec_2 = \{xx\} & (x : xx \text{ guard})
 \end{array}$$

The following sets are inferred for the definition: $D = \{xs\}, R = \{\}, S = \{ys\}, N = \{\}$. Consequently, the type signature for *revauxD* is updated: the first position is now destructive and the second one is safe. Since fixpoint has not been reached yet, another bottom-up traversal is needed.

In the second iteration, variables xx and x_1 appearing in function application *revauxD* are respectively inferred as destructive and safe. Variable xs is also given a danger type, since it shares a recursive structure of xx . However, operator $\triangleright_C^{FV(e_2)}$ in let binding is still well defined, as xs is not free in the main expression. At the end of this iteration no variables are marked as dont-know and a fixpoint has been reached. The final sets are $revauxD \mapsto (\{1\}, \emptyset, \{2\}, \emptyset)$.

In Figure 10 we present more examples already written in *Full-Safe*, where constructor patterns are allowed as parameters in function definitions. A mark ! after a pattern means that the corresponding data structure is condemned. This will be translated into *Core-Safe* as a **case!** expression. Region variables in left hand side of definitions are separated from ordinary variables by means of @ symbol.

Function `insertD` inserts an element into a binary search tree. Those nodes corresponding to the path between the root and the new created node are destroyed and the rest of them are reused.

Function `concatD` returns a list resulting from the concatenation of its parameters. The first list is destroyed while the second list is reused.

Finally the function `splitD` divides a list into its n first components and the remaining ones. The nodes of the linked list corresponding to its n first positions are destroyed, while the remaining ones are reused. The types inferred by the algorithm are as follows:

$$\begin{aligned}
\text{insertD} &:: \text{Int} \rightarrow \text{Tree Int!}@p \rightarrow p \rightarrow \text{Tree Int}@p \\
\text{concatD} &:: \forall a.[a]!@p_1 \rightarrow [a]@p_2 \rightarrow p_2 \rightarrow [a]@p_2 \\
\text{splitD} &:: \forall a.\text{Int} \rightarrow [a]!@p \rightarrow p \rightarrow ([a]@p, [a]@p)@p
\end{aligned}$$

REFERENCES

- [1] R. Peña and C. Segura. A First-Order Functional Language for Reasoning about Heap Consumption. In *Proceedings of the 16th International Workshop on Implementation of Functional Languages, IFL'04. Technical Report 0408, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel*, pages 64–80, 2004.
- [2] R. Peña, C. Segura, and M. Montenegro. A Sharing Analysis for SAFE. In *Trends in Functional Programming (Volume 7) Selected Papers of the Seventh Symposium on Trends in Functional Programming, TFP'06. 20 pages. To appear.* Intellect, 2007.