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
C/ Prof. José García Santesmases s/n. 28040 Madrid.
Tel: 91 394 7646 / 7627 / 7625. Fax: 91 394 7529

Abstract. *Safe* is a first-order eager functional language with destructive pattern matching controlled by the programmer. A previously presented type system is used to avoid dangling pointers arising from the inadequate usage of this facility. In this paper we present a type inference algorithm, prove its correctness w.r.t. the type system, describe its implementation and give a number of successfully typed examples.
Keywords: memory management, type-based analysis, type inference.

1 Introduction

Safe[1] [15, 11] was introduced as a research platform for investigating the suitability of functional languages for programming small devices and embedded systems with strict memory requirements. The final aim is to be able to infer —at compile time— safe upper bounds on memory consumption for most *Safe* programs. The compiler produces as target language Java bytecode, so that *Safe* programs can be executed in most mobile devices and web navigators.

In most functional languages memory management is delegated to the runtime system. Fresh heap memory is allocated during program evaluation as long as there is enough free memory available. Garbage collection interrupts program execution in order to copy or mark the live part of the heap so that the rest is considered as free. This does not avoid memory exhaustion if not enough free memory is recovered to continue execution. The main advantage of this approach is that programmers do not have to bother about low level details concerning memory management. Its main disadvantages are:

1. The time delay introduced by garbage collection may prevent the program from providing an answer in a required reaction time.
2. Memory exhaustion may provoke unacceptable personal or economic damage to program users.
3. It is difficult to predict at compile time when garbage collection will take place during execution and consequently also to determine the lifetime of data structures and to reason about memory consumption.

* Work supported by the projects TIN2008-06622-C03-01/TIN (STAMP), S-0505/TIC/0407 (PROMESAS) and the MEC FPU grant AP2006-02154.

[1] http://dalila.sip.ucm.es/safe

These reasons make this memory management unacceptable in small devices where garbage collectors are a burden both in space and in service availability. Programmers of such devices would like both to have more control over memory and to be able to reason about the memory consumption of their programs. Some works have been done in order to perform compile-time garbage collection [7–9], or to detect safe destructive updates of data structures [6, 13]. However, these implicit approaches do not avoid completely the need for a garbage collector.

Another possibility is to use heap regions, which are parts of the heap that are dynamically allocated and deallocated. Many work has been done in order to incorporate regions in functional languages. They were introduced by Tofte and Talpin [17] in MLKit by means of a nested **letregion** construct inferred by the compiler. The drawbacks of nested regions are well-known and they have been discussed in many papers [4]. The main problem is that in practice data structures do not have the nested lifetimes required by the stack-based region discipline. In order to overcome this limitation several mechanisms have been proposed. An extension of Tofte and Talpin's work [3, 16] allows to *reset* all the data structures in a region without deallocating the whole region. The AFL system [1] 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 [4] a comparison of these works is done. In both cases, although it is not required to write in the program the memory commands, a deep knowledge about the hidden mechanism is needed in order to optimize the memory usage. In particular, it is required to write copy functions in the program which are difficult to justify without knowing the annotations inferred by the compiler.

Another more explicit approach is to introduce a language construct to free heap memory. Hofmann and Jost [5] introduce a *match* construct which destroys individual constructor cells than can be reused by the memory management system. This allows the programmer to control the memory consumed by her program and to reason about it. However, this approach gives the programmer the whole responsibility for reusing memory unless garbage collection is used.

Our functional language *Safe* is a semi-explicit approach to memory control which combines regions and a deallocation construct but with a very low effort from the programmer's point of view.

Safe uses implicit regions to destroy garbage. In our language regions are allocated/deallocated by following a stack discipline associated to function calls and returns. Each function call allocates a local working region, which is deallocated when the function returns. The compiler infers which data structures may be allocated in this local region because they are not needed as part of the result of the function. Region management does not add a significant runtime overhead because its related operations run in constant time [14].

In order to overcome the problems related to nested regions, *Safe* also provides the programmer with a construct **case!** to deallocate individual cells of a data structure, so that they can be reused by the memory management system. Regions and explicit destruction are orthogonal mechanisms: we could have de-

struction without regions and the other way around. This combination of explicit destruction and implicit regions is novel in the functional programming field.

Safe's syntax is a first-order subset of Haskell extended with destructive pattern matching. Consequently, programming in *Safe* is straightforward for Haskell programmers. They only have to write a destructive pattern matching when they want the cell to be reused (see the examples in Sec. 2). Programmer controlled destruction may create dangling references as a side effect. For this reason, we have defined a type system [11] guaranteeing that programmer destructions and region management done by the system do not create dangling pointers in the heap. A correct region inference algorithm was also described in [10].

The contribution of this paper is the description of an inference algorithm for the type system presented in [11]. This task is not trivial as the rules in the type system are non-deterministic and additionally some of them are not syntax-directed. We provide a high level description of its implementation, give some examples of its use, and also prove its correctness with respect to the type system. Our algorithm has been fully implemented as a part of our *Safe* compiler and has an average time cost near to $\Theta(n^2)$ for each function definition, where n is the size of its abstract syntax tree (see Sec. 4).

In Sec. 2 we summarize the language. The type system is presented in Sec. 3 and the corresponding inference algorithm is explained and proven correct in Sec. 4. Section 5 shows some examples whose types have been successfully inferred. Finally, Sec. 6 compares this work with related analyses in other languages with memory management facilities.

2 Summary of *Safe*

Safe is a first-order polymorphic functional language with some facilities to manage memory. These are destructive pattern matching, copy and reuse of data structures. We explain them with examples below.

Destructive pattern matching, denoted by ! or a **case!** expression, deallocates the cell corresponding to the outermost constructor. In this case we say that the data structure 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. Using recursion the recursive spine of the first list is deallocated:

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

This version of appending needs constant additional heap space (a cell is destroyed and another one is created at each call), while the usual version needs additional linear heap space. The fact that the first list is lost must be remembered by the programmer as he will not be able to use it anymore in the program. This is reflected in the type of the function: `concatD :: [a]! -> [a] -> [a]`.

Data structures may also be copied by using a copy expression (denoted by @). Only the recursive spine of the structure is copied, while the elements are shared with the old one. This allows more control over sharing of data structures. In the following function

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

Fig. 1. *Core-Safe* language definition

```
concat []     ys  = ys @
concat (x:xs) ys  = x : concat xs ys
```

the resulting list only shares the elements with the input lists. We could safely destroy this list while preserving the original ones.

When a data structure is condemned by a destructive pattern matching, its recursive children are also condemned in order to avoid dangling pointers (see the type system of Sec. 3). This means that they cannot be returned as part of the result of a function even when they are not explicitly destroyed. This would force us to copy those data substructures if we want them as part of the result, which would be costly. As an example, consider the following destructive tail function: `tailD (x:xs)! = xs@`. The sublist `xs` is condemned so it cannot be returned as result of `tailD`. We need to copy it if we want to keep safety. In order to avoid this costly copies the language offers a safe reuse operator ! which allows to turn a condemned data structure into a safe one so that it can be part of the result of a function. The original reference is no longer accessible in order to keep safety. In the previous example we would write `tailD (x:xs)! = xs!`.

We show another example whose type is successfully inferred. The following function 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!
```

In the first guard the cell just destroyed must be built again since `lt` and `rt` are condemned; they must be reused in order to be part of the result.

2.1 *Core-Safe*

Full-Safe is desugared into an intermediate language called *Core-Safe* where regions are explicit. However, regions can be completely ignored in this paper, as the inference algorithm explained here only concerns destruction. We just show them for completeness. In Fig. 1 we show the syntax of *Core-Safe*. A program

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

Fig. 2. Type expressions

prog is a sequence of possibly recursive polymorphic data and function definitions followed by a main expression e, using 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 in 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 local working region may be used. As an example, let us consider the *Core-Safe* code of the function `concatD`:

$$
\begin{array}{l}
\mathit{concatD}\ \mathit{zs}\ \mathit{ys}\ @\ r = \textbf{case!}\ \mathit{zs}\ \textbf{of} \\
\qquad\qquad [\,] \rightarrow \mathit{ys} \\
\qquad\qquad (x : \mathit{xs}) \rightarrow \textbf{let}\ x_1 = \mathit{concatD}\ \mathit{xs}\ \mathit{ys}\ @\ r\ \textbf{in} \\
\qquad\qquad\qquad\qquad \textbf{let}\ x_2 = (x : x_1)@r\ \textbf{in}\ x_2
\end{array}
$$

In this case the only regions involved are those of the input lists and the output region r where the result is built. The local region of each `concatD` call remains empty during its execution, since nothing is built there.

3 Safe Type System

In this section we briefly describe a polymorphic type system with algebraic data types for programming in a safe way when using the destruction facilities offered by the language (see [11] for more details). 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 and are useful to control the side-effects of the destructions. However, 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 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 them.

– ***In-danger* types (r):** This is a DS sharing a recursive descendant of a condemned DS, so it can potentially 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.

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 $gen(tf, \Gamma)$ 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 $utype?(t, t')$ is true when the underlying Hindley-Milner types of t and t' are the same.

In Fig. 3, the rules for typing expressions are shown. Function $sharerec(x, e)$ 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 [15].

An invariant of the type system tells 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 $type(y)$ represents the Hindley-Milner type inferred for variable y[2].

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

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

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

$$\frac{R = sharerec(x, x!) - \{x\} \quad \Gamma_R = \{y : danger(type(y)) |\ y \in R\}}{\Gamma_R + [x : T!@\rho] \vdash x! : T@\rho}\ [\text{REUSE}] \qquad \frac{\Gamma_1 \vdash e_1 : s_1 \quad \Gamma_2 + [x_1 : \tau_1] \vdash e_2 : s \quad utype?(\tau_1, s_1)}{\Gamma_1 \rhd^{fv(e_2)} \Gamma_2 \vdash \textbf{let } x_1 = e_1 \textbf{ in } e_2 : s}\ [\text{LET}]$$

$$\frac{\Gamma_1 \geq_{x@r} [x : T@\rho', r : \rho]}{\Gamma_1 \vdash x@r : T\ @\rho}\ [\text{COPY}] \qquad \frac{\Sigma(C) = \sigma \quad \overline{s_i}^n \to \rho \to T\ @\overline{\rho}^m \trianglelefteq \sigma \quad \Gamma = \bigoplus_{i=1}^n [a_i : s_i] + [r : \rho]}{\Gamma \vdash C\ \overline{a_i}^n @ r : T\ @\overline{\rho}^m}\ [\text{CONS}]$$

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

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

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

Fig. 3. Type rules for expressions

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

Rule [LET] controls the intermediate data structures, that may be safe, condemned or in-danger in the main expression (τ covers the three cases). 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. $y \notin fv(e_2)$), 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 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 Sec. 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) = xs@`. The type system can give f type $[a] \to [a]$ but also the type $[a]! \to [a]$. 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

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

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

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

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

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

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

Fig. 4. Bottom-up inference rules

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

Fig. 5. Predicates *inh* and *inh*!

$$def(\sqcup_{i=1}^{n}(D_i,R_i,S_i,N_i,P_i)) \equiv \forall i,j \in \{1..n\}\,.\, i \neq j \Rightarrow \begin{array}{l}(D_i - P_i) \cap (R_j - P_j) = \emptyset \wedge \\ (D_i - P_i) \cap (S_j - P_j) = \emptyset \wedge (R_i - P_i) \cap (S_j - P_j) = \emptyset\end{array}$$

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

Fig. 6. Least upper bound definitions

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, occasionally 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 Fig. 4 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 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 Figs. 5 and 6.

In Fig. 7 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 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 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

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

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

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

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

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

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

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

Fig. 7. Top-down checking rules

(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.

In rule [LET_I], 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 function *RecPos* returns the recursive parameter positions of a given data constructor. 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.
3. A variable belongs to R''_i or R'''_i (see 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. The set R'''_i contains those variables that do not belong to any of the (D_i, R_i, S_i, N_i) sets corresponding to the i-th **case** branch, but they share a recursive descendant of a variable being propagated top-down through this branch as condemned. In [12] a few explanatory examples on these sets are given.

The $\vdash_{check}$ rules capture the same verifications as the $\vdash_{inf}$ rules, but in a top-down fashion. See [12] for more details about R'_p in [LET$_C$].

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 traversal will eventually get a d, r or s mark (see [12] 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. However in most cases it is near to $\Theta(n^2)$, corresponding to a single bottom-up traversal and two fixpoint iterations.

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. By induction on the corresponding $\vdash_{inf}$ and $\vdash_{check}$ derivations [12]. □

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. By structural induction on e' [12]. □

5 Small Examples

In this section we show some examples. Firstly, we review the example of appending two lists (*Core-Safe* code of `concatD` in Sec. 2). 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. The variables x and x_1 get an s mark since they are used to build a DS. In addition to this, x_2 is returned as the function's result, so it gets and s mark. 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* without regions is $\forall a.[a]! \to [a] \to [a]$.

The type of function *insertD*, defined in Sec. 2, is *Int* → *Tree Int!* → *Tree Int*. Other successfully typed destructive functions (whose code is not shown) are the following for splitting a list and for inserting an element in an ordered list:

$$splitD :: \forall a.Int \to [a]! \to ([a], [a]) \qquad insertLD :: \forall a.[a]! \to a \to [a]$$

6 Related Work

Our safety type system has some characteristics of linear types (see [18] 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 [13] or detecting program sites where values could be safely deallocated [9]. The work closest to *Safe*'s type system is [2], where the authors present 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 similarities 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 similarities: (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: $(1, 2, 2)$, $(1, 2, 3)$ and $(2, 2, 2)$. The deep reason is our more precise sharing information and the new in-danger type. 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:

   let $x = z : [\,]$ **in case!** x **of** $\ldots$ **case** z **of** $\ldots$
   let $x = z : [\,]$ **in case!** x **of** $\ldots z$

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

Summarising our contribution, we have developed an inference algorithm for safe destruction which improves on previous attempts on this area, has a low cost, and can be applied to other functional languages smilar to *Safe* (i.e. eager and first-order). In particular, Hofmann and Jost's language [5] could benefit from the work described here.

References

1. 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, 1995.
2. D. Aspinall, M. Hofmann, and M. Konečný. A Type System with Usage Aspects. *Journal of Functional Programming*, 18(2):141–178, 2008.
3. 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, 1996.
4. 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, 2001.
5. M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-order Functional Programs. In *POPL'03*, pages 185–197. ACM, 2003.
6. P. Hudak. A Semantic Model of Reference Counting and its Abstraction. In *Lisp and Functional Programming Conference*, pages 351–363. ACM Press, 1986.
7. K. Inoue, H. Seki, and H. Yagi. Analysis of Functional Programs to Detect Run-Time Garbage Cells. *ACM TOPLAS*, 10(4):555–578, 1988.
8. S. B. Jones and D. Le Metayer. Compile Time Garbage Collection by Sharing Analysis. In *FPCA'89*, pages 54–74. ACM Press, 1989.
9. N. Kobayashi. Quasi-linear Types. In *POPL'99*, pages 29–42. ACM Press, 1999.
10. M. Montenegro, R. Peña, and C. Segura. A Simple Region Inference Algorithm for a First-Order Functional Language. In *TFP'08*, pages 194–208, 2008.
11. M. Montenegro, R. Peña, and C. Segura. A Type System for Safe Memory Management and its Proof of Correctness. In *PPDP'08*, pages 152–162. ACM, 2008.
12. M. Montenegro, R. Peña, and C. Segura. An Inference Algorithm for Guaranteeing Safe Destruction (extended version). Technical report, SIC-8-08. UCM, 2008. http://federwin.sip.ucm.es/sic/investigacion/publicaciones/pdfs/SIC-8-08.pdf.
13. M. Odersky. Observers for Linear Types. In *ESOP'92, LNCS 582*, pages 390–407. Springer-Verlag, 1992.
14. R. Peña and C. Segura. Formally Deriving a Compiler for SAFE. In *IFL'06*, pages 429–446, 2006.
15. R. Peña, C. Segura, and M. Montenegro. A Sharing Analysis for Safe. In *Trends in Functional Programming (Vol. 7)*, pages 109–128. Intellect, 2007.
16. 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.
17. M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.
18. 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.