Generación de código con certificado asociado

Ricardo Peña Marí

Catedrático de Lenguajes y Sistemas Informáticos
Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

F. Informática S. Sebastián, enero 2008

Motivation (1)

- Proof Carrying Code (PCC) is an emerging research trend aimed at endowing programs with certificates guaranteeing that certain properties are satisfied. These certificates are in fact mathematical correctness proofs that can be automatically checked by appropriate tools.
- PCC is developing quickly, specially in the area of mobile code, where consumers of (normally untrusted) code wish to ensure that their safety policy is fulfilled, before running the code in their machines.
- Our project SELF (TIN2004-07943-C04-04) is developing SAFE, a functional language targeted at mobile applications with limited resources, and aimed at automatically generating certificates about the following properties:
  1. Absence of dangling pointers
  2. Termination
  3. Bounded heap and stack
- SAFE may use an explicit construction to liberate memory and does not need a garbage collector.

Motivation (2)

FUNCTIONAL LANGUAGES
Implicit Approach (Garbage Collection)

High-level programming vs. Time delay + No reasoning about heap needs

IMPERATIVE LANGUAGES
Explicit Approach (Pointers)

Control over memory usage vs. Error-prone programs (dangling references, undesired sharing)

SAFE
Semi-explicit Approach (Destructive Pattern Matching)
No garbage collection
Reasoning about the heap needed by the program
"Error-free" programs

Conventional version of mergesort

$$
\begin{array}{lll}
split\ 0\ xs & = & ([\,], xs) \\
split\ n\ [\,] & = & ([\,], [\,]) \\
split\ n\ (x : xs) & = & (x : xs_1, xs_2) \\
 & & \quad \textbf{where}\ (xs_1, xs_2) = split\ (n-1)\ xs
\end{array}
$$

$$
\begin{array}{lll}
merge\ [\,]\ ys & = & ys \\
merge\ xs\ [\,] & = & xs \\
merge\ (x : xs)\ (y : ys) & & \\
\quad \mid x \leq y & = & x : merge\ xs\ (y : ys) \\
\quad \mid otherwise & = & y : merge\ (x : xs)\ ys
\end{array}
$$

$$
\begin{array}{lll}
msort\ xs & & \\
\quad \mid n \leq 1 & = & xs \\
\quad \mid otherwise & = & merge\ (msort\ xs_1\ )\ (msort\ xs_2\ ) \\
\quad \textbf{where} & (xs_1, xs_2) & = split\ (n\ `div`\ 2)\ \ xs \\
 & n & = length\ xs
\end{array}
$$

Allocated memory in $\Theta(n \log n)$ and resident memory in $\Theta(n)$

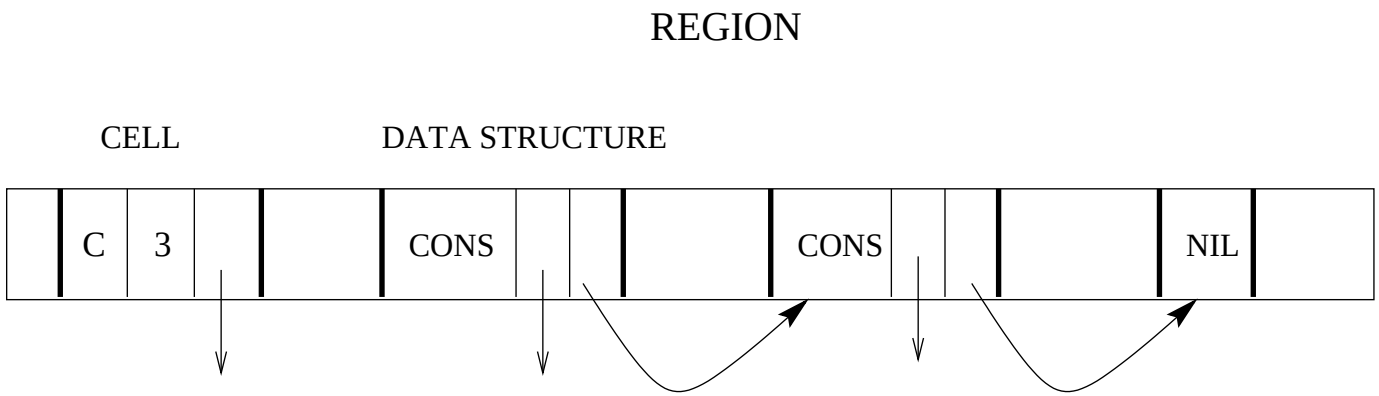
SAFE version of mergesort

$$
\begin{array}{l}
splitD :: \forall a, \rho.Int \to [a]!@\rho \to \rho \to ([a]@\rho, [a]@\rho)@\rho \\
splitD\ 0\ xs! \quad\quad = ([\,], xs!) \\
splitD\ n\ [\,]! \quad\quad = ([\,], [\,]) \\
splitD\ n\ (x : xs)! = (x : xs_1, xs_2) \\
\qquad\qquad\qquad\qquad \textbf{where}\ (xs_1, xs_2) = splitD\ (n-1)\ xs \\
mergeD :: \forall a, \rho.[a]!@\rho \to [a]!@\rho \to \rho \to [a]@\rho \\
mergeD\ [\,]!\ ys! \quad\quad\quad = ys! \\
mergeD\ xs!\ [\,]! \quad\quad\quad = xs! \\
mergeD\ (x : xs)!\ (y : ys)! \\
\qquad\qquad \mid x \leq y \quad\quad = x : mergeD\ xs\ (y : ys!) \\
\qquad\qquad \mid otherwise = y : mergeD\ (x : xs!)\ ys \\
msortD :: \forall a, \rho.[a]!@\rho \to \rho \to [a]@\rho \\
msortD\ xs \\
\quad \mid n \leq 1 \quad\quad = xs! \\
\quad \mid otherwise = mergeD\ (msortD\ xs_1\ )\ (msortD\ xs_2\ ) \\
\quad\ \textbf{where}\ \ (xs_1, xs_2) = splitD\ (n\ `div`\ 2)\ \ xs \\
\qquad\qquad\ \ n \quad\quad\quad\quad = length\ xs
\end{array}
$$

Resident memory in $\Theta(n)$. Additional heap memory in $\Theta(1)$

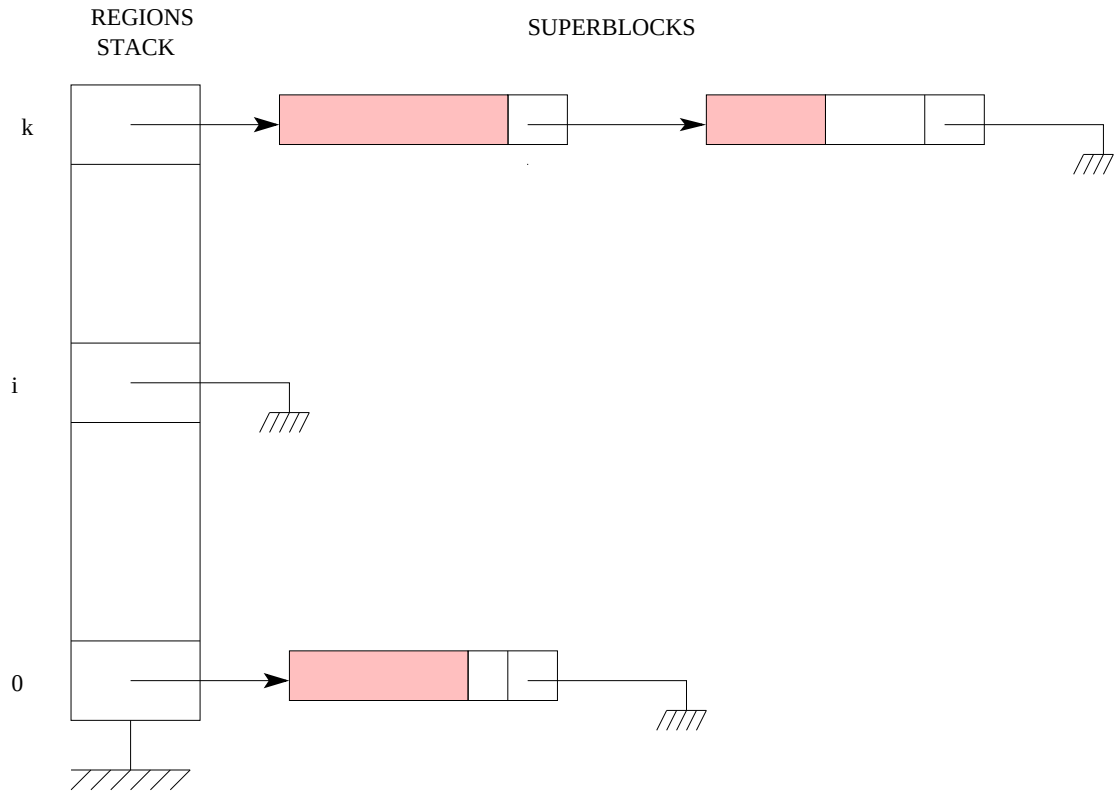
Index

Language Design: memory allocation scheme



- A data structure (DS) completely resides in one region.
- Two DSs can be one of them part of the other, or they can share a third DS.
- The basic values —integers, booleans, etc.— do not allocate cells in regions. They live inside the cells of DSs, or in the stack.

Memory Management



Action	**Cost in time**
Region creation	Constant
Region disposal	Constant
Cell allocation	Constant
Cell disposal	Constant

Core SAFE normalised syntax

$prog$	$\longrightarrow$	$dec_1; \ldots; dec_n; e$	{program}
dec	$\longrightarrow$	$f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e$	{single-recursive, polymorphic function definition}
			{atom}
a	$\longrightarrow$	c	{atomic constant}
		$\mid x$	{variable}
			{expression}
e	$\longrightarrow$	a	{atom}
		$\mid x@r$	{data structure copying}
		$\mid x!$	{data structure reusing}
		$\mid f\ \overline{a_i}^n @\ \overline{r_j}^m$	{function application}
		$\mid$ **let** $x_1 = be$ **in** e	{non-recursive, monomorphic}
		$\mid$ **case** x **of** $\overline{alt_i}^n$	{read-only case}
		$\mid$ **case!** x **of** $\overline{alt_i}^n$	{destructive case}
alt	$\longrightarrow$	$C\ \overline{x_i}^n \longrightarrow e$	{case alternative}
			{binding expression}
be	$\longrightarrow$	$C\ \overline{a_i}^n @ r$	{constructor application}
		$\mid e$	{normal expression}

Example: List inversion

Non-destructive

$$revF :: \forall a, \rho_1, \rho_2.[a]@\rho_1 \to \rho_2 \to [a]@\rho_2$$

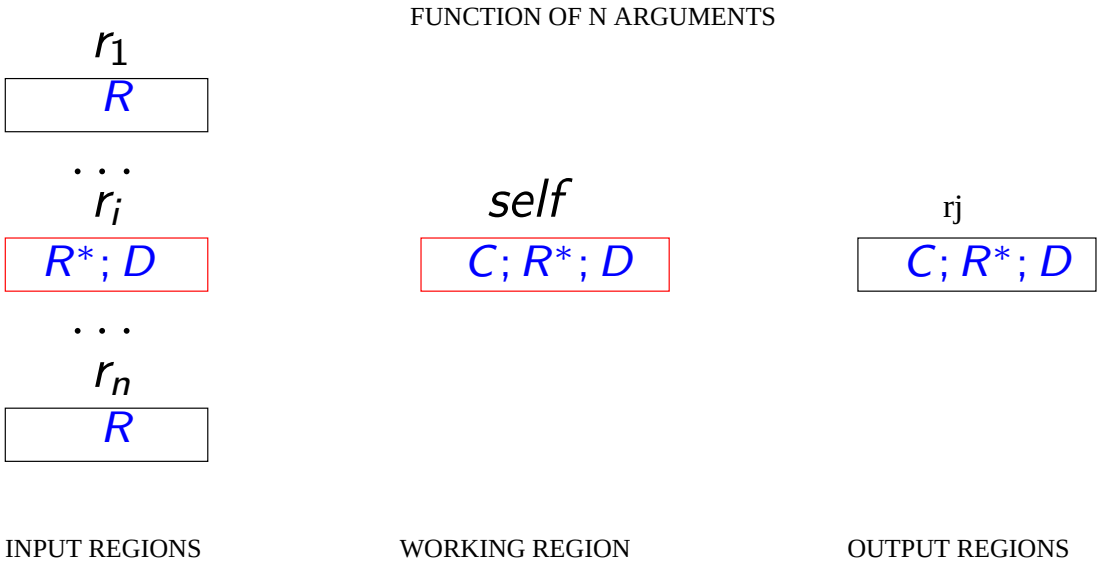
$$\begin{aligned}
&revF\ xs\ @r = \textbf{let}\ ys' = [\,]@r\ \textbf{in}\ (revauxF\ xs\ ys')@r\\
&revauxF :: \forall a, \rho_1, \rho_2.[a]@\rho_1 \to [a]@\rho_2 \to \rho_2 \to [a]@\rho_2\\
&revauxF\ xs\ ys\ @r = \textbf{case}\ xs\ \textbf{of}\\
&\qquad\qquad [\,] \to ys\\
&\qquad\qquad x : xx \to\ \textbf{let}\ ys' = x : ys@r\ \textbf{in}\\
&\qquad\qquad\qquad\qquad (revauxF\ xx\ ys')@r
\end{aligned}$$

Destructive (in-place inversion)

$$revD :: \forall a, \rho_1, \rho_2.[a]!@\rho_1 \to \rho_2 \to [a]@\rho_2$$

$$\begin{aligned}
&revD\ xs\ @r = \textbf{let}\ ys' = [\,]@r\ \textbf{in}\ (revauxD\ xs\ ys')@r\\
&revauxD\ xs\ ys\ @r = \textbf{case!}\ xs\ \textbf{of}\\
&\qquad\qquad [\,] \to ys\\
&\qquad\qquad x : xx \to\ \textbf{let}\ ys' = x : ys@r\ \textbf{in}\\
&\qquad\qquad\qquad\qquad (revauxF\ xx\ ys')@r
\end{aligned}$$

Region usage



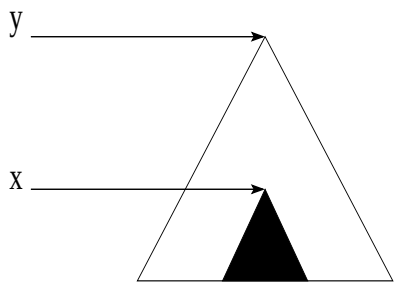
- A function may only read a DS which is a read-only parameter R.
- A function may read (before destroying it), and must destroy, a DS which is a condemned parameter $R^*; D$.
- A function may construct, read, or destroy DSs, in either its output regions or its (single) working region $C; R^*; D$.

Region inference

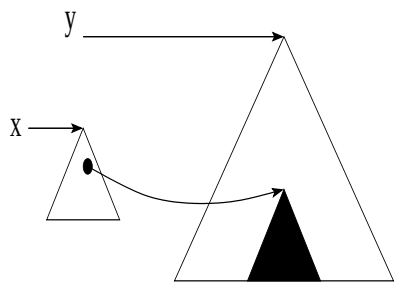
- Region inference takes place as a by-product of the Hindley-Milner type inference phase of the compiler.
- Program region variables r are mandatory at constructions $C\ \overline{a_i}^n@r$ and at function application $f\ \overline{a_i}^n@\ \overline{r_j}^m$ for known functions f. The compiler assigns to them fresh region types.
- The algorithm collects in three different sets,
  1. the region types of the mandatory region variables,
  2. the region types of the data structure produced by the function, and
  3. the region types of the input data structures.
- By solving a collection of restrictions, it takes the following decisions:
  1. Whether zero or more output regions are needed by the function
  2. For each program region variable r appearing in the text, it decides whether it must be assigned to an output region or to the self region.
- The algorithm never fails and it maximises the number of constructions done in the self region. This implies that as much garbage as possible will be collected when the function returns.

Sharing Relations

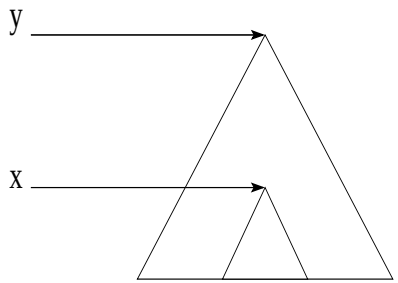
$x \lessdot\!\sim y$ $(SubR)$



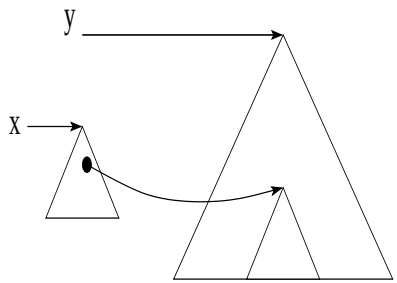
$x \triangle\!\!\sim y$ (ShR)



$x \lessdot y$ (Sub)



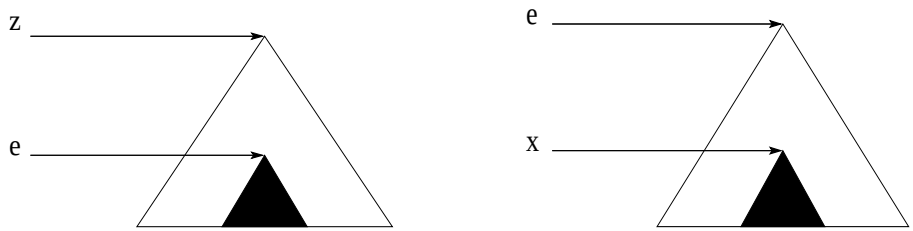
$x \triangle y$ (Sh)



Sharing Analysis

- The interpretation traverses top-down the expression, accumulating the sharing relations (R):

$$S\ [\![e]\!]\ SubR'\ ShR'\ Sub'\ Sh'\ \rho = (SubRP, ShRP, SubP, SubR, ShR, Sub, Sh)$$



- Function signatures are kept in environment ρ:

$$(\{Int\}, \{Int\}, \{Int\}, \{Int\}, \{Int\}, \{Int\}, \{Int\})$$

- Sets $sharerec(x_1, e)$ and $shareall(x_1, e)$ are obtained from a global analysis of the whole program which annotates expression e with inherited information. They are respectively sets $ShR'(x_1)$ and $Sh'(x_1)$.

Examples

- Reverse: the inverse list only shares the elements with the original list:

$$\rho(\mathit{revauxD}) = (\{2\},\{2\},\{2\},\{2\},\{2\},\{2\},\{1,2\})$$

$$\rho(\mathit{revD}) = (\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\{1\})$$

- Tree insert: the new tree shares with the original tree both the elements and the recursive parts that are not on the path from the root to the inserted element:

$$\begin{array}{l}
\mathit{insertD} :: \forall a, \rho.\ a \to \mathit{Tree}\ a!@\rho \to \rho \to \mathit{Tree}\ a@\rho \\
\mathit{insertD}\ x\ \mathit{Empty}! = \mathit{Node}\ \mathit{Empty}\ x\ \mathit{Empty} \\
\mathit{insertD}\ x\ (\mathit{Node}\ l\ y\ r)! \\
\qquad\qquad \mid x < y \quad = \mathit{Node}\ (\mathit{insertD}\ x\ l)\ y\ r! \\
\qquad\qquad \mid x == y = \mathit{Node}\ l!\ y\ r! \\
\qquad\qquad \mid x > y \quad = \mathit{Node}\ l!\ y\ (\mathit{insertD}\ x\ r)
\end{array}$$

$$\rho(\mathit{insertD}) = (\emptyset,\{1,2\},\emptyset,\emptyset,\{2\},\{1\},\{1,2\})$$

Type Expressions

-- Polymorphic, with region type variables

$revF :: \forall a, \rho_1, \rho_2.[a]@\rho_1 \rightarrow \rho_2 \rightarrow [a]@\rho_2$
$insertF :: \forall a, \rho.a \rightarrow Tree\ a@\rho \rightarrow \rho \rightarrow Tree\ a@\rho$

-- Condemned parameters are reflected in the type

$revD :: \forall a, \rho_1, \rho_2.[a]!@\rho_1 \rightarrow \rho_2 \rightarrow [a]@\rho_2$
$insertD :: \forall a, \rho.a \rightarrow Tree\ a!@\rho \rightarrow \rho \rightarrow Tree\ a@\rho$

-- A data structure completely resides in one region

$[\,] : \forall a, \rho.\rho \rightarrow [a]@\rho$
$(:) : \forall a, \rho.a \rightarrow [a]@\rho \rightarrow \rho \rightarrow [a]@\rho$
$Empty : \forall a, \rho : \rho \rightarrow Tree\ a@\rho$
$Node : \forall a, \rho.Tree\ a@\rho \rightarrow a \rightarrow Tree\ a@\rho \rightarrow \rho \rightarrow Tree\ a@\rho$

Rules of the Type System (1)

$$\frac{P = \{x_i \mid t_i = T_i\ \overline{s_{ij}}^k @\rho_i, i = 1, \ldots, n\}, \quad \text{fresh } \rho', \quad \rho' \notin \mathit{regions}(s) \qquad \Gamma^P + \overline{[x_i : t_i]}^n + [r : \rho] + [\mathit{self} : \rho'] + [f : \overline{t_i}^n \to \rho \to s] \vdash e : s}{\{\Gamma^\emptyset\}\ \ f\ \overline{x_i}^n\ r = e\ \ \{\Gamma^\emptyset + [f : \mathit{gen}(\overline{t_i}^n \to \rho \to s, \Gamma)]\}}\ [\text{FUNB}]$$

- DSs may only be constructed in region *self* and in the output region.
- The result of the function is contained in visible regions.
- P collects the read-only parameters.

$$\frac{\Gamma^P(x) = d \quad R = \mathit{sharerec}(x, x!) - \{x\} \qquad \Gamma_R = \{y : \mathit{danger}(\mathit{type}(y)) \mid y \in R\}}{\Gamma_R + \Gamma^P \vdash x! : \overline{d}}\ [\text{REUSE}]$$

- Condemned DSs can be reused by changing its type, but they and their *sharerec* may not be accessed again.

Rules of the Type System (2)

$$\frac{\Gamma_1^P \vdash e_1 : s_1 \qquad \Gamma_2^P + [x_1 : d_1] \vdash e : s \qquad d_1 = \overline{s_1} \qquad P \cap R = \emptyset}{\Gamma_1^P \rhd^{fv(e)} \Gamma_2^P \vdash \textbf{let } x_1 = e_1 \textbf{ in } e : s} \text{ [LET2]}$$

Incorrect typings

let $x =$ **case**! xs **of** $\ldots$ **in** $\ldots xs \ldots$	A variable destructively used in e_1 may not be referenced in e
let $ys = (2 : xs)@self$ **in** **let** $zs = (3 : xs)@self$ **in** **case**! zs **of** $\ldots$	Variables sharing contents with $ys : s$ may not be destroyed in e
let $t = (Node\ i\ x\ d)@r$ **in** **case**! t **of** $\ldots i \ldots$	A recursive substructure of a destroyed DS may not be referenced in e
let $xs = ys$ **in case**! xs **of** $\ldots$	$(ys \in P)$, A read-only parameter may not be destroyed

Rules of the Type System (3)

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

Incorrect typings

case! t **of**
  $Node\ i\ x\ d \to \ldots t \ldots$

A destroyed DS and its *sharerec* may not be referenced again

Correct typings

case! t **of**
  $Node\ i\ x\ d \to$ **case!** i **of**
    $Empty \to d!$

The substructures of a destroyed DS inherit the condemned property and may be either destroyed or reused

case t **of**
  $Node\ i\ x\ d \to$ **case!** t **of** $\ldots$

A condemned DS may be read before being destroyed

Safe Types Inference Algorithm

- The first phase is a Hindley-Milner inference algorithm with some added complications arising from typing region variables and doing region inference.
- Next, the abstract syntax tree (AST) is traversed bottom-up collecting variables names in four sets. The rules $e \vdash_{inf} (D, R, S, N)$ classify variables in scope in e respectively as condemned, in-danger, safe and don't-know.

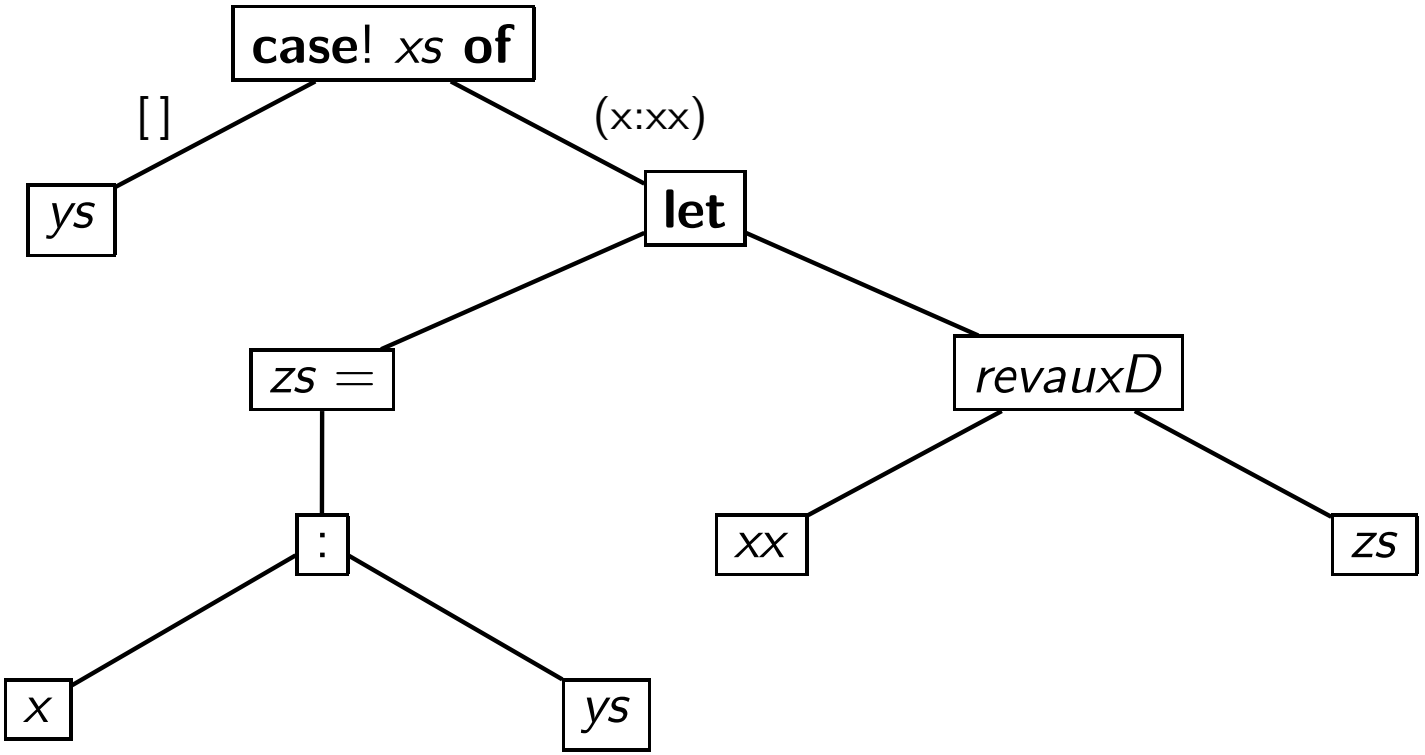
Safe Types Inference Algorithm

- The first phase is a Hindley-Milner inference algorithm with some added complications arising from typing region variables and doing region inference.
- Next, the abstract syntax tree (AST) is traversed bottom-up collecting variables names in four sets. The rules $e \vdash_{inf} (D, R, S, N)$ classify variables in scope in e respectively as condemned, in-danger, safe and don't-know.
- At nodes such as **let** and **case**, the information coming from different subexpressions is checked for consistency, so detecting possible typing errors.
- Also, some variables classified as don't-know in one branch may have acquired a different mark in another branch. This information is propagated top-down by the rules $(D, R, S) \vdash_{check} e$ in order to check that the new marks are consistent in an expression e.

Safe Types Inference Algorithm

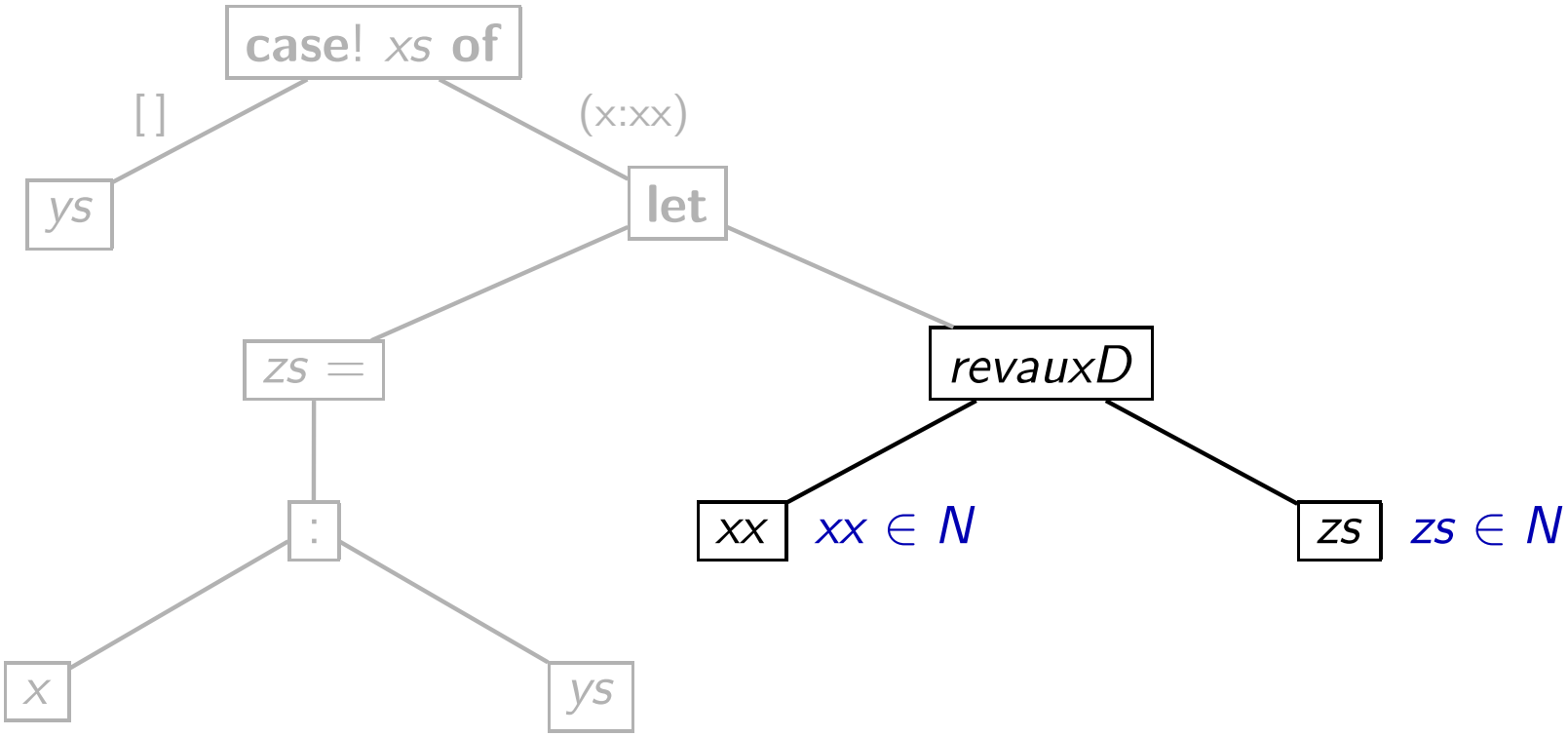
- The first phase is a Hindley-Milner inference algorithm with some added complications arising from typing region variables and doing region inference.
- Next, the abstract syntax tree (AST) is traversed bottom-up collecting variables names in four sets. The rules $e \vdash_{inf} (D, R, S, N)$ classify variables in scope in e respectively as condemned, in-danger, safe and don't-know.
- At nodes such as **let** and **case**, the information coming from different subexpressions is checked for consistency, so detecting possible typing errors.
- Also, some variables classified as don't-know in one branch may have acquired a different mark in another branch. This information is propagated top-down by the rules $(D, R, S) \vdash_{check} e$ in order to check that the new marks are consistent in an expression e.
- So, the algorithm oscillates between the bottom-up and the top-down modes. Also, a fix point computation is done at each recursive function definition. Its arguments are initially classified as don't-know and its body analysed until all argument marks stabilise. In-danger and don't-know arguments are not allowed in a valid stable type.

Example:*revauxD* - First iteration



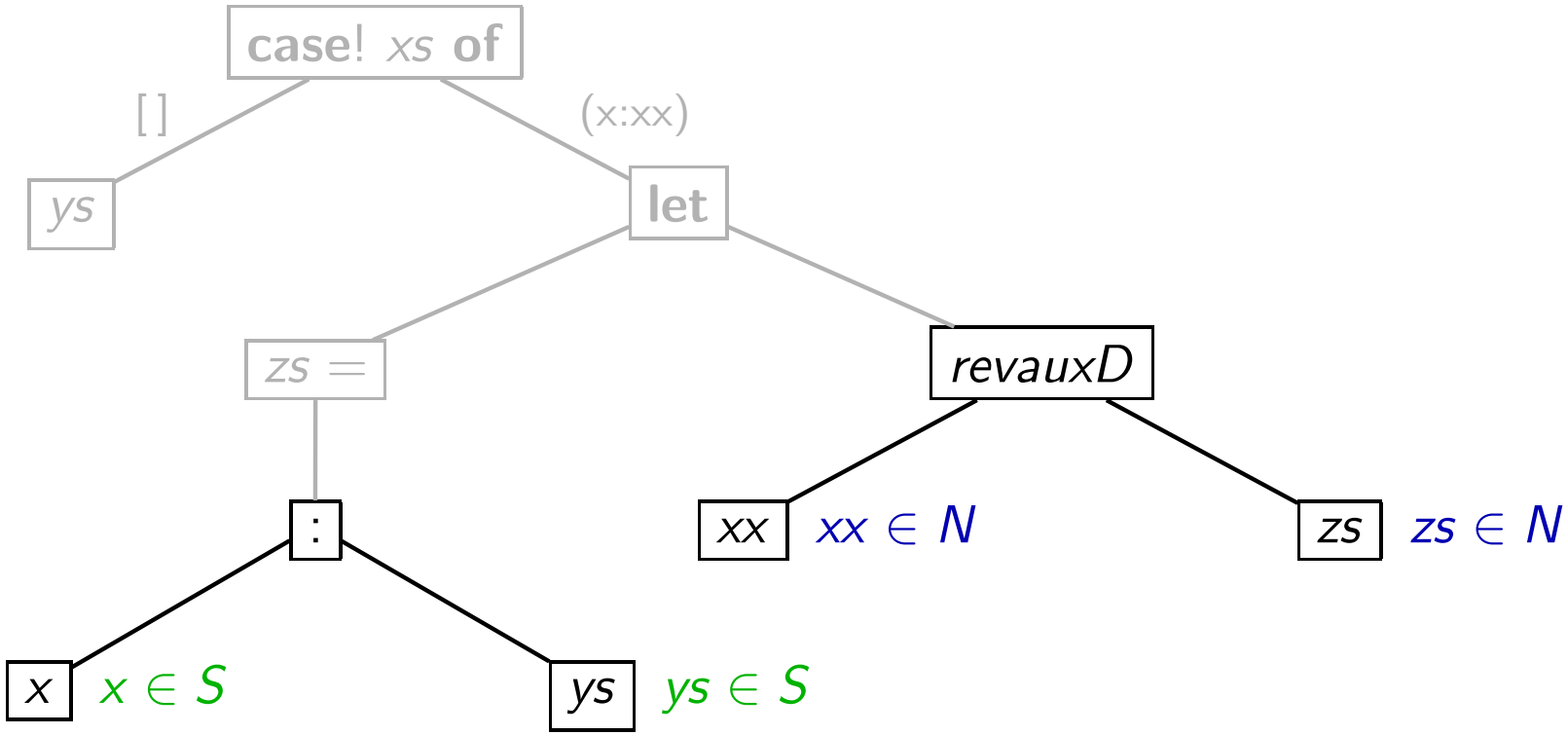
- We review the example of reversing a list. The definition of *revAuxD* is represented by means of its abstract syntax tree.

Example:*revauxD* - First iteration



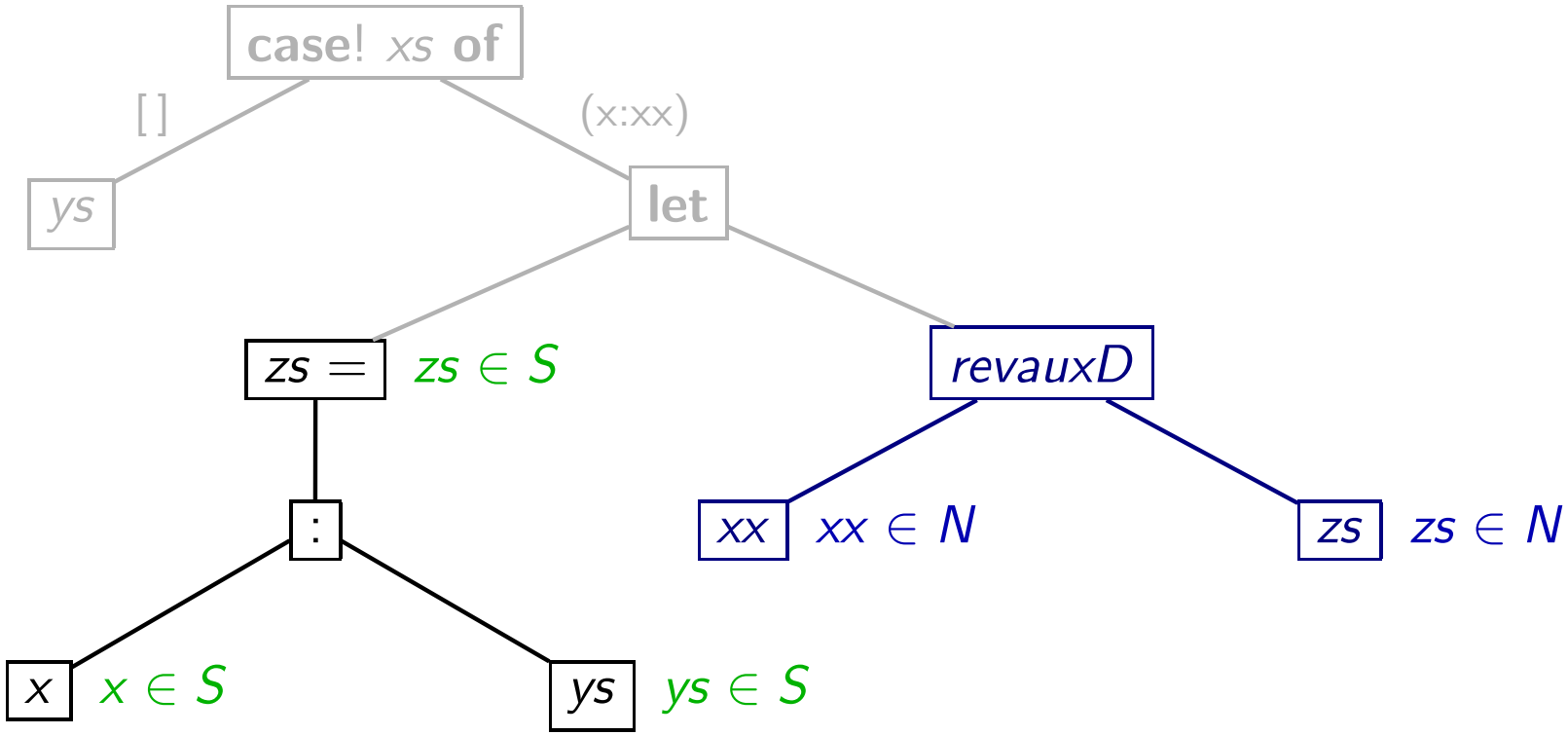
- Initially all parameter positions are marked as don't-know.
- Therefore, the actual parameters *zs* and *xx* in function application belong to the set *N*.

Example:*revauxD* - First iteration



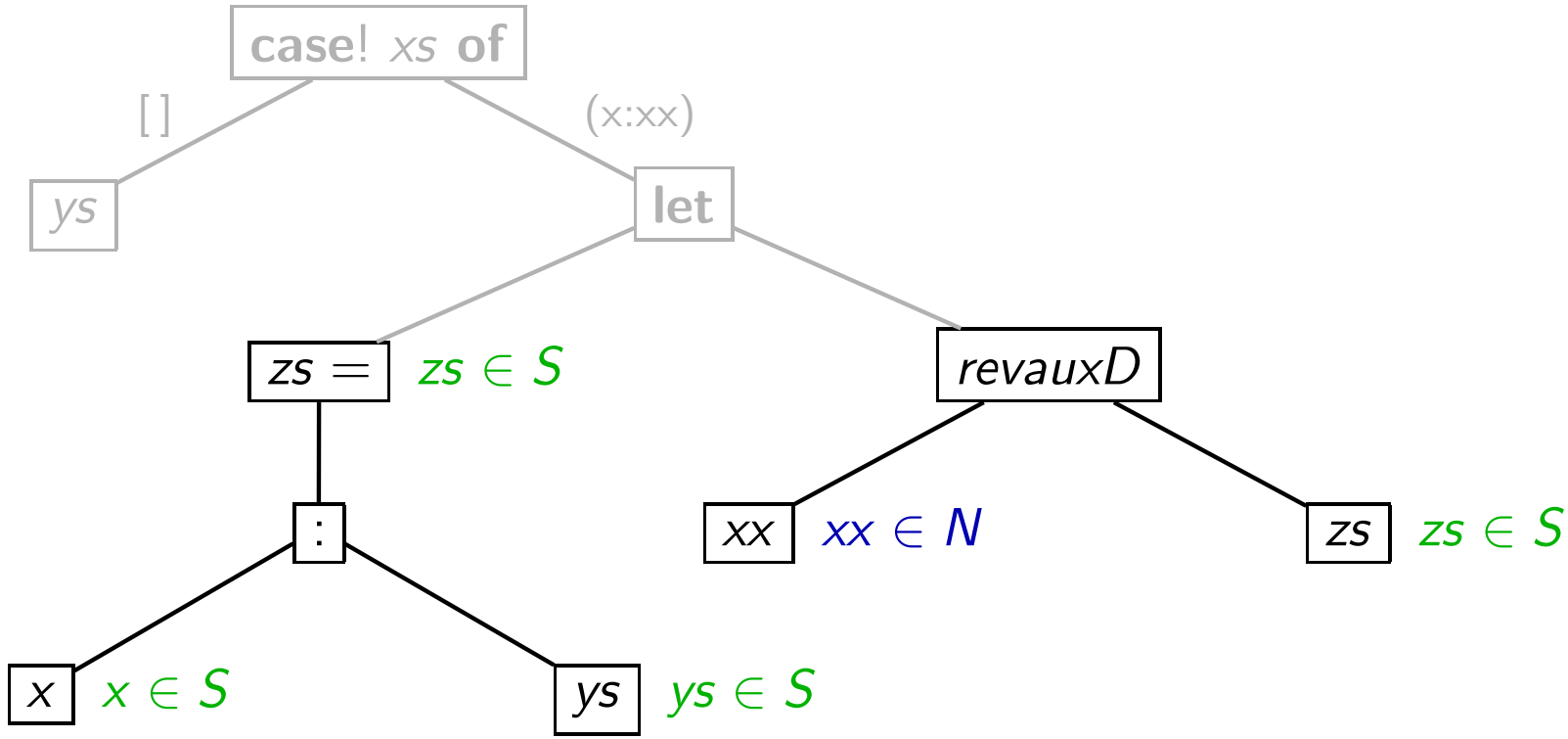
- Variables x and ys are used to construct a data structure. They are marked as safe.

Example:*revauxD* - First iteration



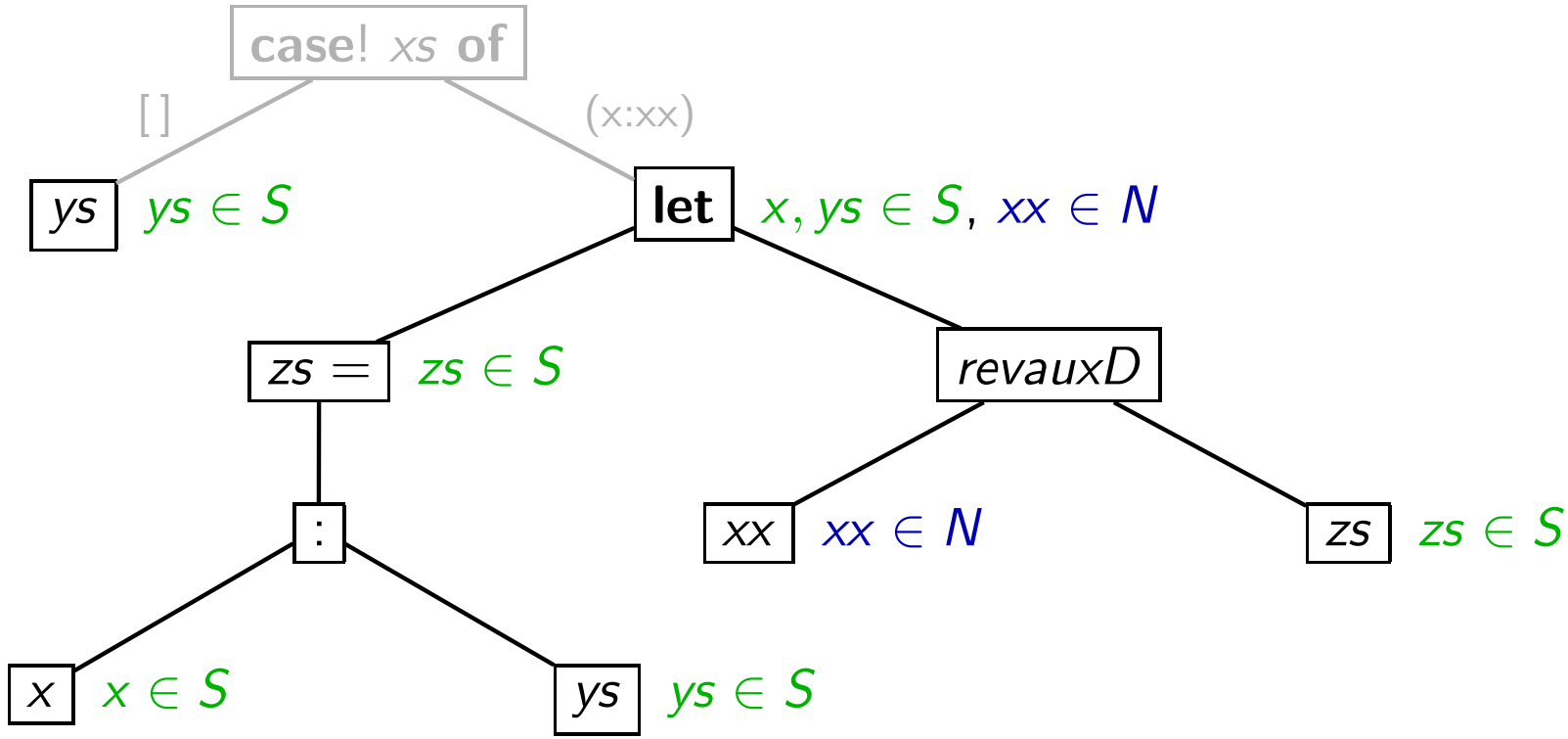
- From the information of the main expression of the let binding, we see that *zs* is not used destructively there. Hence it can be marked as safe.
- However, a top-down traversal of the function application is needed in order to check that the corresponding occurrence of *zs* may be marked as safe.

Example:*revauxD* - First iteration



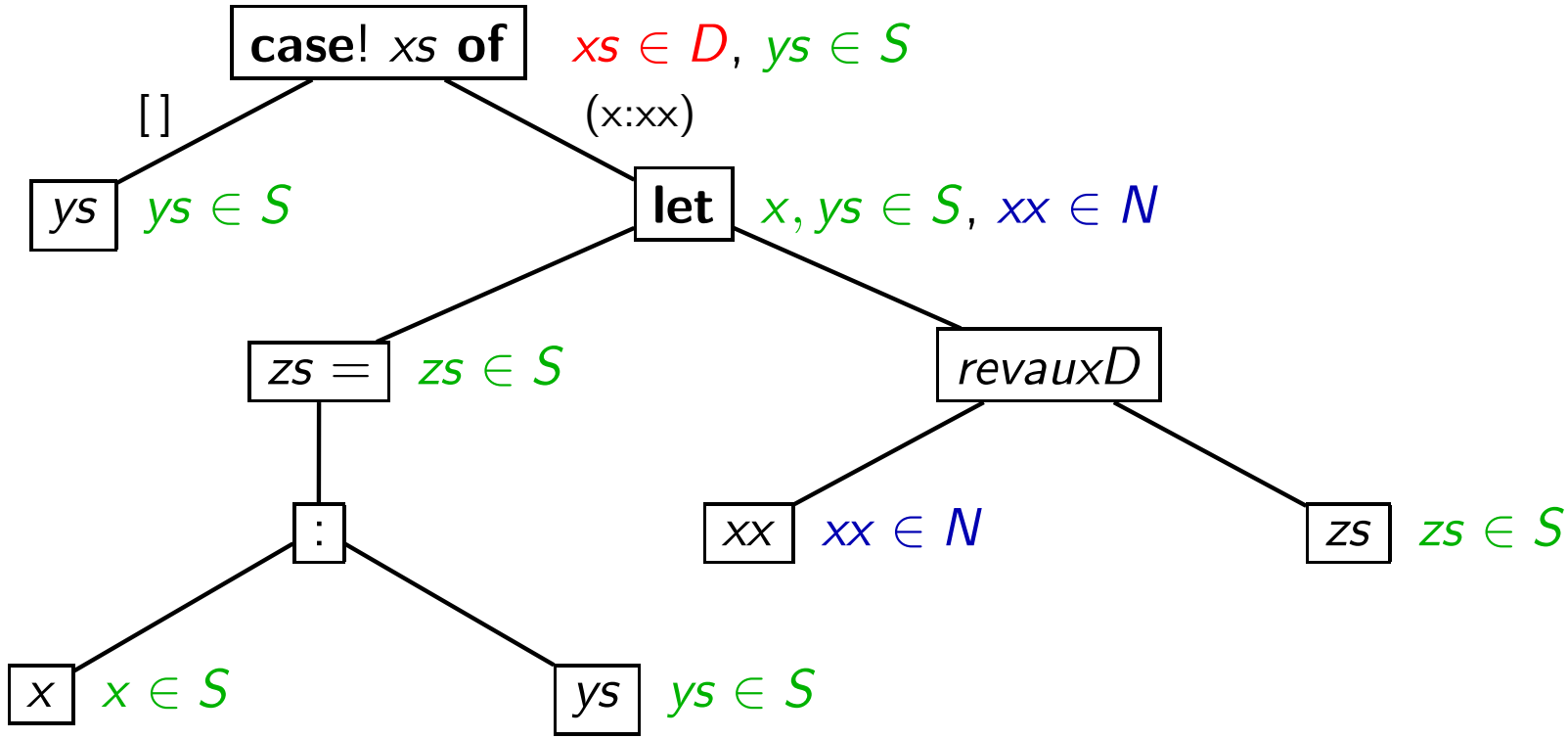
- From the information of the main expression of the let binding, we see that *zs* is not used destructively there. Hence it can be marked as safe.
- However, a top-down traversal of the function application is needed in order to check that the corresponding occurrence of *zs* may be marked as safe.

Example:*revauxD* - First iteration



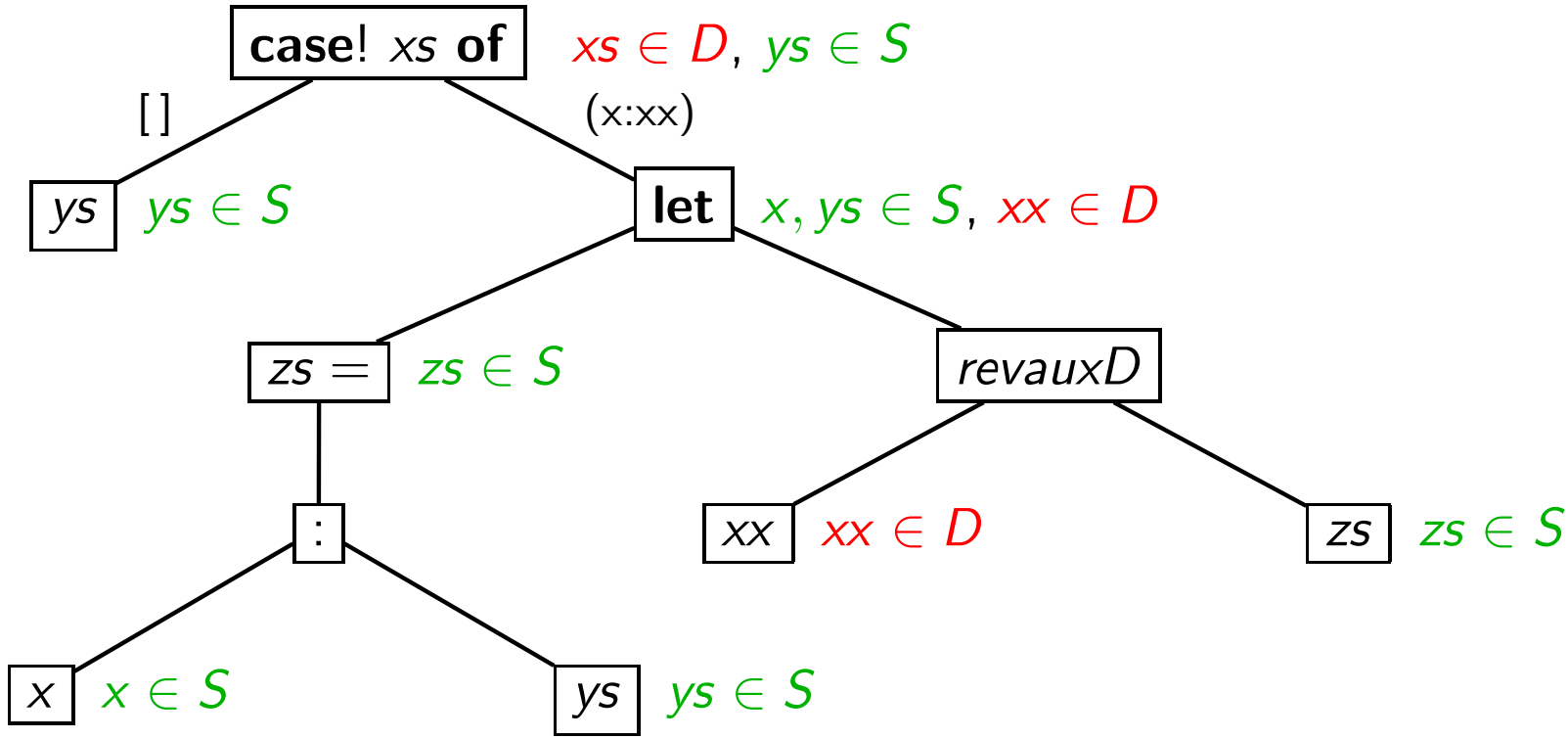
- In the **case!** alternative guarded by [ ], variable *ys* is marked as safe.
- Notice that the marks of *ys* are consistent in both alternatives and hence $\sqcup$ is well-defined.

Example:*revauxD* - First iteration



- In the **case!** expression *xs* is inferred as condemned, since we are destroying its associated data structure.
- However, a top-down traversal of the second alternative is needed to check that *xx* may be marked as condemned, since it is a recursive child of *xs*.

Example:*revauxD* - First iteration



- The following sets are inferred:

$$D = \{xs\} \quad R = \{\} \quad S = \{ys\} \quad N = \{\}$$

- The type signature of *revAuxD* is updated: The first position is now condemned and the second one is safe.

Termination Analysis

- In Safe, heap memory consumption depends on the length of recursive calls chains.
- Being able to giving bounds to those chains length becomes essential in computing space bounds.
- We have investigated how to analyse termination and complexity bounds of Safe programs by using standard term rewriting techniques
- In particular, we have investigated how to use termination proofs which combine the dependency pairs approach, together with polynomial interpretations, to obtain suitable bounds to the length of chains of recursive calls in Safe programs.

Transformation from Core-SAFE to TRS

$$trP(\overline{def_i}^n) \stackrel{\text{def}}{=} \bigcup_{i=1}^{n} trF(def_i)$$

$$trF(f\ \overline{x_i}^n = e) \stackrel{\text{def}}{=} f(x_1, \ldots, x_n) \to trR(e, fv(e), [\,])$$

$$trR(c, V, C) \stackrel{\text{def}}{=} c \Leftarrow C$$

$$trR(x, V, C) \stackrel{\text{def}}{=} x \Leftarrow C$$

$$trR(C_r\ \overline{a_i}^n, V, C) \stackrel{\text{def}}{=} C_r(a_1, \ldots, a_n) \Leftarrow C$$

$$trR(f\ \overline{a_i}^n, V, C) \stackrel{\text{def}}{=} f(a_1, \ldots, a_n) \Leftarrow C$$

$$trR(k : \textbf{case}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n, V, C) \stackrel{\text{def}}{=}$$
$$\{case_k(x, var(V)) \Leftarrow C\} \cup$$
$$\{case_k(C_i(x_{i1}, \ldots, x_{in_i}), var(V)) \to trR(e_i, fv(e_i), [\,]) \mid i \in \{1..n\}\}$$

$$trR(\textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2, V, C) \stackrel{\text{def}}{=} trR(e_2, fv(e_2), C +\!\!+ [, trL(e_1, fv(e_1)) \to x_1])$$

$$trL(e, V) \stackrel{\text{def}}{=} trR(e, V, [\,]) \text{ if } e \in \{c, x, C_r\ \overline{a_i}^n, f\ \overline{a_i}^n, \textbf{case}\}$$

$$trL(\textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2, V) \stackrel{\text{def}}{=} [trL(e_1, fv(e_1)) \to x_1,] +\!\!+ trL(e_2, fv(e_2))$$

Transformation of splitD

Then, we obtain a deterministic 3-CTRS $\mathcal{R}$. Let us call $U(\mathcal{R})$ to the TRS resulting from the standard transformation to unconditional TRSs:

```
splitD(n,xs) -> case1(n,n,xs)
case1(0,n,xs) -> Tup(Nil,xs)
case1(S(x),n,xs) -> case2(xs,n)
case2(Nil,n) -> Tup(Nil,Nil)
case2(Cons(x,xx),n) -> U1(n-1,x,xx)
U1(n',x,xx) -> U2(splitD(n',xx),x)
U2(z,x) -> U3(case3(z),z,x)
U3(xs1,z,x) -> U4(case4(z),x,xs1)
U4(xs2,x,xs1) -> U5(Cons(x,xs1),xs2)
U5(xs1',xs2) -> Tup(xs1',xs2)
case3(Tup(ys1,ys2)) -> ys1
case4(Tup(zs1,zs2)) -> zs2
```

The Dependency Pairs Approach

- Dependency pairs are rewrite rules obtained from a Term Rewriting System which show how function calls evolve in rewriting computations.

Definition

Given a TRS $\mathcal{R} = (\mathcal{C} \uplus \mathcal{D}, R)$ the set $\mathrm{DP}(\mathcal{R})$ of *dependency pairs* for $\mathcal{R}$ is given as follows: if $f(t_1, \ldots, t_m) \to r \in R$ and $r = C[g(s_1, \ldots, s_n)]$ for some defined symbol $g \in \mathcal{D}$, and context $C[\cdot]$, and $s_1, \ldots, s_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, then $f^\sharp(t_1, \ldots, t_m) \to g^\sharp(s_1, \ldots, s_n) \in \mathrm{DP}(R)$, where $f^\sharp$ and $g^\sharp$ are new fresh symbols associated to f and g respectively.

- Dependency pairs can be presented as a (dependency) graph whose cycles represent *recursive calls*.
- Termination of rewriting computations can be analysed by considering the cycles in the dependency graph.
- This corresponds to the standard view of recursion as the source of any non-terminating behaviour.

Polynomial Bounds

Proposition (polynomial bounds)

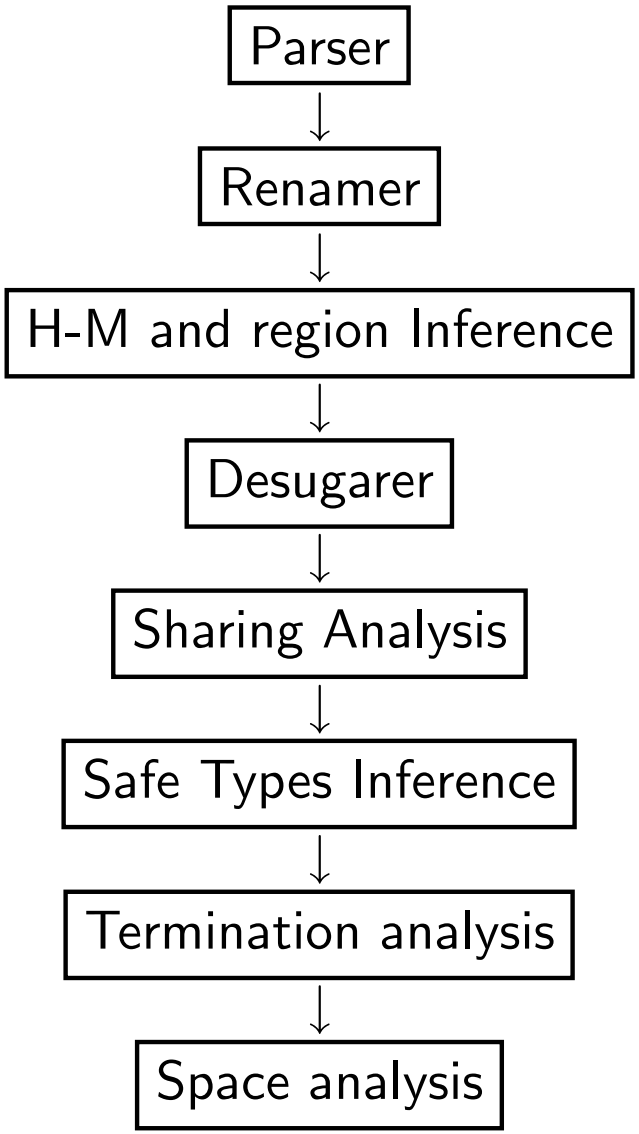
If $[\![f^\sharp]\!]$ is the polynomial interpreting the symbol $f^\sharp$ associated to a n-ary function symbol f in a Core-SAFE program and $x_1, \ldots, x_n$ are interpreted as the sizes of the input arguments to f, then the number $N(x_1, \ldots, x_n)$ of recursive calls to f with arguments $t_1, \ldots, t_n$ of sizes $x_1, \ldots, x_n$, respectively, is bounded by $[\![f^\sharp]\!](x_1, \ldots, x_n)$, i.e. $N(x_1, \ldots, x_n) \leq [\![f^\sharp]\!](x_1, \ldots, x_n)$

Safe function	**Polynomial inferred**
$length(x)$	$x + 1$
$splitD(n, x)$	$n + x + 1$
$mergeD(x, y)$	$x + y + 1$
$msortD(x)$	No proof obtained
$insert(x, t)$	$t + 1$

Memory Consumption Analysis

- Once we have obtained (or manually annotated) a polynomial for every Safe function, we foresee that inferring polynomial space bounds will be relatively easy.
- Operational semantics will be enriched with space cost annotations both for heap and stack consumption, i.e. a cost model of the core language will be developed. Cell deallocation will give rise to negative consumption.
- Every output region and the self region will constitute an independent cost centre. Cost will be charged at construction and at function applications.
- In function applications we will make use of the previously inferred costs for the function. A function may charge costs to each output region and to its self region.
- The cost finally charged for a recursive call to every region will be multiplied by the polynomial representing the bound on the number of recursive calls of the current function.
- Least upper bounds will be computed at **case** expressions.

Implementation



- A complete front-end for SAFE has been developed by using standard tools (about 5,000 Haskell lines).
- The sugared language contains **data** declarations, infix operators, **where** clauses, guards, etc. The desugarer transforms it to core language.
- The Hindley-Milner infers the usual polymorphic types for functions and also the regions needed for each function. The safe types analysis makes use of the underlying H-M types.
- The sharing analysis decorates the program with sharing information from which *shareall* and *sharerec* are extracted.
- Then, safe types are inferred and the AST is decorated with them.
- Finally termination and space analyses are done and the AST is decorated with polynomials for termination and space bounds.

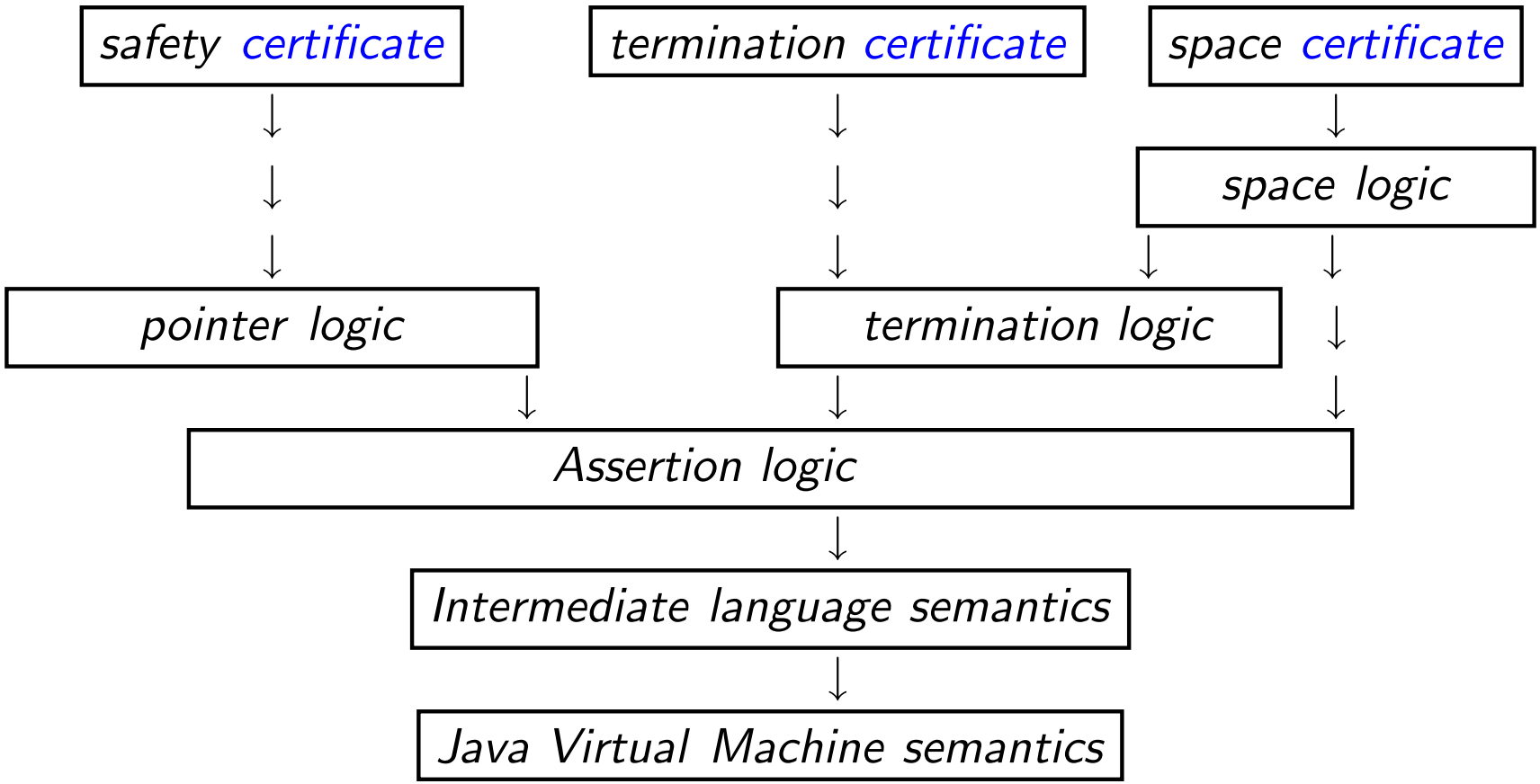
The Proof Carrying Code Paradigm

- There is a producer that generates code, and also a proof or certificate related to this code, asserting that certain properties are satisfied.
- The certificate may be generated by hand, automatically or by using a proof assistant. The paradigm allows any combination of those.
- There is a recipient who gets the code and the proof and checks that the proof is correct. The recipient only trusts in his/her proof checker.
- Once the code is proof-checked, the recipient runs it safely in his machine. This approach is superior to having runtime checks.
- For the paradigm to work, certain properties are desirable:
  1. The code should be as standard as possible. Java bytecode or C are good candidates.
  2. The certificate should be specific to this code.
  3. Certificates should be as small as possible.
  4. The proof checking should be not only automatic but also efficient.

The Proof Assistant Isabelle/HOL

- Isabelle is developed at University of Cambridge (Larry Paulson) and Technical University of Munich (Tobias Nipkow). See
  `http://isabelle.in.tum.de`
- Isabelle is a generic proof assistant. It allows mathematical formulas to be expressed in a formal language and provides tools for proving those formulas in a logical calculus.
- Provides support for nominal datatypes (binding structures) due to the HOL-Nominal logic.
- It has an inductive package for inductive predicates and inductive sets.
- Allows general recursive function definitions.
- Has a code generator for a subset of HOL, targeting SML, Haskell, and OCaml.
- Interactive successful sessions are recorded in theory files, which can be seen as real proofs, since they can be replayed when desired.

Our approach



- Certificates are high-level objects inferred by the compiler: types, polynomials
- The PCC infrastructure consists of layers of Isabelle/HOL definitions and theorems pushing up the level at which the proof is done.

A high-level assertion in pointer logic

Definition

We say that the Safe expression e satisfies the derived assertion $A = [\![L, \Gamma]\!]$ denoted $e : [\![L, \Gamma]\!]$ if:

1. $L = fv(e)$ and $E \vdash h, e \Downarrow h', v$
2. For every pair of variables $x \in dom(E),\ z \in L$,

   $$\Gamma[z] = d \wedge recReach(E, z, h) \cap closure(E, x, h) \neq \emptyset \rightarrow x \in dom(\Gamma) \wedge \Gamma[x] \neq s$$

3. Whenever the initial configuration (Γ, E, h, L) is *good*, the final configuration (v, h') is also good. (Γ, E, h, L) is good whenever:
   1. $S \cap R = \emptyset$ where $S \stackrel{\text{def}}{=} \bigcup_{x \in L, \Gamma[x]=s} \{closure(E, x, h)\}$ and
      $R \stackrel{\text{def}}{=} \bigcup_{x \in L, \Gamma[x]=d} \{p \in live(E, L, h) \mid p \rightarrow_h^* recReach(E, x, h)\}$.
   2. $closed(E, L, h)$

   (v, h') is good if $closed(v, h')$.

Proof Rules in Pointer Logic

$$\frac{e_1 : [\![L_1, \Gamma_1]\!] \quad e_2 : [\![L_2, \Gamma_2 + [x_1 : s]]\!] \quad def(\Gamma_1 \rhd^{L_2} \Gamma_2)}{\textbf{let } x_1 = e_1 \textbf{ in } e_2 : [\![L_1 \cup (L_2 - \{x_1\}), \Gamma_1 \rhd^{L_2} \Gamma_2]\!]} \; LET1$$

$$\frac{e_1 : [\![L_1, \Gamma_1]\!] \quad e_2 : [\![L_2, \Gamma_2 + [x_1 : d]]\!] \quad def(\Gamma_1 \rhd^{L_2} \Gamma_2)}{\textbf{let } x_1 = e_1 \textbf{ in } e_2 : [\![L_1 \cup (L_2 - \{x_1\}), \Gamma_1 \rhd^{L_2} \Gamma_2]\!]} \; LET2$$

$$\frac{\forall i \; . \; e_i : [\![L_i, \Gamma_i]\!] \wedge x \in L_i \quad \Gamma = \bigotimes_i (\Gamma_i / \{\overline{x_{ij}}\}) \quad L = \bigcup_i (L_i - \{\overline{x_{ij}}\})}{\textbf{case } x \textbf{ of } \overline{C_i \overline{x_{ij}} \rightarrow e_i} : [\![L, \Gamma]\!]} \; CASE$$

Conclusion

- PCC is an emerging technology that can dramatically increase our confidence in programs, both in safety critical applications and in frameworks where untrusted code must be accepted.
- The combination of powerful static analyses at source level with automatic generation of certificates simplifies life at the producer's side.
- Writing and analysing programs in a declarative language simplifies both program production and compiler construction.
- Producing proofs at bytecode level would imply a large amount of work. Moreover, certificates and checking time will increase dramatically.
- Developing layers of specialised logics is also a large amount of work but has the advantage that complex theorems are proved only once.