Static Analysis and Certification of Safety Properties of Memory Usage

Ricardo Peña Marí
Full Professor in Programming Languages and Computing Systems

Departament *Sistemas Informáticos y Computación*
Universidad Complutense de Madrid

PROMETIDOS-CM SUMMER SCHOOL, September 2011

Outline

Outline

The Static Analysis and Certification group

- Funded by the projects SELF (TIN2004-07943-C04-04), STAMP (TIN2008-06622-C03-01), and PROMETIDOS, (S2009/TIC-1465) we have developed Safe, a functional language aimed at applications executed in small devices having limited memory.
- The language has explict memory deallocation primitives. By using static analyses we ensure absence of dangling pointers and bounded memory consumption.
- The compiler generates either JVM bytecode or C, so that programs can be executed in virtually all platforms.
- Our compiler also generates certificates about the above properties. The certificates are Isabelle/HOL proof scripts automatically checked by this tool.
- Two PhD thesis are being presented soon this year, one on the analysis aspect, and a second one about certification.
- In the past, together with the Marburg University (Germany), we developed Eden, a parallel version of the functional language Haskell, aimed at exploiting multi-core computers, and low latency computer networks.

International Visibility

- The group publishes in international conferences devoted to declarative programming (IFL, TFP, PPDP, WFLP), program analysis and transformation (LOPSTR), formal methods (FM, FMICS), and theorem proving (TPHOL).
- It participates in the PC's of some of them (IFL, TFP, LOPSTR, PADL).
- It has international cooperation with other research groups on Resource Analysis (RU Nijmegen, TU and LMU Munich, St. Andrews, Heriot-Watt, UP Madrid, . . .).
- Recently we have set-up an International Workshop on this subject (FOPARA'09) with publication in LNCS (Springer). The second edition (FOPARA'11) has been held in Madrid, chaired by us.
- Please visit the following web pages:

http://dalila.sip.ucm.es/safe	Project's home page
http://dalila.sip.ucm.es/~safe	Our on-line compiler
http://dalila.sip.ucm.es/safe/theories	Certified translation theories
http://dalila.sip.ucm.es/safe/certifsvm2jvm	Certified translation theories
http://dalila.sip.ucm.es/safe/certifdangling	Certificate generation theories
http://dalila.sip.ucm.es/safe/bounds	Certificate generation theories

Outline

Static analysis techniques (1)

- Used by compilers to extract from the (static) program text, useful (dynamic) properties that will be satisfied at runtime. Started around 1970.
- The most obvious example is type checking and type inference. The dynamic property here is ensuring that a variable will only contain at runtime values belonging to its (static) type.
- Most industrial compilers include static analyses. The properties extracted are used to optimize the generated target code. Examples:
  1. Constant propagation: Some expressions can be evaluated at compile time, so no code is generated for them.
  2. Live variables: It minimizes the number of registers used.
  3. Dead code: It eliminates code that will never be executed.
  4. Available expressions: Tries to evaluate repeated expressions only once.
  5. Indices out of range: Aimed at avoiding out of range runtime checkings.
  6. Strictness: In functional languages, it replaces some call-by-need parameter passing by the more efficient call-by-value mode.
  7. Mode: In logic languages, it detects the mode (instantiated/uninstantiated) of the logical variables used by predicates. Knowing the mode allows generating a more efficent code.

Static analysis techniques (and 2)

Techniques Two broad families:

- Abstract interpretation: A sort of program symbolic execution on a (frequently) finite domain of values (abstract domains).
- Type-based: Special annotated type systems and type inference algorithms are used, where annotations express properties.

Safety and precision

- The studied properties are usually undecidable. The result may sometimes be 'don't know'.
- Otherwise, the result must be safe (i.e. sound), although lack of precision is allowed.

Modern trends In the last few years more and more properties related to program correctness are being studied:

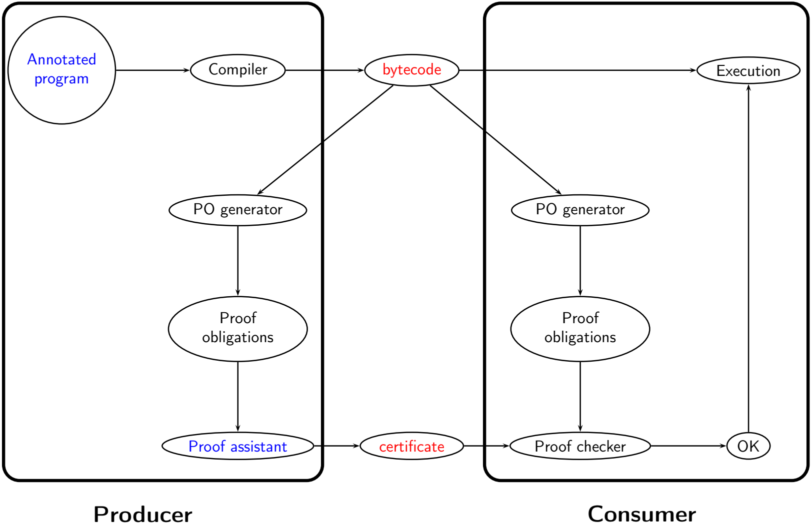
- Termination
- Data sizes
- Upper bounds on time consumption
- Upper bounds on memory consumption

Outline

Proof Carrying Code

- Proof Carrying Code, (PCC) is a new research trend in which programs are produced endowed with certificates guaranteeing that certain properties are satisfied.
- A certificate is a mathematical proof of a collection of theorems about the program code sentences.
- The certificates are automatically checked by the code consumer by using appropriate tools.
- The PCC paradigm is quickly developing in some critical areas: mobile code, third party code, Internet downloaded code, etc, in which the code comes from untrusted sources and it may damage the recipient machine.
- Differently to other security policies, in PCC the consumer only trusts in his/her proof checker.
- There are at present several active projects in this area, either locally or EU funded.

The PCC paradigm (1)



The PCC paradigm (and 2)

- The producer generates the annotated code and a certificate proving that the code satisfies certain properties.
- The certificate may be either manually generated, or interactively by using a proof assistant, or automatically. Any combination is allowed.
- The consumer checks that the certificate is related to the code and that the proofs are correct. The PCC paradigm insists in that this phase must be automatic, for instance by using a proof checker.
- Once the code is validated, it is allowed to be executed without any further checkings (compare with the alternative of having runtime checkings).
- Some properties are desired:
    1. The code should be as standard as possible (e.g. Java bytecode or C).
    2. The certificate should be as small as possible.
    3. The proof checking process should be efficient.

The Proof-Assistant Isabelle/HOL

- Isabelle has been developed in the 1990's by the Cambridge University (Larry Paulson) and Technical University of Munich (Tobias Nipkow) (visit `http://isabelle.in.tum.de`).
- Isabelle is a generic proof-assistant, which can be parameterized by different logics. The one most used is HOL (*Higher Order Logic*).
- It provides a formal language in which definitions and properties can be easily expressed, and also supports a variety of proof techniques the user can choose among in order to prove theorems.
- At every proof-step, it shows the subgoals that remain to be proved. When every subgoal has been discharged, the theorem proof is complete.
- It can store the already proved theorems in a theorem database, so that they could be reused in further proofs.
- The definitions and proof-scripts are always available as text files (called theories) and can be replayed at any time. This feature allows using Isabelle as a proof-checker of proofs conducted in a different Isabelle installation.

Outline

The *Safe* language

- Similar to Haskell, but first-order and eager
- No runtime garbage collector. Inferred *regions* instead
- Explicit memory disposal by using destructive pattern matching
- A special type analysis ensures absence of dangling pointers at runtime
- An abstract interpretation based analysis computes upper bounds on memory comsumption, as symbolic functions on input argument sizes
- Intermediate functional language *Core-Safe*
  1. Regions are explicit in *Core-Safe* as arguments to function and constructor applications
  2. Dangling pointers analysis, memory bound analysis, and certification of both properties are performed at this level
- Certificates are mainly Isabelle/HOL proof-scripts
- Also, the computer algebra tool QEPCAD is used for memory bound certification

Conventional version of mergesort

$$
\begin{aligned}
&\mathit{split}\ 0\ \mathit{xs} && = ([\,], \mathit{xs})\\
&\mathit{split}\ n\ [\,] && = ([\,], [\,])\\
&\mathit{split}\ n\ (x : \mathit{xs}) && = (x : \mathit{xs}_1, \mathit{xs}_2)\\
& && \quad \textbf{where}\ (\mathit{xs}_1, \mathit{xs}_2) = \mathit{split}\ (n-1)\ \mathit{xs}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{merge}\ [\,]\ \mathit{ys} && = \mathit{ys}\\
&\mathit{merge}\ \mathit{xs}\ [\,] && = \mathit{xs}\\
&\mathit{merge}\ (x : \mathit{xs})\ (y : \mathit{ys})\\
&\qquad \mid x \leq y && = x : \mathit{merge}\ \mathit{xs}\ (y : \mathit{ys})\\
&\qquad \mid \mathit{otherwise} && = y : \mathit{merge}\ (x : \mathit{xs})\ \mathit{ys}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{msort}\ \mathit{xs}\\
&\quad \mid n \leq 1 && = \mathit{xs}\\
&\quad \mid \mathit{otherwise} && = \mathit{merge}\ (\mathit{msort}\ \mathit{xs}_1)\ (\mathit{msort}\ \mathit{xs}_2)\\
&\quad \textbf{where}\ (\mathit{xs}_1, \mathit{xs}_2) && = \mathit{split}\ (n\ \text{`}\mathit{div}\text{`}\ 2)\ \mathit{xs}\\
&\qquad\qquad n && = \mathit{length}\ \mathit{xs}
\end{aligned}
$$

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

Safe destructive version of mergesort

$$
\begin{array}{l}
\mathit{splitD}\ 0\ \mathit{xs}! \qquad\qquad = ([\,], \mathit{xs}!) \\
\mathit{splitD}\ n\ [\,]! \qquad\qquad = ([\,], [\,]) \\
\mathit{splitD}\ n\ (x : \mathit{xs})! = (x : \mathit{xs}_1, \mathit{xs}_2) \\
\qquad\qquad\qquad\qquad\quad \textbf{where}\ (\mathit{xs}_1, \mathit{xs}_2) = \mathit{splitD}\ (n-1)\ \mathit{xs} \\
\mathit{mergeD}\ [\,]!\ \mathit{ys}! \qquad\qquad\quad = \mathit{ys}! \\
\mathit{mergeD}\ \mathit{xs}!\ [\,]! \qquad\qquad\quad = \mathit{xs}! \\
\mathit{mergeD}\ (x : \mathit{xs})!\ (y : \mathit{ys})! \\
\qquad\qquad\quad \mid x \leq y \qquad = x : \mathit{mergeD}\ \mathit{xs}\ (y : \mathit{ys}!) \\
\qquad\qquad\quad \mid \mathit{otherwise} = y : \mathit{mergeD}\ (x : \mathit{xs}!)\ \mathit{ys} \\
\quad \mid n \leq 1 \qquad = \mathit{xs}! \\
\quad \mid \mathit{otherwise} = \mathit{mergeD}\ (\mathit{msortD}\ \mathit{xs}_1\ )\ (\mathit{msortD}\ \mathit{xs}_2\ ) \\
\quad\ \textbf{where}\ \ (\mathit{xs}_1, \mathit{xs}_2) = \mathit{splitD}\ (n\ `\mathit{div}`\ 2)\ \ \mathit{xs} \\
\qquad\qquad\quad n \qquad\qquad\ = \mathit{length}\ \mathit{xs}
\end{array}
$$

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

Memory management

Implicit — *Java, Haskell, Python ...*

Semiexplicit

- **Region based memory management** — Implicit
- Destructive pattern matching — Explicit

→ No garbage collection
→ Reasoning about the heap needed by a program
→ Safe destruction

Explicit — *C, C++, ...*

Region inference

Written by the programmer:

$$
\begin{array}{l}
\mathit{partition}\ y\ [\,] = ([\,],\ [\,]) \\
\mathit{partition}\ y\ (x : xs) \\
\qquad\qquad \mid\ x \leq y = (x : ls,\ gs\ ) \\
\qquad\qquad \mid\ x > y = (ls\ ,\ x : gs) \\
\qquad\qquad\qquad \mathbf{where}\ (ls, gs) = \mathit{partition}\ y\ xs
\end{array}
$$

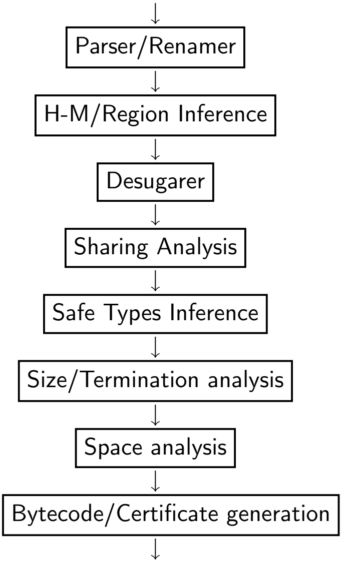
After region inference:

$$
\begin{array}{rcl}
\mathit{partition} & :: & \mathit{Int} \rightarrow [\mathit{Int}]@\rho_1 \rightarrow \rho_2 \rightarrow \rho_3 \rightarrow \rho_4 \rightarrow ([\mathit{Int}]@\rho_2, [\mathit{Int}]@\rho_3)@\rho_4 \\
\text{internal call} & :: & \mathit{Int} \rightarrow [\mathit{Int}]@\rho_1 \rightarrow \rho_2 \rightarrow \rho_3 \rightarrow \rho_9 \rightarrow ([\mathit{Int}]@\rho_2, [\mathit{Int}]@\rho_3)@\rho_9
\end{array}
$$

$$
\begin{array}{l}
\mathit{partition}\ y\ [\,]\ @\ r_2\ r_3\ r_4 = ([\,]\ @\ r_2,\ [\,]\ @\ r_3)\ @\ r_4 \\
\mathit{partition}\ y\ (x : xs)\ @\ r_2\ r_3\ r_4 \\
\qquad\qquad \mid\ x \leq y = (x : ls\ @\ r_2,\ gs\ )\ @\ r_4 \\
\qquad\qquad \mid\ x > y = (ls\ ,\ x : gs\ @\ r_3)\ @\ r_4 \\
\qquad\qquad\qquad \mathbf{where}\ (ls, gs) = \mathit{partition}\ y\ xs)\ @\ r_2\ r_3\ \mathit{self}
\end{array}
$$

Outline

The *Safe* compiler



- A complete *Safe* compiler has been developed in Haskell.
- The Hindley-Milner infers the usual polymorphic types for functions and also the regions needed for data and functions.
- The full language has **data** declarations, infix operators, **where** clauses, guards, etc. The desugarer transforms it to core language.
- The sharing analysis decorates the program with sharing information needed by safe types.
- Then, safe types (no-dangling pointers) are inferred and the AST is decorated with them.
- Then, size, termination and space analyses are done and the AST decorated with space bounds.
- Finally, certificates about these properties and the target code are generated.

Core SAFE normalised syntax

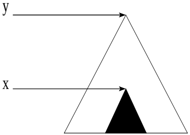
$$
\begin{array}{llll}
prog & \rightarrow & dec_1; \ldots; dec_n; e & \{\text{program}\} \\
dec & \rightarrow & f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e & \{\text{single-recursive, polymorphic function definition}\} \\
 & & & \quad \{\text{atom}\} \\
a & \rightarrow & c & \{\text{atomic constant}\}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

Region inference

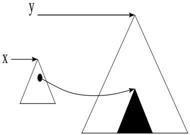
- Region inference takes place as a by-product of the Hindley-Milner type inference phase of the compiler.
- Program region variables r are mandatory at constructions $C\ \overline{a_i}^n$ @ r and at function application $f\ \overline{a_i}^n$ @ $\overline{r_j}^m$ for known functions f. The compiler assigns 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 structures 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.
- It supports also polymorphic recursion on region type variables.

Sharing Relations and Analysis

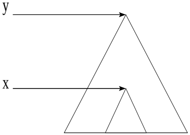
$x \mathrel{\lhd\!\!\sim} y$ (*SubR*)



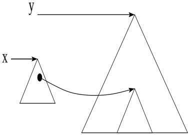
$x \mathrel{\triangle\!\!\!\sim} y$ (*ShR*)



$x \lhd y$ (*Sub*)



$x \triangle y$ (*Sh*)



The analysis is based on abstract interpretation and performs a top-down traversal of the program text, while collecting the sharing relations between variables. It decorates the AST with this information.

Type expressions (1)

τ	$\rightarrow$	t	{visible}	r	$\rightarrow$	$T\ \overline{s}\#@\overline{\rho}^m$		
		$\mid r$	{in-danger}					
		$\mid \sigma$	{polymorphic function}	b	$\rightarrow$	a	{variable}
		$\mid \rho$	{region}			$\mid B$	{basic}
t	$\rightarrow$	s	{safe}	tf	$\rightarrow$	$\overline{t_i}^n \rightarrow \overline{\rho}^l \rightarrow T\ \overline{s}@\overline{\rho}^m$	{function}
		$\mid d$	{condemned}			$\mid \overline{t_i}^n \rightarrow b$	
						$\mid \overline{s_i}^n \rightarrow \rho \rightarrow T\ \overline{s}@\overline{\rho}^m$	{constructor}
s	$\rightarrow$	$T\ \overline{s}@\overline{\rho}^m$					
		$\mid b$		σ	$\rightarrow$	$\forall a.\sigma$	
						$\mid \forall \rho.\sigma$	
d	$\rightarrow$	$T\ \overline{t}!@\overline{\rho}^m$				$\mid tf$	

Non functional algebraic types may be:

- **Condemned (!):** Directy involved in the destructive action of a **case!** expression.
- **In-danger (#):** Pointers to recursive childs of a condemned closure.
- **Safe (s):** May be read, copied or used to build other data structures.

Type expressions (2)

Constructors (polymorphic, read-only, one-region arguments)

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

Functions (polymorphic, read-only and/or condemned arguments, zero or more region arguments)

$concatD :: \forall \rho_1, \rho_2 \,.\, [a]!@\rho_1 \rightarrow [a]@\rho_2 \rightarrow \rho_2 \rightarrow [a]@\rho_2$
$mkTreeD :: \forall \rho_1, \rho_2 \,.\, [Int]!@\rho_1 \rightarrow \rho_2 \rightarrow BSTree\ Int@\rho_2$
$insertD :: \forall \rho \,.\, Int \rightarrow BSTree\ Int!@\rho \rightarrow \rho \rightarrow BSTree\ Int@\rho$
$split :: \forall a, \rho_1, \rho_2, \rho_3 \,.\, Int \rightarrow [a]@\rho_2 \rightarrow \rho_1 \rightarrow \rho_2 \rightarrow \rho_3 \rightarrow ([a]@\rho_1, [a]@\rho_2)@\rho_3$
$length :: \forall a \,.\, [a] \rightarrow Int$

Typing rule for **let**

$$\frac{\Gamma_1 \vdash e_1 : s_1 \qquad \Gamma_2 + [x_1 : \tau_1] \vdash e_2 : s \qquad \mathit{utype?}(\tau_1, s_1)}{\Gamma_1 \rhd^{\mathit{fv}(e_2)} \Gamma_2 \vdash \mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2 : s}\ [\mathrm{LET}]$$

$$\begin{array}{rl} \mathit{def}(\Gamma_1 \rhd^L \Gamma_2) \equiv & (\forall x \in \mathit{dom}(\Gamma_1) \cap \mathit{dom}(\Gamma_2).\ \mathit{utype?}(\Gamma_1(x), \Gamma_2(x))) \wedge \\ & (\forall x \in \mathit{dom}(\Gamma_1).\ \mathit{unsafe?}(\Gamma_1(x)) \rightarrow x \notin L) \end{array}$$

$$\mathbf{let}\ x = \underbrace{\mathbf{case!}\ y\ \mathbf{of} \ldots}_{e_1} \mathbf{in} \underbrace{\ldots y \ldots}_{e_2} \qquad \text{(forbidden)}$$

$$\forall x \in \mathit{dom}(\Gamma_1) \cup \mathit{dom}(\Gamma_2).\ (\Gamma_1 \rhd^L \Gamma_2)(x) = \begin{cases} \Gamma_2(x) & \text{if } x \notin \mathit{dom}(\Gamma_1) \vee \\ & \quad (x \in \mathit{dom}(\Gamma_1) \cap \mathit{dom}(\Gamma_2) \wedge \mathit{safe}(\Gamma_1(x))) \\ \Gamma_1(x) & \text{otherwise} \end{cases}$$

$$\mathbf{let}\ x = \underbrace{\mathbf{case}\ y\ \mathbf{of} \ldots}_{e_1} \mathbf{in} \underbrace{\ldots \mathbf{case!}\ y\ \mathbf{of} \ldots\ldots}_{e_2} \qquad \text{(allowed)}$$

The inference algorithm is based on a bottom-up propagation of marks along the AST, and on a fixpoint algorithm in the case of recursive functions.

Resource-aware semantics

- We add resource computations to the big-step operational semantics.

$$(\delta\,,m\,,s)$$

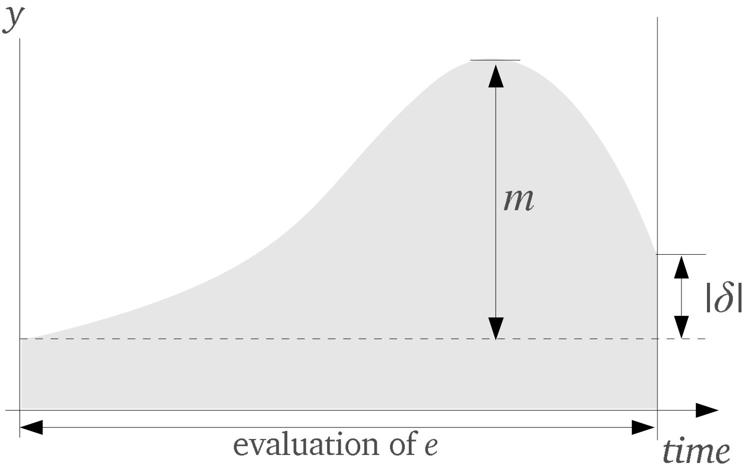
signed difference between the cells in the initial and final heaps

minimum number of cells in heap/stack

Rule for let bindings

$$\frac{\begin{array}{l} E \vdash h, k, 0, e_1 \Downarrow h', k, v_1, (\delta_1, m_1, s_1) \\ E \uplus [x_1 \mapsto v_1] \vdash h', k, td+1, e_2 \Downarrow h'', k, v, (\delta_2, m_2, s_2) \end{array}}{E \vdash h, k, td, \textbf{let } x_1 = e_1 \textbf{ in } e_2 \Downarrow h'', k, v, (\delta_1 + \delta_2, max\{m_1, |\delta_1| + m_2\}, max\{2 + s_1, 1 + s_2\})} \; [Let_1]$$

Relationship between δ and m



Function signatures

- Let f be a *Core-Safe* function with $n + m$ parameters:
  $f\ x_1 \cdots x_n\ @\ r_1\ \cdots\ r_m = e_f$
- Space costs can be expressed as n-ary functions of the sizes of the input arguments. Let $\mathbb{F}$ be the corresponding domain:

$$\mathbb{F} = \{\eta : (\mathbb{R}^+ \cup \{+\infty\})^n \to \mathbb{R}^+ \cup \{+\infty\} \mid \eta \text{ is monotonic}\}$$

- A subexpression of e_f may charge costs to:
  - The output regions $r_1 \ldots r_m$. R^f_{out} contains their region types.
  - The working region of type ρ_{self}.
  - The visible charges of f do not include ρ_{self}.
- Let $\mathbb{D}$ be the domain of functions $R_f \to \mathbb{F}$ that describe the space costs charged by f to every visible region, where $R_f = R^f_{out} \cup \{\rho_{self}\}$ for expressions, and $R_f = R^f_{out}$ for f.

Definition

A **function signature** is a triple $(\Delta_f, \mu_f, \sigma_f)$, where $\Delta_f \in \mathbb{D}$ and $\mu, \sigma \in \mathbb{F}$.

- **Aim**: Δ_f approximates δ, μ approximates m and σ approximates s.

Abstract interpretation rules (I)

- Judgements of the form $[\![e]\!]\ \Sigma\ \Gamma\ td = (\Delta, \mu, \sigma)$.

Rules for atoms and basic operations

$$[\![a]\!]\ \Sigma\ \Gamma\ td = ([\,]_f, 0, 1) \quad [Atom] \qquad [\![a_1 \oplus a_2]\!]\ \Sigma\ \Gamma\ td = ([\,]_f, 0, 2) \quad [Primop]$$

$$\text{where } [\,]_f \equiv [\rho \mapsto \lambda\overline{x_i}^n.0 \mid \rho \in R_f] \text{ and } k \equiv \lambda\overline{x_i}^n.k$$

Rules for let bindings

$$\frac{[\![e_1]\!]\ \Sigma\ \Gamma\ 0 = (\Delta_1, \mu_1, \sigma_1) \quad [\![e_2]\!]\ \Sigma\ \Gamma\ (td+1) = (\Delta_2, \mu_2, \sigma_2)}{[\![\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2]\!]\ \Sigma\ \Gamma\ td = (\Delta_1 + \Delta_2, \sqcup\{\mu_1, |\Delta_1| + \mu_2\}, \sqcup\{2+\sigma_1, 1+\sigma_2\})}\ [Let_1]$$

$$\frac{\Gamma\ r = \rho \quad [\![e_2]\!]\ \Sigma\ \Gamma\ (td+1) = (\Delta, \mu, \sigma)}{[\![\mathbf{let}\ x_1 = C\ \overline{a_i}^n\ @\ r\ \mathbf{in}\ e_2]\!]\ \Sigma\ \Gamma\ td = (\Delta + [\rho \mapsto 1], \mu + 1, \sigma + 1)}\ [Let_2]$$

$$\text{where } |\Delta| = \sum_{\rho \in dom(\Delta)} \Delta\ \rho$$

Case study: Mergesort

Function	Heap charges $\triangle$	Heap needs μ	Stack needs σ
$length(x)$	$[\,]$	0	$5x-4$
$split(n,x)$	$\begin{bmatrix} \rho_1 \mapsto 1 \\ \rho_2 \mapsto min(n,x-1)+1 \\ \rho_3 \mapsto min(n,x-1)+1 \end{bmatrix}$	$2min(n,x-1)+3$	$9min(n,x-1)+4$
$merge(x,y)$	$\begin{bmatrix} \rho_1 \mapsto max(1,2x+2y-5) \end{bmatrix}$	$max(1,2x+2y-5)$	$11(x+y-4)+20$
$msort^1(x)$	$\begin{bmatrix} \rho_1 \mapsto \frac{x^2}{2}-\frac{1}{2} \\ \rho_2 \mapsto 2x^2-3x+3 \end{bmatrix}$	$0{,}31x^2+0{,}25x\log(x+1)+14{,}3x$ $+0{,}75\log(x+1)+10{,}3$	$max(80,13x-10)$
$msort^2(x)$	$\begin{bmatrix} \rho_1 \mapsto \frac{x^2}{4}+x-\frac{1}{4} \\ \rho_2 \mapsto x^2+x+1 \end{bmatrix}$	$0{,}31x^2+8{,}38x+13{,}31$	$max(80,11x-25)$
$msort^3(x)$	$\begin{bmatrix} \rho_1 \mapsto \frac{x^2}{8}+\frac{7x}{4}+\frac{9}{8} \\ \rho_2 \mapsto \frac{x^2}{2}+4x+\frac{1}{2} \end{bmatrix}$	$0{,}31x^2+8{,}38x+13{,}31$	$max(80,11x-25)$

$\square \equiv$ fixpoint reached

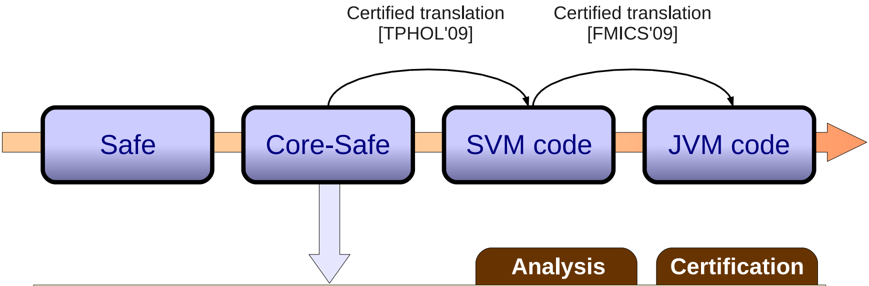
Case studies

Function	Heap needs μ	Stack needs σ
$partition(p, x)$	$\boxed{3x - 1}$	$\boxed{9x - 5}$
$append(x, y)$	$\boxed{x - 1}$	$\boxed{max(8, 7x - 6)}$
$quicksort(x)$	$3x^2 - 20x + 76$	$max(40, 20x - 27)$
$insertD(e, x)$	$\boxed{1}$	$\boxed{9x - 1}$
$insertTD(x, t)$	$\boxed{2}$	$\boxed{\frac{11}{2}t + \frac{7}{2}}$
$fib(n)$	$2^n + 2^{n-3} + 2^{n-4} - 3$	$max(10, 7n - 11)$
$sum(n)$	$\boxed{0}$	$\boxed{3n + 6}$
$sumT(a, n)$	$\boxed{0}$	$\boxed{5}$

$\boxed{} \equiv$ fixpoint reached

Outline

Overview



	Analysis	Certification
Absence of dangling pointers	✔	✔
Memory consumption	✔	✔

Absence of dangling pointers: correctness of cell destruction

Definition: Given the following properties:

$$
\begin{array}{lcl}
P1 & \equiv & E \vdash h, k, e \Downarrow h', k, v \\
P2 & \equiv & dom(\Gamma) \subseteq dom(E) \\
P3 & \equiv & L \subseteq dom(\Gamma) \\
P4 & \equiv & fv(e) \subseteq L \\
P5 & \equiv & \forall x \in dom(E).\ \forall z \in L\ . \\
 & & \Gamma[z] = d\ \wedge\ recReach(E, z, h) \cap closure(E, x, h) \neq \emptyset \rightarrow x \in dom(\Gamma) \wedge \Gamma[x] \neq s \\
P6 & \equiv & \forall x \in dom(E)\ .\ closure\ (E, x, h) \not\equiv closure\ (E, x, h') \rightarrow x \in dom(\Gamma) \wedge \Gamma[x] \neq s \\
P7 & \equiv & S_{L,\Gamma,E,h} \cap R_{L,\Gamma,E,h} = \emptyset \\
P8 & \equiv & closed(E, L, h) \\
P9 & \equiv & closed(v, h')
\end{array}
$$

we say that the expression e satisfies the static assertion $[\![L, \Gamma]\!]$, denoted $e : [\![L, \Gamma]\!]$, if $\forall E\ h\ k\ h'\ v\ .\ (P1 \wedge P2 \rightarrow P3 \wedge P4 \wedge P5 \wedge P6 \wedge (P7 \wedge P8 \rightarrow P9))$.

Absence of dangling pointers: correctness of region deallocation

Definition: An expression e satisfies the pair (θ, t), denoted $e : [\![\theta, t]\!]$ if:

$$
\begin{array}{llll}
\forall E\ h\ k\ h'\ v\ \eta & . & E \vdash h, k, e \Downarrow h', k, v & \text{-- } P1 \\
 & \wedge & dom(E) \subseteq dom(\theta) & \text{-- } P2 \\
 & \wedge & admissible\ (\eta, k) & \text{-- } P3 \\
 & \wedge & consistent\ (\theta, \eta, E, h) & \text{-- } P4 \\
 & \rightarrow & consistent\ (t, \eta, v, h') & \text{-- } P5
\end{array}
$$

This and the premise $\rho^{g}_{self} \notin regions(t_g)$ in the application rule, guarantee that when the *self* region is deallocated, there are no cells there pointed to by the result. So, region deallocation does not create dangling pointers.

Proof rules for explicit deallocation

$$\frac{e_1 \neq C\ \overline{a_i}^n \quad e_1, \Sigma_M \vdash (L_1, \Gamma_1) \quad x_1 \notin L_1 \quad e_2, \Sigma_M \vdash (L_2, \Gamma_2' + [x_1 : s]) \quad def(\Gamma_1 \triangleright^{L_2} \Gamma_2')}{\textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2, \Sigma_M \vdash (L_1 \cup (L_2 - \{x_1\}), \Gamma_1 \triangleright^{L_2} \Gamma_2')}\ LET1$$

$$\frac{f\ \overline{x_i}^n @\ \overline{r_j}^m = e_f \quad L_f = \{\overline{x_i}^n\} \quad \Gamma_f = [\overline{x_i \mapsto m_i}^n] \quad e_f, \Sigma_M \uplus [f \mapsto \overline{m_i}^n] \vdash (L_f, \Gamma_f)}{e_f, \Sigma_M \vdash (L_f, \Gamma_f)}\ REC$$

For each expression e, the compiler generates a pair (L, Γ). According to e's syntax, the certificate chooses the appropriate proof rule, checks that its premises are satisfied, and applies it in order to get the conclusion $e, \Sigma_M \vdash (L, \Gamma)$.

Theorem (soundness)

If $e, \Sigma_M \vdash (L, \Gamma)$ *then* $e, \Sigma_M : [\![L, \Gamma]\!]$.

Proof rules for region deallocation

- We provide a proof rule for every syntactic form of *Core-Safe* expressions

$$\frac{\begin{array}{ll}\Gamma(C) = \overline{t_i}^n \to \rho \to t & \mathit{wellT}(\overline{t_i}^n, \rho, t) \\ e_2, \Sigma_T \vdash \theta \uplus [x_1 \mapsto \mu(t)] \leadsto t_2 & \mathit{argP}(\overline{\mu(t_i)}^n, \mu(\rho), \overline{a_i}^n, r, \theta)\end{array}}{\textbf{let } x_1 = C\ \overline{a_i}^n\ @\ r \textbf{ in } e_2, \Sigma_T \vdash \theta \leadsto t_2}\ \mathit{LET}_C$$

$$\frac{\begin{array}{ll}\Sigma_T(g) = \overline{t_i}^n \to \overline{\rho_j}^m \to t_g & \rho^g_{self} \notin \mathit{regions}\,(t_g) \\ \mathit{argP}\,(\overline{\mu(t_i)}^n, \overline{\mu(\rho_j)}^m, \overline{a_i}^n, \overline{r_j}^m, \theta) & t = \mu(t_g)\end{array}}{g\ \overline{a_i}^n @\ \overline{r_j}^m, \Sigma_T \vdash \theta \leadsto t}\ \mathit{APP}$$

- $\rho^g_{self} \notin \mathit{regions}\,(t_g)$ is the key property guaranteeing that dangling pointers are not created when returning from a function application

Theorem (soundness)

If $e, \Sigma_T \vdash \theta \leadsto t$ *then* $e, \Sigma_T : [\![\theta, t]\!]$.

Absence of dangling pointers: certificate generation

		Expression	L	Γ
e_1	$\stackrel{\text{def}}{=}$	$\mathit{unshuffle}\ x_{50}\ @\ r_2\ r_1\ \mathit{self}$	$\{x_{50}\}$	$[x_{50} : d,\ x_{34} : r]$
e_2	$\stackrel{\text{def}}{=}$	x_{45}	$\{x_{45}\}$	$[x_{45} : s,\ x_{34} : r]$
e_3	$\stackrel{\text{def}}{=}$	$\textbf{case}\ x_{40}\ \textbf{of}\ (x_{45}, x_{46}) \rightarrow e_2$	$\{x_{40}\}$	$[x_{40} : s,\ x_{34} : r]$
e_4	$\stackrel{\text{def}}{=}$	x_{48}	$\{x_{48}\}$	$[x_{48} : s,\ x_{34} : r]$
e_5	$\stackrel{\text{def}}{=}$	$\textbf{case}\ x_{40}\ \textbf{of}\ (x_{47}, x_{48}) \rightarrow e_4$	$\{x_{40}\}$	$[x_{40} : s,\ x_{34} : r]$
e_6	$\stackrel{\text{def}}{=}$	x_{39}	$\{x_{39}\}$	$[x_{39} : s,\ x_{34} : r]$
e_7	$\stackrel{\text{def}}{=}$	$\textbf{let}\ x_{39} = (x_{38}, x_{15})\ @\ r_3\ \textbf{in}\ e_6$	$\{x_{15}, x_{38}\}$	$[x_{15} : s,\ x_{38} : s,\ x_{34} : r]$
e_8	$\stackrel{\text{def}}{=}$	$\textbf{let}\ x_{38} = x_{49} : x_{16}\ @\ r_1\ \textbf{in}\ e_7$	$\{x_{15}, x_{16}, x_{49}\}$	$[x_{15} : s,\ x_{16} : s,\ x_{49} : s,\ x_{34} : r]$
e_9	$\stackrel{\text{def}}{=}$	$\textbf{let}\ x_{16} = e_5\ \textbf{in}\ e_4$	$\{x_{15}, x_{40}, x_{49}\}$	$[x_{15} : s,\ x_{40} : s,\ x_{49} : s,\ x_{34} : r]$
e_{10}	$\stackrel{\text{def}}{=}$	$\textbf{let}\ x_{15} = e_3\ \textbf{in}\ e_2$	$\{x_{40}, x_{49}\}$	$[x_{40} : s,\ x_{49} : s,\ x_{34} : r]$
e_{11}	$\stackrel{\text{def}}{=}$	$\textbf{let}\ x_{40} = e_1\ \textbf{in}\ e_{10}$	$\{x_{49}, x_{50}\}$	$[x_{49} : s,\ x_{50} : d,\ x_{34} : r]$
e_{12}	$\stackrel{\text{def}}{=}$	x_{37}	$\{x_{37}\}$	$[x_{37} : s,\ x_{34} : r]$
e_{13}	$\stackrel{\text{def}}{=}$	$\textbf{let}\ x_{37} = (x_{36}, x_{35})\ @\ r_3\ \textbf{in}\ e_{12}$	$\{x_{35}, x_{36}\}$	$[x_{35} : s,\ x_{36} : s,\ x_{34} : r]$
e_{14}	$\stackrel{\text{def}}{=}$	$\textbf{let}\ x_{35} = [\,]\ @\ r_2\ \textbf{in}\ e_{13}$	$\{x_{36}\}$	$[x_{36} : s,\ x_{34} : r]$
e_{15}	$\stackrel{\text{def}}{=}$	$\textbf{let}\ x_{36} = [\,]\ @\ r_1\ \textbf{in}\ e_{14}$	$\{\,\}$	$[x_{34} : r]$
e_{16}	$\stackrel{\text{def}}{=}$	$\textbf{case!}\ x_{34}\ \textbf{of}\ \{x_{49} : x_{50} \rightarrow e_{11}; [\,] \rightarrow e_{15}\}$	$\{x_{34}\}$	$[x_{34} : d]$

$$
\begin{array}{lll}
\theta_{16} & \stackrel{\text{def}}{=} & [x_{34} : [a]@\rho_4,\ r_1 : \rho_1,\ r_2 : \rho_2,\ r_3 : \rho_3, self : \rho_{self}] \\
\theta_{15} & \stackrel{\text{def}}{=} & \theta_{16} \\
\theta_{14} & \stackrel{\text{def}}{=} & \theta_{15} + [x_{36} : [a]@\rho_1] \\
\theta_{13} & \stackrel{\text{def}}{=} & \theta_{14} + [x_{35} : [a]@\rho_2] \\
\theta_{12} & \stackrel{\text{def}}{=} & \theta_{13} + [x_{37} : ([a]@\rho_1, [a]@\rho_2)@\rho_3] \\
\theta_{11} & \stackrel{\text{def}}{=} & \theta_{16} + [x_{49} : a,\ x_{50} : [a]@\rho_4] \\
\theta_{10} & \stackrel{\text{def}}{=} & \theta_{11} + [x_{40} : ([a]@\rho_2, [a]@\rho_1)@\rho_{self}] \\
\theta_9 & \stackrel{\text{def}}{=} & \theta_{10} + [x_{15} : [a]@\rho_2] \\
\theta_8 & \stackrel{\text{def}}{=} & \theta_9 + [x_{16} : [a]@\rho_1] \\
\theta_7 & \stackrel{\text{def}}{=} & \theta_8 + [x_{38} : [a]@\rho_1] \\
\theta_6 & \stackrel{\text{def}}{=} & \theta_7 + [x_{39} : ([a]@\rho_1, [a]@\rho_2)@\rho_3] \\
\theta_5 & \stackrel{\text{def}}{=} & \theta_9 \\
\theta_4 & \stackrel{\text{def}}{=} & \theta_5 + [x_{47} : [a]@\rho_2,\ x_{48} : [a]@\rho_1] \\
\theta_3 & \stackrel{\text{def}}{=} & \theta_{10} \\
\theta_2 & \stackrel{\text{def}}{=} & \theta_3 + [x_{45} : [a]@\rho_2,\ x_{46} : [a]@\rho_1] \\
\theta_1 & \stackrel{\text{def}}{=} & \theta_{11}
\end{array}
\qquad
\begin{array}{lll}
t_{16} & \stackrel{\text{def}}{=} & ([a]@\rho_1, [a]@\rho_2)@\rho_3 \\
t_{15} & \stackrel{\text{def}}{=} & ([a]@\rho_1, [a]@\rho_2)@\rho_3 \\
t_{14} & \stackrel{\text{def}}{=} & ([a]@\rho_1, [a]@\rho_2)@\rho_3 \\
t_{13} & \stackrel{\text{def}}{=} & ([a]@\rho_1, [a]@\rho_2)@\rho_3 \\
t_{12} & \stackrel{\text{def}}{=} & ([a]@\rho_1, [a]@\rho_2)@\rho_3 \\
t_{11} & \stackrel{\text{def}}{=} & ([a]@\rho_1, [a]@\rho_2)@\rho_3 \\
t_{10} & \stackrel{\text{def}}{=} & ([a]@\rho_1, [a]@\rho_2)@\rho_3 \\
t_9 & \stackrel{\text{def}}{=} & ([a]@\rho_1, [a]@\rho_2)@\rho_3 \\
t_8 & \stackrel{\text{def}}{=} & ([a]@\rho_1, [a]@\rho_2)@\rho_3 \\
t_7 & \stackrel{\text{def}}{=} & ([a]@\rho_1, [a]@\rho_2)@\rho_3 \\
t_6 & \stackrel{\text{def}}{=} & ([a]@\rho_1, [a]@\rho_2)@\rho_3 \\
t_5 & \stackrel{\text{def}}{=} & [a]@\rho_1 \\
t_4 & \stackrel{\text{def}}{=} & [a]@\rho_1 \\
t_3 & \stackrel{\text{def}}{=} & [a]@\rho_2 \\
t_2 & \stackrel{\text{def}}{=} & [a]@\rho_2 \\
t_1 & \stackrel{\text{def}}{=} & ([a]@\rho_2, [a]@\rho_1)@\rho_{self}
\end{array}
$$

$$
\begin{array}{lll}
\mu_1 & \stackrel{\text{def}}{=} & \{a \mapsto a,\ \rho_4 \mapsto \rho_4,\ \rho_1 \mapsto \rho_2,\ \rho_2 \mapsto \rho_1,\ \rho_3 \mapsto \rho_{self}\} \\
\mu_3 & \stackrel{\text{def}}{=} & \{a_1 \mapsto [a]@\rho_2,\ a_2 \mapsto [a]@\rho_1,\ \rho_1 \mapsto \rho_{self}\} \\
\mu_7 & \stackrel{\text{def}}{=} & \{a_1 \mapsto [a]@\rho_1,\ a_2 \mapsto [a]@\rho_2,\ \rho_1 \mapsto \rho_3\} \\
\mu_8 & \stackrel{\text{def}}{=} & \{a \mapsto a,\ \rho_1 \mapsto \rho_1\} \\
\mu_{14} & \stackrel{\text{def}}{=} & \{a \mapsto a,\ \rho_1 \mapsto \rho_2\} \\
\mu_{16} & \stackrel{\text{def}}{=} & \{a \mapsto a,\ \rho_1 \mapsto \rho_4\}
\end{array}
\qquad
\begin{array}{lll}
\Gamma((,)) & = & a_1 \to a_2 \to \rho_1 \to (a_1, a_2)@\rho_1 \\
\Gamma([\,]) & = & \rho_1 \to [a]@\rho_1 \\
\Gamma(:) & = & a \to [a]@\rho_1 \to \rho_1 \to [a]@\rho_1 \\
 & & \mu_5 = \mu_3 \\
 & & \mu_{13} = \mu_7 \\
 & & \mu_{15} = \mu_8
\end{array}
$$

Semantic notion of bound satisfaction

Definition

Let $f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f$ be the context function, and e a sub-expression of e_f. We say that e satisfies the bound (Δ, μ, σ) in the context of θ, ϕ, and td, denoted $\theta, \phi, td \rhd_f e \models [\![(\Delta, \mu, \sigma)]\!]$, if:

$$valid_f\ \theta\ \phi \rightarrow P_{dom} \wedge (\forall E\ h\ k\ h'\ v\ \eta\ \delta\ m\ s\ \overline{s_i}^n\ .\ P_{\Downarrow} \wedge P_{dyn} \wedge P_{size} \wedge P_{\eta} \rightarrow P_{\Delta} \wedge P_{\mu} \wedge P_{\sigma})$$

$$\begin{array}{lcl}
P_{dom} & \stackrel{\text{def}}{=} & dom\ \Delta = R_f \cup \{\rho^f_{self}\} \\
P_{\Downarrow} & \stackrel{\text{def}}{=} & E \vdash (h,k), td, e \Downarrow (h',k), v, (\delta, m, s) \\
P_{dyn} & \stackrel{\text{def}}{=} & (\overline{x_i}^n \cup fv\ \ e \cup \overline{r_j}^m \cup self) \subseteq dom\ E \wedge dom\ \eta = dom\ \Delta \\
P_{size} & \stackrel{\text{def}}{=} & \forall i \in \{1..n\}\ .\ s_i = size(h, E\ x_i) \\
P_{\eta} & \stackrel{\text{def}}{=} & admissible\,(\eta, k) \\
P_{\Delta} & \stackrel{\text{def}}{=} & \forall j \in \{0 \ldots k\}\ .\ \sum_{\eta\ \ \rho = j}\ \Delta\ \rho\ \overline{s_i}^n \geq \delta\ j \\
P_{\mu} & \stackrel{\text{def}}{=} & \mu\ \overline{s_i}^n \geq m \\
P_{\sigma} & \stackrel{\text{def}}{=} & \sigma\ \overline{s_i}^n \geq s
\end{array}$$

The *Rec* Proof-Rule

$$\frac{(f\ \overline{x_i}^{l} @\ \overline{r_j}^{q} = e_f) \in \Sigma_D \qquad \theta, \phi, l+q \rhd_f e_f, \Sigma \uplus \{f \mapsto (\Delta, \mu, \sigma)\} \vdash (\Delta', \mu', \sigma') \qquad (\lfloor \Delta' \rfloor, \mu', \sigma') \sqsubseteq (\Delta, \mu, \sigma)}{\theta, \phi, l+q \rhd_f e_f, \Sigma \vdash (\Delta', \mu', \sigma')}\ [Rec]$$

By $\lfloor \Delta \rfloor$ we denote the projection of Δ over R_f, obtained by removing the region ρ^f_{self} from Δ.

In words, its says that if a triple (Δ, μ, σ) (obtained by whatever means) is to be proved a safe upper bound for the recursive function f, a sufficient condition is:

1. Introduce (Δ, μ, σ) in the bound environment Σ.
2. Derive a triple (Δ', μ', σ') for f's body by using the remaining proof-rules.
3. Prove $(\lfloor \Delta' \rfloor, \mu', \sigma') \sqsubseteq (\Delta, \mu, \sigma)$.

The only difficulty is proving (3). For polynomial functions this can be done by converting it into a decision problem of Tarski's theory of closed real fields.

Theorem (Soundness)

If $\theta, \phi, td \rhd_f e, \Sigma \vdash (\Delta, \mu, \sigma)$, *then* $\theta, \phi, td \rhd_f e, \Sigma \models [\![(\Delta, \mu, \sigma)]\!]$

The proof amounts to 4 500 Isabelle/HOL lines. The main steps are:

1. Restricted big-step semantics with an upper bound n to the longest chain of f's recursive calls: $E \vdash (h, k), td, e \Downarrow_{f,n} (h', k), v, r$.
2. Define appropriate notions of satisfaction $\theta, \phi, td \rhd_f e \models_{f,n} [\![(\Delta, \mu, \sigma)]\!]$, validity $\models_{f,n} \Sigma$, and conditional validity $\theta, \phi, td \rhd_f e, \Sigma \models_{f,n} [\![(\Delta, \mu, \sigma)]\!]$ in which the longest chain of f's recursive calls is bounded by n.
3. Prove $\forall n \,.\, \theta, \phi, td \rhd_f e, \Sigma \models_{f,n} [\![(\Delta, \mu, \sigma)]\!] \Rightarrow \theta, \phi, td \rhd_f e, \Sigma \models [\![(\Delta, \mu, \sigma)]\!]$
4. By induction on the $\vdash$ derivation, and by cases on the last rule applied, prove: $\theta, \phi, td \rhd_f e, \Sigma \vdash (\Delta, \mu, \sigma) \Rightarrow \forall n \,.\, \theta, \phi, td \rhd_f e, \Sigma \models_{f,n} [\![(\Delta, \mu, \sigma)]\!]$.

Restricting the class of functions

The proof-rules presented are valid whatever the monotonic functions considered for describing sizes and costs are. For certification we restrict ourselves to:

Max-Poly

The class **Max-Poly** over $\overline{x}^n$ is the smallest set of monotonic expressions containing constants in $\mathbb{R}^+$, variables $y \in \overline{x}^n$, and closed under the operations $\{+, *, \sqcup\}$. We will call any element of **Max-Poly** a max-poly.

- A max-poly function is a function of the form $\lambda \overline{x}^n .\, p$ in $(\mathbb{R}^+)^n \rightarrow \mathbb{R}^+$, where p is a max-poly over $\overline{x}^n$.
- $\{+, *, \sqcup\}$ are commutative and associative, and $+$ and $*$ distribute over $\sqcup$ in $\mathbb{R}^+$. Any max-poly (max-poly function) can be normalized to $p_1 \sqcup \ldots \sqcup p_n$, where p_i are ordinary polynomials (poly-functions).
- For practical purposes we use atomic guarded functions (AGF):

$$[G \rightarrow f] \stackrel{\text{def}}{=} \lambda \overline{x}^n .\, \begin{cases} -\infty & \text{if } \neg G(\overline{x}^n) \\ f(\overline{x}^n) & \text{if } G(\overline{x}^n) \end{cases}$$

where G is a guard of the form $\bigwedge_{i=1}^{n} (p_i \geq k_i)$, $k_i \in \mathbb{R}^+$, and $p_i(\overline{x}^n), f(\overline{x}^n)$ are multivariate max-polys.

Deciding $p \sqsubseteq q$ by Tarski's decision method

- Operating with AGFs satisfies the following properties (a, b, c denote AGFs):
  1. $[G_1 \rightarrow f_1] + [G_2 \rightarrow f_2] = [G_1 \wedge G_2 \rightarrow f_1 + f_2]$
  2. $[G_1 \rightarrow f_1] * [G_2 \rightarrow f_2] = [G_1 \wedge G_2 \rightarrow f_1 * f_2]$
  3. $[G_1 \rightarrow [G_2 \rightarrow f]] = [G_1 \wedge G_2 \rightarrow f]$
  4. $(a \sqcup b) + c = (a + c) \sqcup (b + c)$
  5. $(a \sqcup b) * c = (a * c) \sqcup (b * c)$
- Any function obtained by combining AGFs with $\{+, *, \sqcup\}$ can be normalized:

$$[G_1 \rightarrow f_1] \sqcup \ldots \sqcup [G_l \rightarrow f_l]$$

- Inequalities of the form:

$$[G_1 \rightarrow f_1] \sqcup \ldots \sqcup [G_l \rightarrow f_l] \sqsubseteq [G'_1 \rightarrow f'_1] \sqcup \ldots \sqcup [G'_m \rightarrow f'_m]$$

  can be decided by:

$$\forall \overline{x}^n \, . \bigwedge_{i=1}^{l} \bigvee_{j=1}^{m} [G_i \rightarrow f_i] \sqsubseteq [G'_j \rightarrow f'_j]$$

- The elementary operation of comparing two AGFs can be expressed as:

$$[G \rightarrow f] \sqsubseteq [G' \rightarrow f'] \equiv G \rightarrow (G' \wedge f \leq f')$$

  where $f \leq f'$ consists of comparing two max-polys.

The merge function

```
merge x y @ r = case x of
                 []    -> y
                 ex:x' -> case y of
                           []    -> x
                           ey:y' -> let c = ex <= ey in
                                    case c of
                                     True  -> let z1 = merge x' y @ r in
                                              ex:z1 @ r
                                     False -> let z2 = merge x y' @ r in
                                              ey:z2 @ r
```

- Let us assume that the candidate memory bound obtained by the *Safe* compiler for `merge` live heap, assuming $\theta\ r = \rho$, is:

$$
\begin{array}{rcll}
\Delta_{merge}\ \rho & = & [x \geq 2 \wedge y \geq 1 \rightarrow x + y - 2] & \text{-- } A \\
 & \sqcup & [x \geq 1 \wedge y \geq 2 \rightarrow x + y - 2] & \text{-- } B \\
 & \sqcup & [x \geq 1 \wedge y \geq 1 \rightarrow 0] & \text{-- } C
\end{array}
$$

- This signature gives 0 cells when both lists are empty, i.e. $x = 1 \wedge y = 1$, and $x + y - 2$ cells otherwise.

The `merge` function (2)

- Now, we introduce this signature in the environment Σ and apply the remaining proof rules.
- The *Cons* proof-rule gets $[x \geq 1 \wedge y \geq 1 \rightarrow 1]$ charged to region ρ.
- The *Let* rule asks for the addition of the involved Δ's.
- *Case* rule asks for the $\sqcup$ of the branches.
- The sizes of the internal call arguments are $x' = x - 1$ and $y' = y - 1$,
- We obtain as derived bound the following function:

$$
\begin{array}{lcl}
\Delta'_{merge}\ \rho & = & [x \geq 1 \wedge y \geq 1 \rightarrow 0] \\
& \sqcup & [x \geq 1 \wedge y \geq 1 \rightarrow 0] \\
& \sqcup & (([x - 1 \geq 2 \wedge y \geq 1 \rightarrow x - 1 + y - 2] \\
& & \quad \sqcup [x - 1 \geq 1 \wedge y \geq 2 \rightarrow x - 1 + y - 2] \\
& & \quad \sqcup [x - 1 \geq 1 \wedge y \geq 1 \rightarrow 0]) + [x \geq 1 \wedge y \geq 1 \rightarrow 1]) \\
& \sqcup & (([x \geq 2 \wedge y - 1 \geq 1 \rightarrow x + y - 1 - 2] \\
& & \quad \sqcup [x \geq 1 \wedge y - 1 \geq 2 \rightarrow x + y - 1 - 2] \\
& & \quad \sqcup [x \geq 1 \wedge y - 1 \geq 1 \rightarrow 0]) + [x \geq 1 \wedge y \geq 1 \rightarrow 1])
\end{array}
$$

The `merge` function (3)

- After normalization and simplification, we get:

$$
\begin{array}{lcll}
\Delta'_{merge}\ \rho & = & [x \geq 1 \wedge y \geq 1 \rightarrow 0] & \text{-- } C' \\
 & \sqcup & [x \geq 3 \wedge y \geq 1 \rightarrow x + y - 2] \sqcup [x \geq 2 \wedge y \geq 1 \rightarrow 1] & \text{-- } A' \sqcup A'' \\
 & \sqcup & [x \geq 2 \wedge y \geq 2 \rightarrow x + y - 2] & \text{-- } D' \\
 & \sqcup & [x \geq 1 \wedge y \geq 3 \rightarrow x + y - 2] \sqcup [x \geq 1 \wedge y \geq 2 \rightarrow 1] & \text{-- } B' \sqcup B''
\end{array}
$$

- Obviously, for all x, y we get $C' \sqsubseteq C$, $A' \sqsubseteq A$, $B' \sqsubseteq B$, and both $D' \sqsubseteq A$ and $D' \sqsubseteq B$
- It is also easy to convince ourselves that A'' is dominated by A, and B'' is dominated by B.
- Then, $\lfloor \Delta'_{merge} \rfloor \sqsubseteq \Delta_{merge}$ holds.

The `msort` function

```
msort x @ r = case x of
             []    -> x
             ex:x' -> case x' of
                        []  -> x
                        _:_ -> let (x1,x2) = unshuffle x @ self self in
                               let z1      = msort x1 @ r            in
                               let z2      = msort x2 @ r            in
                               merge z1 z2 @ r
```

The candidate `msort` live memory bound inferred by our compiler, assuming Δ_{merge} as above, and the following bound obtained for `unshuffle`:

$$\Delta_{unshuffle} = \left[\begin{array}{l} \rho_1 \mapsto [x \geq 2 \rightarrow x+1] \sqcup [x \geq 1 \rightarrow 2] \\ \rho_2 \mapsto [x \geq 2 \rightarrow x] \sqcup [x \geq 1 \rightarrow 1] \end{array}\right]$$

is:

$$\Delta_{msort}\ \rho = [x \geq 2 \rightarrow \frac{4}{3}x^2 - 3x] \sqcup [x \geq 1 \rightarrow 0]$$

Introducing this candidate bound in the environment, applying the proof-rules, normalizing, and simplifying lead to:

$$\Delta'_{msort} = \left[\begin{array}{l} \rho \mapsto [x \geq 3 \rightarrow \frac{2}{3}x^2 - \frac{3}{2}x - \frac{17}{6}] \sqcup [x \geq 1 \rightarrow 0] \\ \rho_{self} \mapsto [x \geq 2 \rightarrow 2x+1] \sqcup [x \geq 1 \rightarrow 3] \end{array}\right]$$

The `msort` function (2)

- Notice that the charges to the *self* region are not needed in the comparison $\lfloor \Delta'_{msort} \rfloor \sqsubseteq \Delta_{msort}$. The relevant inequality is then:

$$\forall x \,. \quad \ldots \left(x \geq 3 \rightarrow x \geq 2 \wedge (\frac{2}{3}x^2 - \frac{3}{2}x - \frac{17}{6} \leq \frac{4}{3}x^2 - 3x) \right) \ldots$$

- When this formula is given to QEPCAD, it answers *True* in about 100 msec. Then, $\lfloor \Delta'_{msort} \rfloor \sqsubseteq \Delta_{msort}$ holds.

Outline

Summary of the research area

Static analysis

- Abstract interpretation
- Type systems
- Linear programming
- Constraint solvers

Certification

- Formal verification
- Language semantics
- Certified compilation
- Proof-assistants
- Computer algebra tools