Abstract

We describe a new automatic static analysis for determining upper-bound functions on the use of quantitative resources for strict, higher-order, polymorphic, recursive programs dealing with possibly-aliased data. Our analysis is a variant of Tarjan’s manual amortised cost analysis technique. We use a type-based approach, exploiting linearity to allow inference, and place a new emphasis on the number of references to a data object. The bounds we infer depend on the sizes of the various inputs to a program. They thus expose the impact of specific inputs on the overall cost behaviour. The key novel aspect of our work is that it deals directly with polymorphic higher-order functions without requiring source-level transformations that could alter resource usage. We thus obtain safe and accurate compile-time bounds. Our work is generic in that it deals with a variety of quantitative resources. We illustrate our approach with reference to dynamic memory allocations/deallocations, stack usage, and worst-case execution time, using metrics taken from a real implementation on a simple micro-controller platform that is used in safety-critical automotive applications.
Static Determination of Quantitative Resource Usage for Higher-Order Programs

Steffen Jost – St Andrews University, UK
Hans-Wolfgang Loidl – Heriot-Watt University, UK
Kevin Hammond – St Andrews University, UK
Martin Hofmann – Ludwig-Maximilians University, Germany

Madrid, Spain
January 21, 2010
Resource Usage Analysis

- **Static**  No change of runtime behaviour
- **Automatic**  No programmer annotations required
- **Reliable**  Delivers formally proven upper-bound functions
- **Generic**  Bounding usage of heap, stack, time, calls, . . .
- **Efficient**  Analyse source while programming
- **Modular**  Analyse libraries only once

▶ Mutual recursive higher-order functional language
   (Implementation at http://www.embounded.org)

▶ Restricted to linear cost bounds  Not linearly typed!
Resource Usage Analysis

**Static**  No change of runtime behaviour

**Automatic**  No programmer annotations required

**Reliable**  Delivers formally proven upper-bound functions

**Generic**  Bounding usage of heap, stack, time, calls, …

**Efficient**  Analyse source while programming

**Modular**  Analyse libraries only once

- Mutual recursive higher-order functional language
  (Implementation at [http://www.embounded.org](http://www.embounded.org))

- Restricted to linear cost bounds  Not linearly typed!
Underlying principle

Based on manual “Amortised Analysis” R.E. Tarjan’85

- Abstract state to single non-negative number: \( \Phi(\text{state}) \geq 0 \)
- Ensure for all transitions: actual cost \( \leq \Phi(\text{before}) - \Phi(\text{after}) \)
- Overall cost easy then to determine: \( \Phi(\text{initial state}) \)

Problems:

- Ingenuity required for application!

Our solution: Small additions to standard type system + LP Solving = Automatic analysis
Underlying principle

Based on manual “Amortised Analysis” R.E.Tarjan’85

- Abstract state to single non-negative number: \( \Phi(\text{state}) \geq 0 \)
- Ensure for all transitions: actual cost \( \leq \Phi(\text{before}) - \Phi(\text{after}) \)
- Overall cost easy then to determine: \( \Phi(\text{initial state}) \)

Problems:

- Ingenuity required for application!


\[ [\text{AA}] \text{ is often tremendously effective in practice.} \]

Unfortunately, \([\text{AA}]\) breaks in the presence of persistence.

Our solution: Small additions to standard type system + LP Solving

\[ = \text{Automatic analysis} \]

- Linear bounds allow automatic inference
Underlying principle

Based on manual “Amortised Analysis” R.E. Tarjan’85

- Abstract state to single non-negative number: $\Phi(\text{state}) \geq 0$
- Ensure for all transitions: actual cost $\leq \Phi(\text{before}) - \Phi(\text{after})$
- Overall cost easy then to determine: $\Phi(\text{initial state})$

Problems:
- Ingenuity required for application!
- C. Okasaki, 1989, “Purely Functional Data Structures”:
  
  [AA] is often tremendously effective in practice.
  Unfortunately, [AA] breaks in the presence of persistence.

Our solution: Small additions to standard type system + LP Solving = Automatic analysis

- Linear bounds allow automatic inference
- Potential assigned per reference allows persistence
  (no ref. counting, but explicit structural type rules required)
Underlying principle

Based on manual “Amortised Analysis” R.E.Tarjan’85

- Abstract state to single non-negative number: \( \Phi(\text{state}) \geq 0 \)
- Ensure for all transitions: actual cost \( \leq \Phi(\text{before}) − \Phi(\text{after}) \)
- Overall cost easy then to determine: \( \Phi(\text{initial state}) \)

Problems:

- Ingenuity required for application!
- C. Okasaki, 1989, “Purely Functional Data Structures”:
  
  \[ [AA] \text{ is often tremendously effective in practice.} \]
  
  \[ \text{Unfortunately, [AA] breaks in the presence of persistence.} \]

Our solution: Small additions to standard type system + LP Solving = Automatic analysis

- Linear bounds allow automatic inference
- Potential assigned per reference allows persistence
  
  (no ref. counting, but explicit structural type rules required)
Underlying principle

Based on manual “Amortised Analysis” R.E. Tarjan’85

- Abstract state to single non-negative number: \( \Phi(\text{state}) \geq 0 \)
- Ensure for all transitions: actual cost \( \leq \Phi(\text{before}) - \Phi(\text{after}) \)
- Overall cost easy then to determine: \( \Phi(\text{initial state}) \)

Problems:

- Ingenuity required for application!
- C. Okasaki, 1989, “Purely Functional Data Structures”:
  
  [AA] is often tremendously effective in practice.
  Unfortunately, [AA] breaks in the presence of persistence.

Our solution: Small additions to standard type system + LP Solving = Automatic analysis

- Linear bounds allow automatic inference
- Potential assigned per reference allows persistence (no ref. counting, but explicit structural type rules required)
Underlying principle

Based on manual “Amortised Analysis” R.E.Tarjan’85

- Abstract state to single non-negative number: \( \Phi(\text{state}) \geq 0 \)
- Ensure for all transitions: actual cost \( \leq \Phi(\text{before}) - \Phi(\text{after}) \)
- Overall cost easy then to determine: \( \Phi(\text{initial state}) \)

Problems:

- Ingenuity required for application!
- C. Okasaki, 1989, “Purely Functional Data Structures”:

  \[ [AA] \text{ is often tremendously effective in practice.} \]
  \[ \text{Unfortunately, [AA] breaks in the presence of persistence.} \]

Our solution: Small additions to standard type system + LP Solving

= Automatic analysis

- Linear bounds allow automatic inference
- Potential assigned per reference allows persistence
  (no ref. counting, but explicit structural type rules required)
**Data-dependent bounds**

**Example:** analysing space cost of function RedBlackInsert for node insertion into a balanced red-black tree yields

Worst-case Heap-units required to call RedBlackInsert:

\[ 20 + 10 \times X_1 + 18 \times X_2 \]

where

- \( X_1 \) = number of "Node" nodes at 1. input position
- \( X_2 \) = number of "Black" nodes at 1. input position

- ▶ Bounds are *data-dependent* and not size-dependent
- ▶ Bounds hold for *all* execution paths on any well-typed input, even for input that does not meet the intended specifications
- ▶ Cost-formulas derived from annotated types

\[
\begin{align*}
\text{int} \times \mu \alpha. \left\{ \text{Leaf:0 | Node:10, } \langle \{\text{Red:0 | Black:18}\}, \alpha, \text{int}, \alpha \rangle \right\} \\
\quad \rightarrow_{0}^{20} \mu \beta. \left\{ \text{Leaf:0 | Node:0, } \langle \{\text{Red:0 | Black:0}\}, \beta, \text{int}, \beta \rangle \right\}
\end{align*}
\]
Real-world examples

- Real-time control engineering problem (∼180 lines)
- Performed and measured using Renesas M32C/85U test board

Results:

- 36118 to 47635 clock cycles observed on typical iterations
- 63678 clock cycles inferred upper bound on WCET (33.7%)

- Stack Space: Exact Prediction!
- Dynamic Memory: Exact Prediction!

- Analysis generates 1115 linear constraints over 2214 variables, solved in 0.67s on 1.73Ghz Pentium M, 2MB cache
Why higher-order?

Transformation to first-order is unacceptable, because

**may change execution costs**
- destroy programmers’ intuition about cost
- unacceptable for resource-aware applications
- may affect whether analysis is successful

**destroys compositionality**
- restricts access to open-source libraries
- or first-order libraries
Challenges for H-O analysis: Resource Parametricity

Functions may admit several cost bounds, “best” depends on use

Example: zipping two lists to a (truncated) list of pairs

\[
\begin{align*}
\text{zip} : & \quad \text{list}(1537, \tau) \times \text{list}(0, \varrho) \quad \xrightarrow{763} \quad \text{list}(0, \tau \times \varrho) \\
\text{zip} : & \quad \text{list}(768.5, \tau) \times \text{list}(768.5, \varrho) \quad \xrightarrow{763} \quad \text{list}(0, \tau \times \varrho) \\
\text{zip} : & \quad \text{list}(0, \tau) \times \text{list}(1537, \varrho) \quad \xrightarrow{763} \quad \text{list}(0, \tau \times \varrho)
\end{align*}
\]

Solution: bundling constraints with function type

\[
\forall \{a, b, c\} \in \{a + b \geq 1537 + c\}.
\]

\[
\text{list}(a, \tau) \times \text{list}(b, \varrho) \quad \xrightarrow{763} \quad \text{list}(c, \tau \times \varrho)
\]

+ each function analysed only once
  ⇒ Libraries

+ best fitting bound automatically choosen

− application may cause exponential number of constraints

− issues similar to polymorphic recursion arise
Challenges for H-O analysis: Resource Parametricity

Functions may admit several cost bounds, “best” depends on use

*Example:* zipping two lists to a (truncated) list of pairs

\[
\begin{align*}
\text{zip} : & \quad \text{list}(1537, \tau) \times \text{list}(0, \varrho) \quad \xrightarrow{763_0} \quad \text{list}(0, \tau \times \varrho) \\
\text{zip} : & \quad \text{list}(768.5, \tau) \times \text{list}(768.5, \varrho) \quad \xrightarrow{763_0} \quad \text{list}(0, \tau \times \varrho) \\
\text{zip} : & \quad \text{list}(0, \tau) \times \text{list}(1537, \varrho) \quad \xrightarrow{763_0} \quad \text{list}(0, \tau \times \varrho)
\end{align*}
\]

*Solution:* bundling constraints with function type

\[
\forall \{a, b, c\} \in \{a + b \geq 1537 + c\}.
\text{list}(a, \tau) \times \text{list}(b, \varrho) \xrightarrow{763_0} \text{list}(c, \tau \times \varrho)
\]

+ each function analysed only once
+ best fitting bound automatically choosen
- application may cause exponential number of constraints
- issues similar to polymorphic recursion arise

\(\implies\) Libraries
Challenges for H-O analysis: Subtyping

∀\{a, b, c}\in\{a + b \geq 1537 + c\}.

\[
\text{list}(a, \tau) \times \text{list}(b, \varrho) \xrightarrow{\frac{763}{0}} \text{list}(c, \tau \times \varrho)
\]

\[\vdash\]

∀\{a, b, c, x, y\}\in\{a + b \geq 1537 + c, x \geq 763 + y\}.

\[
\text{list}(a, \tau) \times \text{list}(b, \varrho) \xrightarrow{\frac{x}{y}} \text{list}(c, \tau \times \varrho)
\]

\[\vdash\]

∀\{a, b, c, x, y\}\in\{a + b \geq 1537 + c, x \geq 600 + y, c \geq 100\}.

\[
\text{list}(a, \tau) \times \text{list}(b, \varrho) \xrightarrow{\frac{x}{y}} \text{list}(c, \tau \times \varrho)
\]

- Unlike resource parametricity, subtyping may lose precision
- Subtyping allows introduction of additional constraints

\[\phi \vdash A <: B\]

Additional constraints added as needed, no issue for inference

Likewise for Sharing
Challenges for H-O analysis: Subtyping

\[ \forall \{a, b, c \} \in \{a + b \geq 1537 + c\}. \]

\[ \text{list}(a, \tau) \times \text{list}(b, \varnothing) \xrightarrow{763} \text{list}(c, \tau \times \varnothing) \]

\[ \forall \{a, b, c, x, y \} \in \{a + b \geq 1537 + c, \ x \geq 763 + y\}. \]

\[ \text{list}(a, \tau) \times \text{list}(b, \varnothing) \xrightarrow{x} \text{list}(c, \tau \times \varnothing) \]

\[ \forall \{a, b, c, x, y \} \in \{a + b \geq 1537 + c, \ x \geq 600 + y, \ c \geq 100\}. \]

\[ \text{list}(a, \tau) \times \text{list}(b, \varnothing) \xrightarrow{x} \text{list}(c, \tau \times \varnothing) \]

- Unlike resource parametricity, subtyping may lose precision
- Subtyping allows introduction of additional constraints

\[ \phi \vdash A <: B \]

Additional constraints added as needed, no issue for inference
Challenges for H-O analysis: Subtyping

∀\{a, b, c\} \in \{a + b \geq 1537 + c\}.

\[
\text{list}(a, \tau) \times \text{list}(b, \varrho) \xrightarrow{763/0} \text{list}(c, \tau \times \varrho)
\]

\[
\Downarrow
\]

∀\{a, b, c, x, y\} \in \{a + b \geq 1537 + c, \ x \geq 763 + y\}.

\[
\text{list}(a, \tau) \times \text{list}(b, \varrho) \xrightarrow{x/y} \text{list}(c, \tau \times \varrho)
\]

\[
\Downarrow
\]

∀\{a, b, c, x, y\} \in \{a + b \geq 1537 + c, \ x \geq 600 + y, \ c \geq 100\}.

\[
\text{list}(a, \tau) \times \text{list}(b, \varrho) \xrightarrow{x/y} \text{list}(c, \tau \times \varrho)
\]

- Unlike resource parametricity, subtyping may lose precision
- Subtyping allows introduction of additional constraints

\[
\phi \vdash A <: B
\]

Additional constraints added as needed, no issue for inference likewise for Sharing
A Taste of Typing

\[
\text{dom}(\Gamma) = \text{FV}(e) \setminus x \quad \phi \cup \psi \Rightarrow \xi \quad \vec{r} \notin \text{FV}_\diamond(\Gamma) \cup \text{FV}_\diamond(\phi)
\]

\[
\Gamma, x:A \vdash \frac{\phi}{\phi} e : C \mid \xi \quad \phi \Rightarrow \bigcup_{D \in \text{ran}(\Gamma)} \forall (D \mid D, D)
\]

\[
\Gamma \quad \frac{\text{KmkFun}(\mid \Gamma \mid)}{0} \quad \lambda x.e : \forall \vec{r} \in \psi.A \frac{\phi}{\phi} C \mid \phi
\]

(ABS)

\[
\sigma(B) = A \frac{\phi}{\phi} C
\]

\[
\sigma : \vec{r} \to \text{CV} \text{ a substitution to fresh resource variables}
\]

\[
\frac{\sigma(\psi)}{x:A, y:\forall \vec{r} \in \psi.B \vdash q + \text{Kapp} + \text{Knext} \quad q' - \text{Kapp}' \quad yx : C \mid \sigma(\psi)}
\]

(APP)
A Taste of Typing

\[
\text{dom}(\Gamma) = \text{FV}(e) \setminus x \quad \phi \cup \psi \Rightarrow \xi \quad \vec{r} \notin \text{FV}_\bullet(\Gamma) \cup \text{FV}_\bullet(\phi)
\]

\[
\Gamma, x:A \models^q_{q'} e : C \mid \xi \quad \phi \Rightarrow \bigcup_{D \in \text{ran}(\Gamma)} \forall(D \mid D, D)
\]

\[
\Gamma \models^{\text{KmkFun}(\mid \Gamma \mid)}_0 \lambda x.e : \forall \vec{r} \in \psi. A \overset{q}{q'} C \mid \phi
\]  

(ABS)

- predictable (minimal) closure size
- split constraints of \(\xi\) to delayed \(\psi\) and applied now \(\phi\) constraints
- ensure newly bound variables are independent
- allow repeated function application by zero closure potential

\[
\sigma(B) = A \overset{q}{q'} C
\]

\[\sigma : \vec{r} \rightarrow \text{CV} \text{ a substitution to fresh resource variables}\]

\[
\text{x:A, y:\forall \vec{r} \in \psi.B} \models_{q' - \text{Kapp'}}^{q + \text{Kapp} + \text{Knext}} y x : C \mid \sigma(\psi)
\]  

(APP)
Function Closures

**Use-many times versus Use-once**

- Use-many times functions require closures with zero potential
- Use-once functions have no such restriction

  Closures with potential can be used for linear pairs

- Use-$n$-times functions types possible

*Conclusion:* No convincing example for non-zero potential closures

All our function closures are use-many times!

**Lambda-Abstraction versus Under-Application**

- Lambda-Abstraction allows static determination of closure size
- Incremental closures allow same for Under-Application

*Conclusion:* Either possible and available in implementation
Higher-order Example: \texttt{twice twice}

\[
\text{twice} :: (T \to T) \to (T \to T);
\text{twice } f \ x = f (f \ x);
\]

\[
\text{quad} :: (T \to T) \to (T \to T);
\text{quad } f \ x = \text{let } f' = \text{twice } f
\text{ in } \text{twice } f' \ x;
\]

Essentially \text{quad} \ f \ x = f (f (f (f x))),
but \text{quad} uses \texttt{twice twice} twice with different resource usage

Analysis instantaneously delivers exact results

\[
\text{Boxed Heap} \quad \text{quad} : \tau \xrightarrow{0} \tau \xrightarrow{5} \tau \quad \Rightarrow \quad \text{five cells for closure}
\]

\[
\text{quad succ} : \text{int} \xrightarrow{21} \text{int} \quad \Rightarrow \quad 21 = 5 + 4 \cdot 4
\]

\[
\text{Call Count} \quad \text{quad succ} : \text{int} \xrightarrow{4} \text{int} \quad \Rightarrow \quad \text{four calls to succ}
\]
Higher-order Example: Sum of Squares

Computing sum of squares: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100 ...]

---- map, (left-) fold over lists are standard
---- enumFromTo m n generates [m..n] (tail-recursive)

---- VARIANT 1: direct recursion (first-order)
sum_sqs’ n m s = if (m>n) then s else sum_sqs’ (n-1) m (s+(sq n));
sum_sqs n = sum_sqs’ n 1 0;

---- VARIANT 2: uses h-o fcts fold and map
sum xs = fold add 0 xs;
sum_sqs n = sum (map sq (enumFromTo 1 n));

---- VARIANT 3: uses h-o fcts unfold, fold and map
data maybenum = Nothing | Just num;
---- countdown subtracts one and returns result greater zero or Nothing
---- unfoldr generates list by repeated function application until Nothing
enum :: num -> [num]; -- generates [n,n-1..1]
enum n = if (n<1) then [] else n:(unfoldr countdown n);

sum_sqs :: num -> num;
sum_sqs n = sum (map sq (enum n));
Higher-order Example:
Computing sum of squares: \([1, 4, 9, 16, 25, 36, 49, 64, 81, \ldots]\)

<table>
<thead>
<tr>
<th></th>
<th>Calls</th>
<th>Heap</th>
<th>Stack</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{sum-of-squares} (variant 1: direct recursion, 1\textsuperscript{st} order)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Analysis</td>
<td>22</td>
<td>114</td>
<td>30</td>
<td>18091</td>
</tr>
<tr>
<td>Measured</td>
<td>22</td>
<td>108</td>
<td>30</td>
<td>16874</td>
</tr>
<tr>
<td>Ratio</td>
<td>1.00</td>
<td>1.06</td>
<td>1.00</td>
<td>1.07</td>
</tr>
<tr>
<td>\textit{sum-of-squares} (variant 2: with map &amp; fold)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Analysis</td>
<td>56</td>
<td>200</td>
<td>114</td>
<td>53612</td>
</tr>
<tr>
<td>Measured</td>
<td>56</td>
<td>200</td>
<td>112</td>
<td>42252</td>
</tr>
<tr>
<td>Ratio</td>
<td>1.00</td>
<td>1.00</td>
<td>1.02</td>
<td>1.27</td>
</tr>
<tr>
<td>\textit{sum-of-squares} (variant 3: map, fold &amp; unfold)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Analysis</td>
<td>71</td>
<td>272</td>
<td>181</td>
<td>77437</td>
</tr>
<tr>
<td>Measured</td>
<td>71</td>
<td>272</td>
<td>174</td>
<td>59560</td>
</tr>
<tr>
<td>Ratio</td>
<td>1.00</td>
<td>1.00</td>
<td>1.04</td>
<td>1.30</td>
</tr>
</tbody>
</table>

\textbf{Other examples covered:} general list folding, tree traversals, in-place insertion sort, evaluator for loop-free expressions
Research on Amortised Analysis

**Past:**
Heap usage for first-order language Hofmann & Jost, POPL’03
Java & Storeless Semantics Hofmann et al., ESOP’06, EACSL’09
Stack space usage & Depth Campbell, ESOP’09
WCET, Algebraic Datatypes & Cost Genericity Jost et al., FM’09

**Present:**
Higher-order & Polymorphism Jost et al., POPL’10

**Future:**
Non-linear bounds for lists Hofmann & Hoffmann, ESOP’10
Lazy evaluation Simões & Vasconcelos
Non-linear bounds in combination with Sized Types Jost et al.
Negative credits for monotone resources
Assigning credits to numeric types
Summary

Program analysis for

fully recursive eager higher-order functional language

▶ High quality cost bounds for various metrics successfully derived for many common program examples
▶ Higher-order is not a hindrance for analysis
▶ Very efficient “at the touch of a button”
▶ Generally succeeds for programs with linear resource usage

Limitations

▶ Analysis cannot be complete
▶ No safety guarantees for deallocation
▶ Only linear bounds inferred . . . thus far!
▶ Eager evaluation model . . . thus far!