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`)
- 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: $\textit{actual cost} \leq \Phi(\textit{before}) - \Phi(\textit{after})$
- Overall cost easy then to determine: $\Phi(\textit{initial state})$

Problems:

- Ingenuity required for application!

Our solution: Small additions to standard type sysem + 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: $\textit{actual cost} \leq \Phi(\textit{before}) - \Phi(\textit{after})$
- Overall cost easy then to determine: $\Phi(\textit{initial state})$

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 sysem + LP Solving
= 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: $\textit{actual cost} \leq \Phi(\textit{before}) - \Phi(\textit{after})$
- Overall cost easy then to determine: $\Phi(\textit{initial state})$

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 sysem + 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: $\textit{actual cost} \leq \Phi(\textit{before}) - \Phi(\textit{after})$
- Overall cost easy then to determine: $\Phi(\textit{initial state})$

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 sysem + 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: $\textit{actual cost} \leq \Phi(\textit{before}) - \Phi(\textit{after})$
- Overall cost easy then to determine: $\Phi(\textit{initial state})$

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 sysem + 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: $\textit{actual cost} \leq \Phi(\textit{before}) - \Phi(\textit{after})$
- Overall cost easy then to determine: $\Phi(\textit{initial state})$

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 sysem + 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

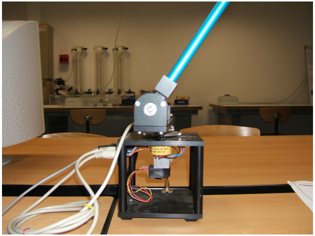
```
Worst-case Heap-units required to call RedBlackInsert:
  20 + 10*X1 + 18*X2
    where
      X1 = number of "Node" nodes at 1. input position
      X2 = number of "Black" nodes at 1. input position
```

- 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

$$\mathsf{int} \times \mu\alpha.\Big\{\mathtt{Leaf}{:}0 \mid \mathtt{Node}{:}10, \big\langle\{\mathtt{Red}{:}0 \mid \mathtt{Black}{:}18\}, \alpha, \mathsf{int}, \alpha\big\rangle\Big\} \xrightarrow[0]{20} \mu\beta.\Big\{\mathtt{Leaf}{:}0 \mid \mathtt{Node}{:}0, \big\langle\{\mathtt{Red}{:}0 \mid \mathtt{Black}{:}0\}, \beta, \mathsf{int}, \beta\big\rangle\Big\}$$

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

$$
\begin{aligned}
zip &: \mathrm{list}(1537,\tau)\times \mathrm{list}(0,\varrho) \xrightarrow[0]{763} \mathrm{list}(0,\tau\times\varrho)\\
zip &: \mathrm{list}(768.5,\tau)\times \mathrm{list}(768.5,\varrho) \xrightarrow[0]{763} \mathrm{list}(0,\tau\times\varrho)\\
zip &: \mathrm{list}(0,\tau)\times \mathrm{list}(1537,\varrho) \xrightarrow[0]{763} \mathrm{list}(0,\tau\times\varrho)
\end{aligned}
$$

Solution: bundling constraints with function type

$$
\forall\{a,b,c\}\in\{a+b\geq 1537+c\}.\quad \mathrm{list}(a,\tau)\times \mathrm{list}(b,\varrho) \xrightarrow[0]{763} \mathrm{list}(c,\tau\times\varrho)
$$

+ each function analysed only once $\Rightarrow$ 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

$$zip : \ \mathsf{list}(1537, \tau) \times \mathsf{list}(0, \varrho) \xrightarrow[0]{763} \mathsf{list}(0, \tau \times \varrho)$$

$$zip : \mathsf{list}(768.5, \tau) \times \mathsf{list}(768.5, \varrho) \xrightarrow[0]{763} \mathsf{list}(0, \tau \times \varrho)$$

$$zip : \ \mathsf{list}(0, \tau) \times \mathsf{list}(1537, \varrho) \xrightarrow[0]{763} \mathsf{list}(0, \tau \times \varrho)$$

Solution: bundling constraints with function type

$$\forall\{a, b, c\} \in \{a + b \geq 1537 + c\}. \quad \mathsf{list}(a, \tau) \times \mathsf{list}(b, \varrho) \xrightarrow[0]{763} \mathsf{list}(c, \tau \times \varrho)$$

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

Challenges for H-O analysis: Subtyping

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

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

$$\ddot{\vee}$$

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

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

$$\ddot{\vee}$$

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

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

- Unlike resource parametricty, 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

$$\forall\{a,b,c\}\in\{a+b\geq 1537+c\}.$$
$$\mathsf{list}(a,\tau)\times\mathsf{list}(b,\varrho)\xrightarrow[0]{763}\mathsf{list}(c,\tau\times\varrho)$$
$$\ddot{\vee}$$
$$\forall\{a,b,c,x,y\}\in\{a+b\geq 1537+c,\ x\geq 763+y\}.$$
$$\mathsf{list}(a,\tau)\times\mathsf{list}(b,\varrho)\xrightarrow[y]{x}\mathsf{list}(c,\tau\times\varrho)$$
$$\ddot{\vee}$$
$$\forall\{a,b,c,x,y\}\in\{a+b\geq 1537+c,\ x\geq 600+y,c\geq 100\}.$$
$$\mathsf{list}(a,\tau)\times\mathsf{list}(b,\varrho)\xrightarrow[y]{x}\mathsf{list}(c,\tau\times\varrho)$$

- Unlike resource parametricty, 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

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

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

$$\ddot{\vee}$$

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

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

$$\ddot{\vee}$$

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

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

- Unlike resource parametricty, 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

$$\frac{\mathrm{dom}(\Gamma) = \mathrm{FV}(e) \setminus x \qquad \phi \cup \psi \Rightarrow \xi \qquad \vec{r} \notin \mathrm{FV}_{\diamond}(\Gamma) \cup \mathrm{FV}_{\diamond}(\phi) \quad \Gamma, x{:}A \vdash^{q}_{q'} e : C \mid \xi \qquad \phi \Rightarrow \bigcup_{D \in \mathrm{ran}(\Gamma)} \curlyvee(D \,|\, D, D)}{\Gamma \vdash^{\mathtt{KmkFun}(|\Gamma|)}_{0} \lambda x.e : \forall \vec{r} {\in} \psi . A \xrightarrow[q']{q} C \mid \phi} \quad (\textsc{Abs})$$

$$\frac{\sigma(B) = A \xrightarrow[q']{q} C \quad \sigma : \vec{r} \to \mathsf{CV} \text{ a substitution to fresh resource variables}}{x{:}A,\ y{:}\forall \vec{r} {\in} \psi . B \vdash^{q + \mathtt{Kapp} + \mathtt{Knext}}_{q' - \mathtt{Kapp}'} y\, x : C \mid \sigma(\psi)} \quad (\textsc{App})$$

A Taste of Typing

$$\frac{\mathrm{dom}(\Gamma) = \mathrm{FV}(e) \setminus x \qquad \phi \cup \psi \Rightarrow \xi \qquad \vec{r} \notin \mathrm{FV}_{\diamond}(\Gamma) \cup \mathrm{FV}_{\diamond}(\phi) \\ \Gamma, x{:}A \vdash^{q}_{q'} e : C \mid \xi \qquad \phi \Rightarrow \bigcup_{D \in \mathrm{ran}(\Gamma)} \curlyvee(D \mid D, D)}{\Gamma \vdash^{\mathtt{KmkFun}(|\Gamma|)}_{0} \lambda x.e : \forall \vec{r} {\in} \psi . A \xrightarrow[q']{q} C \mid \phi} \text{(ABS)}$$

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

$$\frac{\sigma(B) = A \xrightarrow[q']{q} C \\ \sigma : \vec{r} \to \mathrm{CV} \text{ a substitution to fresh resource variables}}{x{:}A,\ y{:}\forall \vec{r} {\in} \psi . B \vdash^{q + \mathtt{Kapp} + \mathtt{Knext}}_{q' - \mathtt{Kapp}'} y\, x : C \mid \sigma(\psi)} \text{(APP)}$$

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: `twice` twice

```
twice :: ( T -> T ) -> ( T -> T );
twice f x = f (f x);

quad  :: ( T -> T ) -> ( T -> T );
quad  f x = let f' = twice f
            in  twice f' x;
```

Essentially `quad f x` $=$ `f (f (f (f x)))`,
but `quad` uses `twice` twice with different resource usage

Analysis instantaneously delivers exact results

Boxed Heap	`quad` : $\tau \xrightarrow[0]{0} \tau \xrightarrow[0]{5} \tau$	$\Rightarrow$	five cells for closure
	`quad succ` : $\mathsf{int} \xrightarrow[0]{21} \mathsf{int}$	$\Rightarrow$	$21 = 5 + 4 \cdot 4$
Call Count	`quad succ` : $\mathsf{int} \xrightarrow[0]{4} \mathsf{int}$	$\Rightarrow$	four calls to `succ`

Higher-order Example: Sum of Squares

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

```
---- 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: $[1, 4, 9, 16, 25, 36, 49, 64, 81, \ldots]$

	N = 10			
	Calls	Heap	Stack	Time
sum-of-squares (variant 1: direct recursion, 1^{st} order)				
Analysis	22	114	30	18091
Measured	22	108	30	16874
Ratio	1.00	1.06	1.00	1.07
sum-of-squares (variant 2: with map & fold)				
Analysis	56	200	114	53612
Measured	56	200	112	42252
Ratio	1.00	1.00	1.02	1.27
sum-of-squares (variant 3: map, fold & unfold)				
Analysis	71	272	181	77437
Measured	71	272	174	59560
Ratio	1.00	1.00	1.04	1.30

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 montone 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!