Automatic Amortised Analysis of Dynamic Memory Allocation for Lazy Functional Programs

Kevin Hammond
University of St Andrews, Scotland

Hugo Simoes, Steffen Jost, Pedro Vasconcelos, Mario Florido
Universidad do Porto, Ludwig-Maximilians Universität

http://www.embounded.org

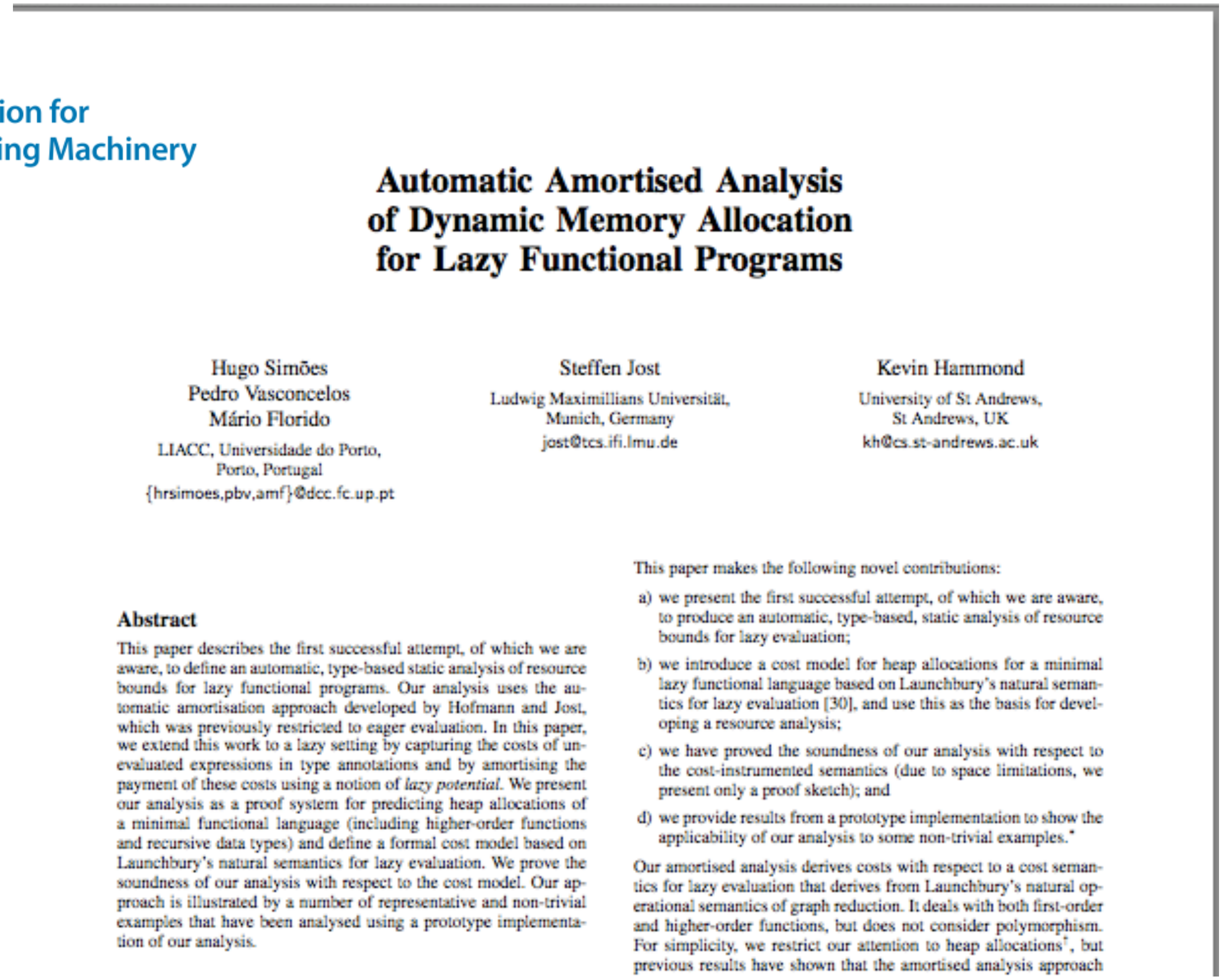
http://www.hume-lang.org

MMNet Workshop, Edinburgh, May2013

Full Technical Details can be found in

University of St Andrews

Association for Computing Machinery



Automatic Amortised Analysis of Dynamic Memory Allocation for Lazy Functional Programs

Hugo Simões
Pedro Vasconcelos
Mário Florido
LIACC, Universidade do Porto,
Porto, Portugal
{hrsimoes,pbv,amf}@dcc.fc.up.pt

Steffen Jost
Ludwig Maximillians Universität,
Munich, Germany
jost@tcs.ifi.lmu.de

Kevin Hammond
University of St Andrews,
St Andrews, UK
kh@cs.st-andrews.ac.uk

Abstract

This paper describes the first successful attempt, of which we are aware, to define an automatic, type-based static analysis of resource bounds for lazy functional programs. Our analysis uses the automatic amortisation approach developed by Hofmann and Jost, which was previously restricted to eager evaluation. In this paper, we extend this work to a lazy setting by capturing the costs of unevaluated expressions in type annotations and by amortising the payment of these costs using a notion of *lazy potential*. We present our analysis as a proof system for predicting heap allocations of a minimal functional language (including higher-order functions and recursive data types) and define a formal cost model based on Launchbury's natural semantics for lazy evaluation. We prove the soundness of our analysis with respect to the cost model. Our approach is illustrated by a number of representative and non-trivial examples that have been analysed using a prototype implementation of our analysis.

This paper makes the following novel contributions:

a) we present the first successful attempt, of which we are aware, to produce an automatic, type-based, static analysis of resource bounds for lazy evaluation;

b) we introduce a cost model for heap allocations for a minimal lazy functional language based on Launchbury's natural semantics for lazy evaluation [30], and use this as the basis for developing a resource analysis;

c) we have proved the soundness of our analysis with respect to the cost-instrumented semantics (due to space limitations, we present only a proof sketch); and

d) we provide results from a prototype implementation to show the applicability of our analysis to some non-trivial examples.*

Our amortised analysis derives costs with respect to a cost semantics for lazy evaluation that derives from Launchbury's natural operational semantics of graph reduction. It deals with both first-order and higher-order functions, but does not consider polymorphism. For simplicity, we restrict our attention to heap allocations†, but previous results have shown that the amortised analysis approach

ICFP 2012: ACM *International Conference on Functional Programming* Copenhagen Sept. 2012

Research Objectives

- Predict cost bounds for *Lazily Evaluated* Programs
  - Haskell is an example of such a language
- Initially heap allocation costs, but later
  - stack high-watermarks
  - deallocation costs
  - garbage collection
  - execution time
- Allows costs of pure functional programs to be determined *a-priori*
  - lazy functional programs are *bounded* and *predictable*
  - can be used for embedded systems
  - assists parallelisation

Why is this Important?

- Laziness supports compositionality and reuse
  - valuable for design, prototyping and ease of change
- Parallelism is much easier for pure (functional) programs
  - but laziness is necessary to support e.g. I/O
- Opens new application areas
  - e.g. embedded systems, real-time systems

John Hughes: “Why Functional Programming Matters”
The Computer Journal, 32(2):98-107, 1989.

(Incredibly) Simple Example

```
let x = [1..20000000] in
        …
```

- **How many heap cells are allocated?**
  - **assume each list element and each integer takes one cell**

Heap Allocation Costs

University of St Andrews

```
main = let x = [1..20000000] in
        print x
```

eager	lazy	string
40M	40M	40M

```
main = let x = [1..20000000] in
        print (x,x)
```

eager	lazy	string
40M	40M	80M

```
main = let x = [1..20000000] in
        print ()
```

eager	lazy	string
40M	0	0

Call by Value (eager evaluation)	**evaluate even if not needed**
Call by Need (lazy evaluation)	**evaluate only if needed**
Call by Name (string reduction)	**evaluate whenever needed**

Why Costing Lazy Evaluation is Hard

- In a lazy language, we need to know
  - which expressions are *needed*
  - whether expressions have previously been evaluated

- *This is dynamic*
  - *we need to know the evaluation context*
  - *we also need to know about sharing*

Key Technical Contributions

- First automatic static analysis for predicting lazy evaluation costs
  - Type-based
- Uses *lazy potential* to track evaluation status
  - thunk types allow pre-paid execution costs to be stored for later use
- Tracks sharing of evaluation costs
- Deals with higher-order functions
- Is cost-preserving: analysis doesn't alter execution costs

Prototype Implementation
http://kashmir.dcc.fc.up.pt/cgi/aalazy.cgi

The Core Language

$$\begin{array}{lcl} e & ::= & x \;\mid\; \lambda x.e \;\mid\; e\,x \\ & \mid & \text{let } x = e_1 \text{ in } e_2 \\ & \mid & \text{letcons } x = c(\vec{y}) \text{ in } e \\ & \mid & \text{match } e_0 \text{ with } c(\vec{x}) \text{ -> } e_1 \text{ otherwise } e_2 \end{array}$$

- **Based on Launchbury's 1993 Semantics for Lazy Evaluation**
  - **plus *letcons* to expose constructor allocations**
  - **function arguments are *normalised* (must be identifiers)**
    - **simplifies analysis without affecting power**
    - **easy transformation/compiler simplification**

Execution Example

$$\text{let } z = z \text{ in } (\lambda x.\, \lambda y.\, y)\ z$$

```
         let z = z in (λx . λy. y ) z
=>       (λx . λy. y ) z    (where z = z) NEW HEAP CELL (THUNK)
=>       (λy. y) [z/x]      (where z = z)
=        (λy.y)
```

only *let* and *letcons* allocate memory

one cell for a *thunk* *(let)*

one cell for a *constructor* *(letcons)*

Example Operational Semantics Rule

University of St Andrews

$$\frac{\ell \text{ is fresh} \qquad e_1' = e_1[\ell/x] \qquad e_2' = e_2[\ell/x] \\ \mathcal{H}[\ell \mapsto e_1'], \ldots \vdash e_2' \Downarrow w, \mathcal{H}'}{\mathcal{H}, \ldots \vdash \text{let } x = e_1 \text{ in } e_2 \Downarrow w, \mathcal{H}'} \quad (\text{LET}_{\Downarrow})$$

$H \vdash e \Downarrow w, H'$

means that given initial heap *H*, *e* reduces to *w*, with new heap *H'*

w* is either *λx.e* or *c(→y))

Corresponding Cost Rule

$$\frac{\ell \text{ is fresh} \qquad e_1' = e_1[\ell/x] \qquad e_2' = e_2[\ell/x] \qquad \mathcal{H}[\ell \mapsto e_1'], \ldots \vdash^{m}_{m'} e_2' \Downarrow w, \mathcal{H}'}{\mathcal{H}, \ldots \vdash^{m+1}_{m'} \text{let } x = e_1 \text{ in } e_2 \Downarrow w, \mathcal{H}'} \quad (\textsc{Let}_{\Downarrow})$$

Counts the number of heap allocations

m **is the potential before evaluation**

m' **is the potential remaining after evaluation**

Costing Example

$$\text{let } z = z \text{ in } (\lambda x.\, \lambda y.\, y)\; z$$

- **If evaluated eagerly, the cost will be infinite**
- **The semantics gives**

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{1}_{0} \text{let } z = z \text{ in } (\lambda x.\, \lambda y.\, y)\; z \Downarrow \lambda y.\, y, \mathcal{H}[\ell_3 \mapsto \ell_3]$$

- **This shows that we allocate precisely *1* cell (for the thunk z)**

Cost Rule for LetCons

$$\frac{\ell \text{ is fresh} \qquad y_i' = y_i[\ell/x] \qquad e' = e[\ell/x] \\ \mathcal{H}[\ell \mapsto c(\vec{y'})], \mathcal{S}, \mathcal{L} \vdash_{m'}^{m} e' \Downarrow w, \mathcal{H}'}{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash_{m'}^{m+1} \text{letcons } x = c(\vec{y}) \text{ in } e \Downarrow w, \mathcal{H}'} \ (\text{LETCONS}_{\Downarrow})$$

Counts the number of heap allocations

m **is the potential before evaluation**

m' **is the potential remaining after evaluation**

LetCons Example

University of St Andrews

let one = 1 in
letcons ones = Cons (one,ones) in
(λx. λy. y) ones

- The semantics gives a cost of 2
  - one for the let
  - one for the letcons

Building an Automatic Analysis

- **Type-based approach**
  - one type rule per language construct
  - annotated types associate the potential with constructs

- **Types and annotations are inferred automatically**
  - normal Hindley-Milner type inference determines the types
    - » and also exposes constraints on cost annnotations

  - the constraints are then solved using a standard linear solver (e.g. *lp-solve*)

University of St Andrews

Annotated Types

$$
\begin{array}{llll}
A,\ B, C & ::= & X & \text{variable} \\
 & \mid & \mu X.\{c_1 : (q_1, \vec{B}_1) \mid \cdots \mid c_n : (q_n, \vec{B}_n)\} & \text{data type} \\
 & \mid & A \xrightarrow[p']{p} B & \text{function} \\
 & \mid & \mathrm{T}^{p}_{p'}(A) & \text{thunk}
\end{array}
$$

q_i	**future costs inferred for processing data**	**(potential)**
p, p′	**cost annotations**	

Examples of data types:

$$
\begin{array}{rl}
\text{Naturals} & \mu X.\{\mathrm{Zero} : (q_0, ()) \mid \mathrm{Succ} : (q_s, \mathrm{T}^{?}_{?}(X))\} \\
\text{Lists} & \mu X.\{\mathrm{Nil} : (q_n, ()) \mid \mathrm{Cons} : (q_c, (\mathrm{T}^{?}_{?}(A), \mathrm{T}^{?}_{?}(X)))\} \\
\text{Maybe} & \mu X.\{\mathrm{Nothing} : (q_n, ()) \mid \mathrm{Just} : (q_j, \mathrm{T}^{?}_{?}(A))\}
\end{array}
$$

Key Type Rules

University of St Andrews

- **Full details in paper**
  - **including soundness proofs**

$$\frac{\Gamma \overset{q}{\underset{q'}{\vdash}} e_1 : A \qquad \Delta, x{:}\mathrm{T}^{q}_{q'}(A) \overset{p}{\underset{p'}{\vdash}} e_2 : C}{\Gamma, \Delta \overset{1+p}{\underset{p'}{\vdash}} \mathrm{let}\ x = e_1\ \mathrm{in}\ e_2 : C} \qquad (\textsc{Let}^{*})$$

$$\frac{\Delta, x{:}\mathrm{T}^{0}_{0}(A) \overset{p}{\underset{p'}{\vdash}} e : C \qquad A = \mu X.\{\cdots \mid c : (q, \vec{B}) \mid \cdots\}}{\vec{y}{:}\vec{B}, \Delta \overset{1+p+q}{\underset{p'}{\vdash}} \mathrm{letcons}\ x = c(\vec{y})\ \mathrm{in}\ e : C} \qquad (\textsc{Letcons}^{*})$$

$$\frac{}{x{:}\mathrm{T}^{p}_{p'}(A) \overset{p}{\underset{p'}{\vdash}} x : A} \qquad (\textsc{Var})$$

Kevin Hammond, University of St Andrews

EmBounded

Prepay Type Rule

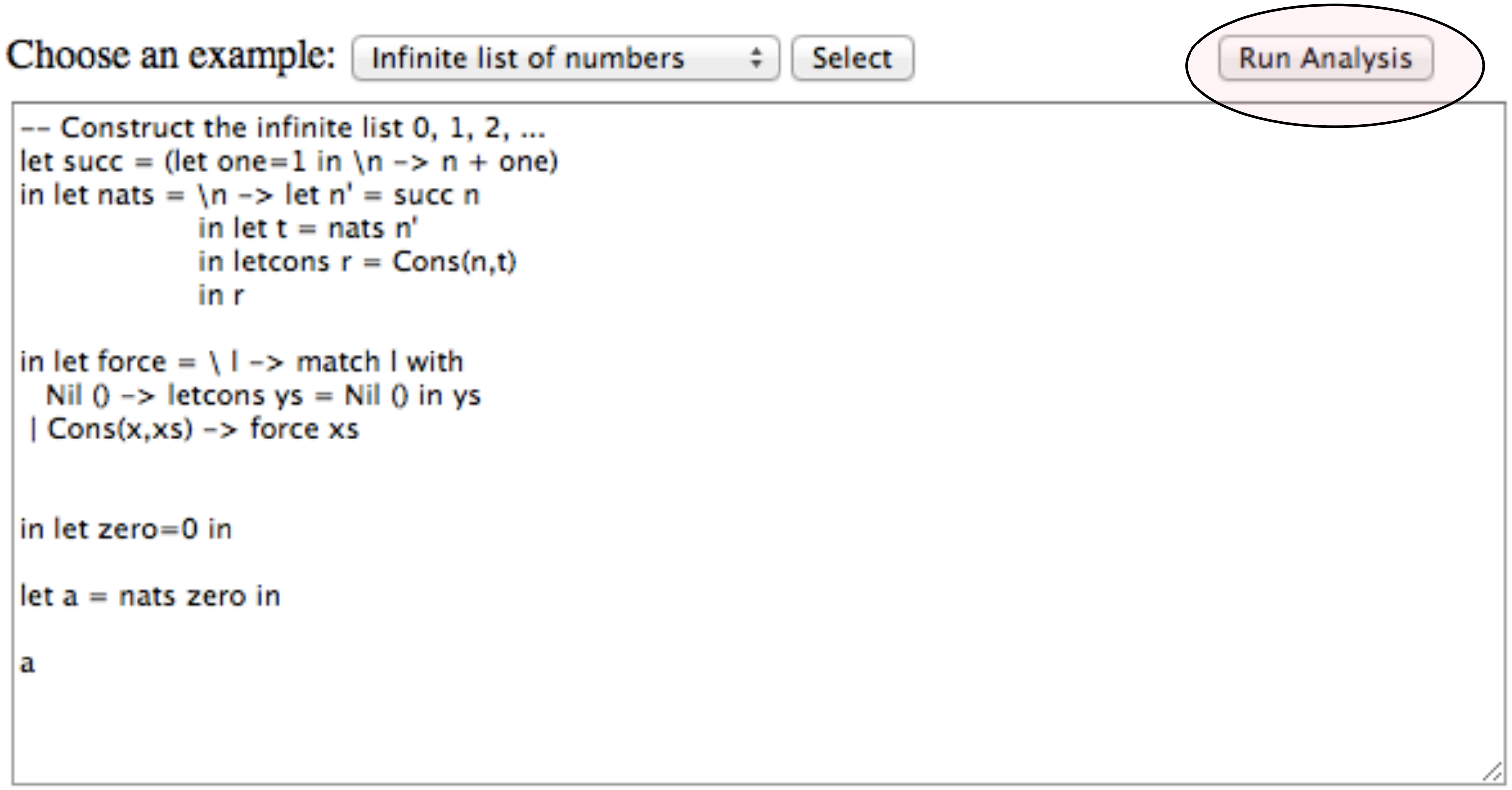
- Lets us pay up-front for all or part of the cost of a *thunk*
  - so we can record possible costs for lazy evaluation and share costs among multiple uses

$$\frac{\Gamma, x{:}\mathrm{T}^{q_0}_{q'}(A) \;\vdash^{p}_{p'}\; e : C}{\Gamma, x{:}\mathrm{T}^{q_0+q_1}_{q'}(A) \;\vdash^{p+q_1}_{p'}\; e : C} \qquad (\textsc{Prepay})$$

Kevin Hammond, University of St Andrews

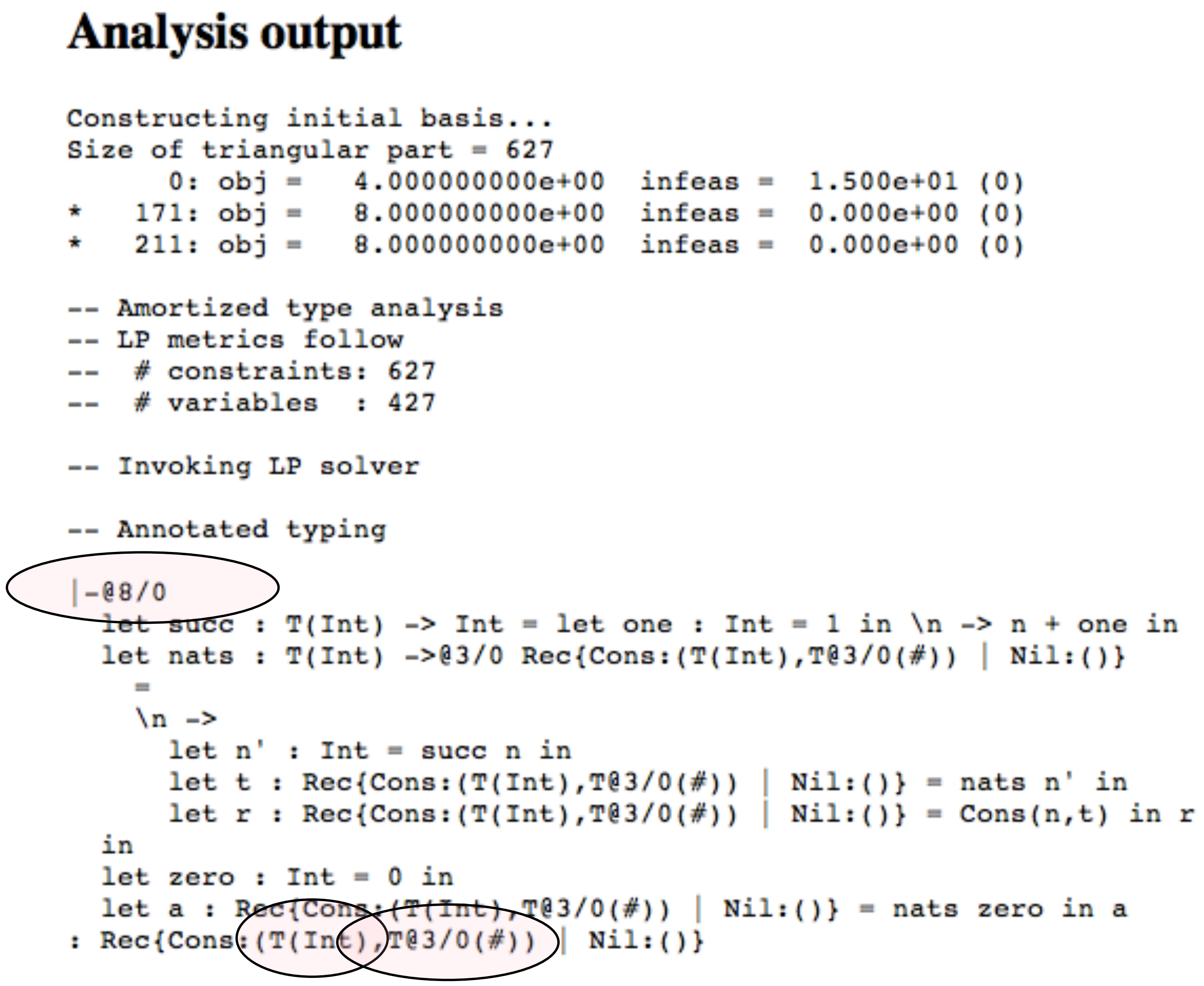
EmBounded

Demonstration



```
-- Construct the infinite list 0, 1, 2, ...
let succ = (let one=1 in \n -> n + one)
in let nats = \n -> let n' = succ n
                 in let t = nats n'
                 in letcons r = Cons(n,t)
                 in r

in let force = \ l -> match l with
   Nil () -> letcons ys = Nil () in ys
 | Cons(x,xs) -> force xs


in let zero=0 in

let a = nats zero in

a
```

Demonstration

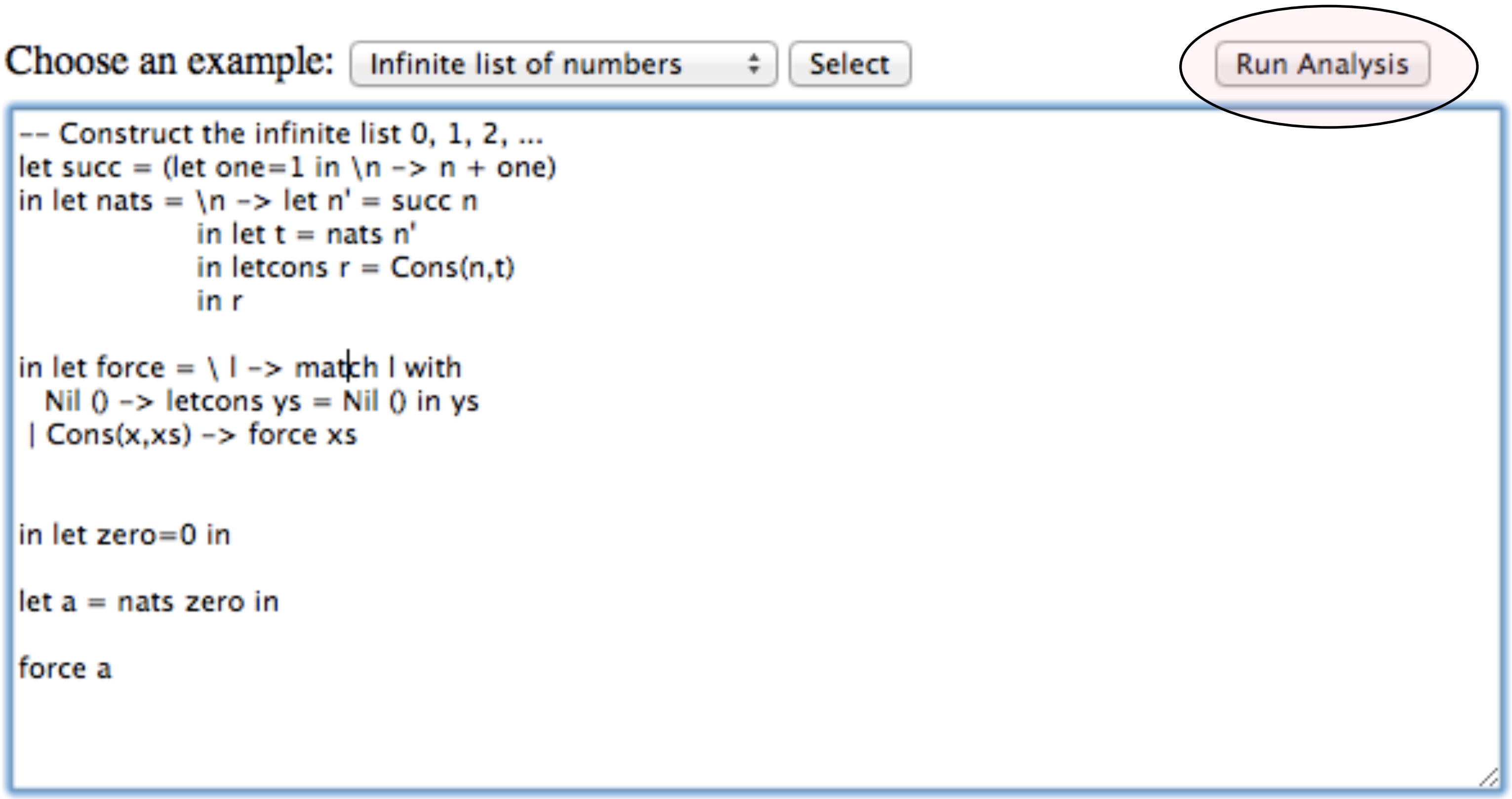


Analysis output

```
Constructing initial basis...
Size of triangular part = 627
      0: obj =   4.000000000e+00  infeas =  1.500e+01 (0)
*   171: obj =   8.000000000e+00  infeas =  0.000e+00 (0)
*   211: obj =   8.000000000e+00  infeas =  0.000e+00 (0)

-- Amortized type analysis
-- LP metrics follow
--  # constraints: 627
--  # variables  : 427

-- Invoking LP solver

-- Annotated typing

|-@8/0
  let succ : T(Int) -> Int = let one : Int = 1 in \n -> n + one in
  let nats : T(Int) ->@3/0 Rec{Cons:(T(Int),T@3/0(#)) | Nil:()}
    =
    \n ->
      let n' : Int = succ n in
      let t : Rec{Cons:(T(Int),T@3/0(#)) | Nil:()} = nats n' in
      let r : Rec{Cons:(T(Int),T@3/0(#)) | Nil:()} = Cons(n,t) in r
  in
  let zero : Int = 0 in
  let a : Rec{Cons:(T(Int),T@3/0(#)) | Nil:()} = nats zero in a
: Rec{Cons:(T(Int),T@3/0(#)) | Nil:()}
```

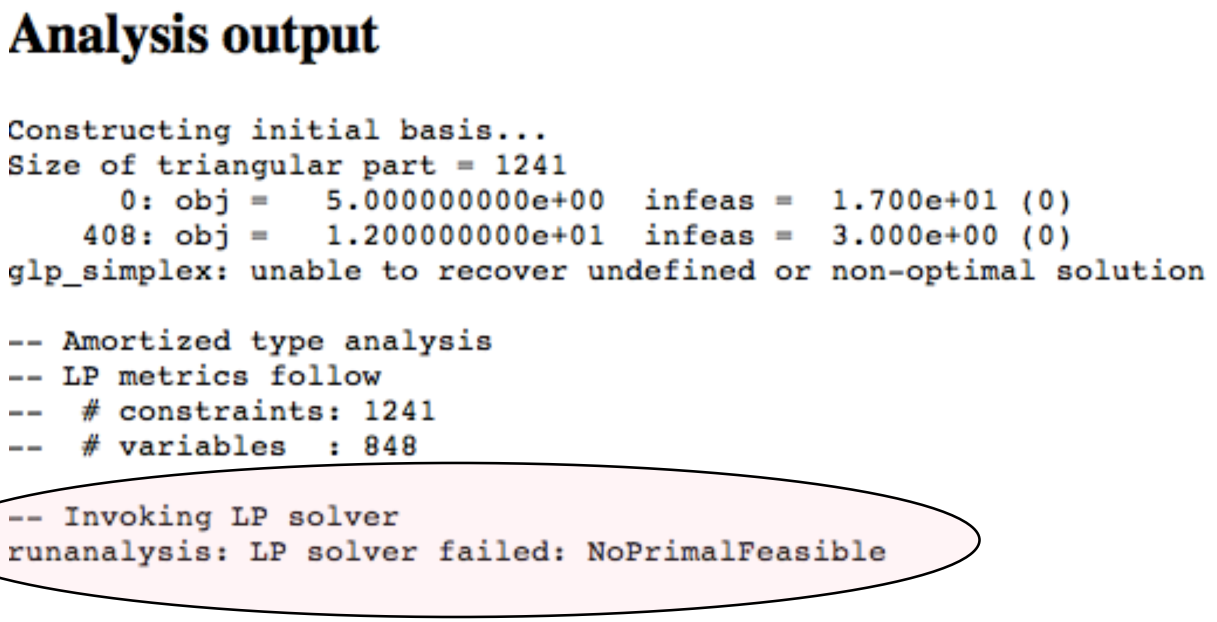
Kevin Hammond, University of St Andrews

Demonstration



```
-- Construct the infinite list 0, 1, 2, ...
let succ = (let one=1 in \n -> n + one)
in let nats = \n -> let n' = succ n
              in let t = nats n'
              in letcons r = Cons(n,t)
              in r

in let force = \ l -> match l with
   Nil () -> letcons ys = Nil () in ys
 | Cons(x,xs) -> force xs


in let zero=0 in

let a = nats zero in

force a
```

Demonstration



Analysis output

```
Constructing initial basis...
Size of triangular part = 1241
      0: obj =   5.000000000e+00  infeas =  1.700e+01 (0)
    408: obj =   1.200000000e+01  infeas =  3.000e+00 (0)
glp_simplex: unable to recover undefined or non-optimal solution

-- Amortized type analysis
-- LP metrics follow
--   # constraints: 1241
--   # variables  : 848

-- Invoking LP solver
runanalysis: LP solver failed: NoPrimalFeasible
```

Conclusions

- IT IS NOW POSSIBLE TO (ACCURATELY) COST LAZILY EVALUATED PROGRAMS
  - Heap Allocations Only

Conclusions

- First automatic static analysis for costing lazy evaluation
  - *Guaranteed worst-case bounds for all possible inputs*
  - NOT simply symbolic execution / profiling
- Full soundness proof against (a variant of) Launchbury's 1993 semantics
- Prototype implementation available
  - http://kashmir.dcc.fc.up.pt/cgi/aalazy.cgi
- Accurate against a range of simple examples:
  - Finite/infinite lists
  - Higher-order functions
  - Functional queues

Ongoing/Future Work

- Deallocation
  - use e.g. an explicit reuse primitive as in Hofmann and Jost (POPL 2003)
- Non-Linear Bounds/Wider range of applications
  - e.g. Hoffmann, Aehrig and Hofmann's approach (POPL 2012)
  - incorporate Campbell's *give-back* annotations for stacks
- Garbage Collection
  - Adapt region-based approach to give countable costs
  - Lifetime/Pointer Safety Analysis
    - » An issue if regions are seen as a programmer level notation
    - » *Not really an issue if the mechanisms are to be handled automatically/for experimental testbed purposes*
- Multicore/Manycore
  - We are looking at new statistical ways to combine worst-case information
  - We are also looking at costs for patterns of parallelism
  - Energy usage is also interesting
  - We need to find the right balance between lazy and eager evaluation
- Extend towards Haskell
  - additional language constructs
  - polymporphic types
  - ...

Some Papers

Automatic Amortized Analysis of Dynamic Memory Allocation for Lazy Functional Programs
Hugo Simoes, Pedro Vasconcelos, Mario Florido, Steffen Jost, and Kevin Hammond
Proc. ACM Conf. on Functional Programming. (ICFP 2012), Copenhagen, Sept. 2012.

Static Determination of Quantitative Resource Usage for Higher-Order Programs
Steffen Jost, Hans-Wolfgang Loidl, Kevin Hammond, Norman Scaife, and Martin Hofmann
Proc. ACM Symp. on Principles of Prog. Langs. (POPL 2010), Madrid, January 2010.

"Carbon Credits" for Resource-Bounded Computations using Amortised Analysis
Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann
Proc. 2009 Conf. on Formal Methods (FM 2009), Eindhoven, The Netherlands, November 2009.

Resource-safe Systems Programming with Embedded Domain Specific Languages,
Edwin Brady and Kevin Hammond,
Fourteenth International Symposium on Practical Aspects of Declarative Languages: PADL 2012. ACM

Inferring Costs for Recursive, Polymorphic and Higher-Order Functional Programs
Pedro Vasconcelos and Kevin Hammond
Proc. 2003 Intl. Workshop on Implementation of Functional Languages (IFL '03), **Edinburgh, Springer-Verlag LNCS, 2004.** ***Winner of the Peter Landin Prize for best paper***

Predictable Space Behaviour in FSM-Hume,
Kevin Hammond and Greg Michaelson,
Proc. 2002 Intl. Workshop on Implementation of Functional Languages (IFL '02), **Madrid, Spain, Sept. 2002, Springer-Verlag LNCS 2670, ISBN 3-540-40190-3, 2003, pp. 1-16**

Funded by

€1.3M grant under EU Framework 6

EmBounded: IST-2004-510255, 2005-2009

€2.5M grant under EU Framework 7

Advance: IST-248828, 2010-2013

£1.25M grants from the UK's EPSRC

MetaHume: EP/C001346/0, 2005-2008

Adaptive Hardware Systems: EP/F020657, 2008-2011

£650K grants from the UK's Ministry of Defence

Sensor Applications for Autonomous Vehicles, SEN002

Dynamic, Cost-directed Reconfiguration of Multi-Asset Systems, SEN015

Travel grants/Fellowships from the *Royal Society of Edinburgh*, *British Council, CNRS* etc.