

Space cost analysis using sized types

Pedro Vasconcelos
School of Computer Science,
University of St Andrews
`pv@dcs.st-and.ac.uk`

15th May 2003

Outline for this talk:

- Motivation: the Hume language
- The sized time type system
- The inference algorithm
- Examples of heap space analysis
- Conclusions and further work

Hume: Higher-order Uniform Meta-Environment



David Hume: Scottish
Enlightenment Philosopher and
Sceptic (1711–1776)

<http://www.hume-lang.org>

Hume Design Objectives

- Targets embedded/critical applications:
 - ★ Hard real-time target
 - ★ Predictable space/time behaviour
 - ★ Asynchronous concurrency model
- High level expressiveness:
 - ★ Automatic memory management
 - ★ Fewer runtime errors: *strong typing*
 - ★ Code re-use: *polymorphism* and *higher-order functions*



The Expression Language \mathcal{L}

- Pure functional language
- No state, concurrency or I/O
- No recursion (for now)
- Higher-order & polymorphic
- Expression layer of Hume (almost)
- Target for our cost analysis

Abstract syntax for \mathcal{L}

$x \in \mathbf{Var}$

$c \in \mathbf{Const} = \{0, 1, 2, \dots, \text{true}, \text{false} \dots\}$

$e ::= x \mid c$

$\mid \text{fn } x \rightarrow e$

$\mid e_1 e_2$

$\mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3$

$\mid \text{let } x = e_1 \text{ in } e_2 .$

What are we going to measure?

- Number of heap allocated cells
- No garbage collection (maximum usage)
- Simple model: 1 cell per list cons
- Can easily be extended
- Alternative metrics:
 - ★ Number of reduction steps
 - ★ Size of the execution stack frame

The Sized-Time Type System

- An instance of *type and effect* analysis
- Annotate base types with *size*
- Annotate function type with *latent cost*
- Infer sizes and costs together with *type inference*

Sized Types

$$\tau ::= \alpha \mid \text{Bool} \mid \text{Float} \mid \text{Nat}^z \mid \text{List}^z \tau \mid \tau_1 \xrightarrow{z} \tau_2 .$$

Naturals: z is the *magnitude* of the natural

Lists: z is the *length* of the list

Functions: z is the *latent cost* of the function

Size/Latent Cost Expressions

$$z ::= \ell \mid n \mid z_1 + z_2 \mid z_1 - z_2 \mid z_1 \times z_2 \\ \mid \max(z_1, z_2) \mid \min(z_1, z_2) \mid \omega .$$

ℓ is a size/cost variable

$n \in \{0, 1, 2 \dots\}$: finite upper bound on size/cost

ω : potentially unbounded size/cost

Examples of Sized Types

5	Nat^5
[2, 3, 7, 1, 5]	$\text{List}^5 \text{Nat}^7$
$\text{fn } n \rightarrow n + 1$	$\text{Nat}^n \xrightarrow{0} \text{Nat}^{n+1}$
$\text{fn } n \rightarrow 2^n$	$\text{Nat}^n \xrightarrow{0} \text{Nat}^\omega$
cons	$\alpha \xrightarrow{0} \text{List}^k \alpha \xrightarrow{1} \text{List}^{k+1} \alpha$

Relaxing sizes and costs

We want to be able to relax size/cost information:

$$\text{Nat}^5 \subseteq \text{Nat}^7 \subseteq \text{Nat}^\omega$$

- Formally: a *subtyping* relation \trianglelefteq
- Defined structurally for arbitrary types
- Contravariant on function arguments:

$$\text{Nat}^7 \xrightarrow{\approx} \text{Nat}^{10} \trianglelefteq \text{Nat}^5 \xrightarrow{\approx} \text{Nat}^\omega$$

Cost model assumptions

$$\text{cons} : \forall. \alpha \xrightarrow{0} \text{List}^n \alpha \xrightarrow{1} \text{List}^{n+1} \alpha$$

$$\text{nil} : \forall. \text{List}^0 \alpha$$

$$\text{map} : \forall. (\alpha \xrightarrow{k} \beta) \xrightarrow{0} \text{List}^n \alpha \xrightarrow{n(1+k)} \text{List}^n \beta$$

$$\text{foldl} : \forall. (\alpha \xrightarrow{k_1} \beta \xrightarrow{k_2} \alpha) \xrightarrow{0} \alpha \xrightarrow{0} \text{List}^n \beta \xrightarrow{n(k_1+k_2)} \alpha$$

$$\text{iterate} : \forall. \text{Nat}^n \xrightarrow{0} (\alpha \xrightarrow{k} \alpha) \xrightarrow{0} \alpha \xrightarrow{n(1+k)} \text{List}^n \alpha$$

All other functions: zero heap costs.

The Sized-Time Type System

- Judgements $\Gamma \vdash e : \tau \ \& \ z$
 - ★ Γ is a set of assumptions
 - ★ e is an \mathcal{L} -expression
 - ★ τ is a sized type for e
 - ★ z is the heap cost for e
- Assumptions map identifiers to types schemes
- Universally quantified over type or size variables

The Sized Time System: `var` & `nat`

$$\frac{x \in \text{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x) \ \& \ 0} \quad [\text{var}_{st}]$$

$$\frac{}{\Gamma \vdash n : \text{Nat}^n \ \& \ 0} \quad [\text{nat}_{st}]$$

The Sized Time System: abs & app

$$\frac{\Gamma[x : \tau_1] \vdash e : \tau_2 \ \& \ z}{\Gamma \vdash \text{fn } x \rightarrow e : \tau_1 \xrightarrow{z} \tau_2 \ \& \ 0} [\text{abs}_{st}]$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \xrightarrow{z} \tau_2 \ \& \ z_1 \quad \Gamma \vdash e_2 : \tau_1 \ \& \ z_2}{\Gamma \vdash e_1 e_2 : \tau_2 \ \& \ z + z_1 + z_2} [\text{app}_{st}]$$

The Sized Time System: if & weak

$$\frac{\Gamma \vdash e_0 : \text{Bool} \ \& \ z_0 \quad \Gamma \vdash e_1 : \tau \ \& \ z \quad \Gamma \vdash e_2 : \tau \ \& \ z}{\Gamma \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : \tau \ \& \ z_0 + z} [\text{if}_{st}]$$

$$\frac{\Gamma \vdash e : \tau \ \& \ z \quad \tau \sqsubseteq \tau' \quad z \leq z'}{\Gamma \vdash e : \tau' \ \& \ z'} [\text{weak}_{st}]$$

The Sized Time System: polymorphism

$$\frac{\Gamma \vdash e_1 : \sigma_1 \ \& \ z_1 \quad \Gamma[x : \sigma_1] \vdash e_2 : \tau_2 \ \& \ z_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2 \ \& \ z_1 + z_2} \text{[let}_{st}\text{]}$$

$$\frac{\Gamma \vdash e : \forall \vec{\gamma}_i. \tau \ \& \ z \quad \text{dom}(\theta) \cup \text{dom}(\vartheta) \subseteq \{\vec{\gamma}_i\}}{\Gamma \vdash e : \vartheta(\theta\tau) \ \& \ z} \text{[ins}_{st}\text{]}$$

$$\frac{\Gamma \vdash e : \tau \ \& \ z \quad \{\vec{\gamma}_i\} \cap \text{FV}(\Gamma) = \emptyset}{\Gamma \vdash e : \forall \vec{\gamma}_i. \tau \ \& \ z} \text{[gen}_{st}\text{]}$$

Inference Algorithm: Overview

- Reconstruct types and costs for \mathcal{L} -expressions
- Extension of the Damas-Milner inference algorithm
- Type structure handled separately from costs:
 - ★ Size/cost annotations restricted to single variables
 - ★ *Constraints* on size/cost variables
 - ★ Inference algorithm outputs annotated type plus a *constraint set*
 - ★ Solving the constraint set yields sizes and costs

Inference Algorithm: Constraints

List cons type “ $\alpha \xrightarrow{0} \text{List}^k \alpha \xrightarrow{1} \text{List}^{k+1} \alpha$ ” becomes

$$(\alpha \xrightarrow{k_1} \text{List}^k \alpha \xrightarrow{k_2} \text{List}^{k'} \alpha, \{k_1 \geq 0, k_2 \geq 1, k' \geq k + 1\}).$$

- Annotations in the type are always variables
- Constraints restricted to “ $var \geq \dots$ ”
- \forall -quantification allows polymorphism

Inference Algorithm: Details

- Accepts all non-recursive Hindley-Milner typable programs
- Constraint sets always have a solution
- Efficient algorithm for solving constraints (variant of flow analysis)
- Outputs symbolic size & cost expressions
- Prototype implementation in Haskell

Example 1: Conditional list insert

$\text{insert } x \ ys = \text{if elem } x \ ys \text{ then } ys \text{ else cons } x \ ys$

$\Rightarrow \text{insert} : \alpha \xrightarrow{0} \text{List}^n \alpha \xrightarrow{1} \text{List}^{n+1} \alpha$

- Worst-case:
 - ★ List grows by one
 - ★ One cell allocated
- Polymorphic conclusion: valid for any α

Example 2: List append

$$\begin{aligned}
 xs ++ ys &= [x_1, x_2, \dots, x_n, y_1, y_2, \dots, y_m] \\
 &= \underbrace{x_1 : (x_2 : (\dots : (x_n : ys) \dots))}_{n \text{ cons}}
 \end{aligned}$$

Using a higher-order function:

$$\text{append } xs \ ys = \text{foldr cons } ys \ xs$$

$$\Rightarrow \text{append} : \text{List}^n \alpha \xrightarrow{0} \text{List}^m \alpha \xrightarrow{n} \text{List}^\omega \alpha$$

- Accurate cost of n cells
- Result list size: ω because of **size-aliasing!**

Example 3: Numerical summation

Given a function $f : [0, 1] \rightarrow \mathbb{R}$ and n , compute

$$\sum_{i=0}^n f(i/n) = f(0) + f\left(\frac{1}{n}\right) + \cdots + f\left(\frac{n-1}{n}\right) + f(1)$$

- Generate a list with the $n + 1$ points;
- Map the function over the list of points;
- Sum the resulting list

Example 3: Numerical summation

```
fsum  $f$   $n$  =
  let  $h = 1.0 / (\text{float } n)$  ;
       $xs = \text{iterate } (\text{succ } n) (\text{fn } x \rightarrow x + h) 0.0$  ;
      sum = foldl (+) 0.0
  in
      sum (map  $f$   $xs$ )
```

\Rightarrow $\text{fsum} : (\text{Float} \xrightarrow{k} \text{Float}) \xrightarrow{0} \text{Nat}^n \xrightarrow{(n+1)(k+2)} \text{Float}$

Example 3: Numerical summation

- If f has zero cost: $k=0 \Rightarrow (n+1)(k+2) = 2(n+1)$,

$$\begin{array}{r}
 n+1 \quad \text{from iterate} \\
 + \quad n+1 \quad \text{from map} \\
 \hline
 2(n+1) \quad \text{total.}
 \end{array}$$

- In general: f latent cost added for each list element
- Higher-order: inferred type depends on latent cost of f
- Re-usable analysis: partial application, instantiation

Conclusions

- Automatic cost analysis combining *higher-order* and *polymorphic* functions
- Allows modular analysis (separate compilation)
- Extension to the standard type inference algorithm
- Semantic correctness proof (for timing costs, w.r.t. a step-counting operational semantics)
- Results from prototype implementation are encouraging

Issues in applying this analysis

- Refine the cost model (adding costs for naturals, floats, booleans and closures)
- Perform the analysis *after* source-to-source optimisations
- For timing costs: make sure the back-end doesn't invalidate the cost model (peephole optimisations, instruction re-ordering, cache issues. . .)
- Easier to assure by design: Hume

Further Work

- Extend the analysis to:
 - ★ Primitive recursion (*under way*)
 - ★ Tuples, vectors, user defined recursive data-types
 - ★ More general forms of recursion?
- Proof the soundness of the algorithm (& completeness?)
- Avoid unnecessary widening of solutions (*size-aliasing*)
- Measure the quality of the solutions against the Hume implementation