# Amortized Heap-Space Analysis for First-Order Functional Programs

O. Shkaravska

Inst. of Cybernetics
at Tallinn Univ. of Technology

Kalvi, 2005

# Outline

1. Motivation
   - Amortization-based Evaluation of Heap Consumption
   - Previous Work
2. Results
   - Heap-aware Type System for Programs over Lists
   - Soundness Theorem

Some problems are reported "on-line"...

# Outline

1. Motivation
   - Amortization-based Evaluation of Heap Consumption
   - Previous Work
2. Results
   - Heap-aware Type System for Programs over Lists
   - Soundness Theorem

Some problems are reported “on-line”...

# Outline

1. Motivation
   - Amortization-based Evaluation of Heap Consumption
   - Previous Work
2. Results
   - Heap-aware Type System for Programs over Lists
   - Soundness Theorem

Some problems are reported “on-line”...

# Practical Aspect

Heap-space deficit in run-time leads to crash.

- Small devices: smartcards, mobile phones, ...
- a few programs are expected to be run on one machine,

Solution: evaluate heap consumption before running programs.

# What is Amortization

- Given: a sequence of operations.
- Find: the cost of the entire sequence.
- Remark:
  - The *actual cost* $t_i$, not that important!
  - The *amortized cost* $a_i$, s.t. $\Sigma_{i=1}^{j} a_i \geq \Sigma_{i=1}^{j} t_i$.

**Banker's View**

If $c_i := a_i - t_i > 0$, it is called a *credit*.

**Physicist's View**

Data: $D_0, \ldots, D_i, \ldots$

A Potential Function $\Phi : D_i \mapsto \Upsilon_i \geq 0$.

$c_i = \Upsilon_i - \Upsilon_{i-1}$

# What is Amortization

- Given: a sequence of operations.
- Find: the cost of the entire sequence.
- Remark:
  - The *actual cost* $t_i$, not that important!
  - The *amortized cost* $a_i$, s.t. $\Sigma_{i=1}^{j} a_i \geq \Sigma_{i=1}^{j} t_i$.

**Banker's View**

If $c_i := a_i - t_i > 0$, it is called a *credit*.

**Physicist's View**

Data: $D_0, \ldots, D_i, \ldots$
A Potential Function $\Phi : D_i \mapsto \Upsilon_i \geq 0$.

$c_i = \Upsilon_i - \Upsilon_{i-1}$

# What is Amortization

- Given: a sequence of operations.
- Find: the cost of the entire sequence.
- Remark:
  - The *actual cost* $t_i$, not that important!
  - The *amortized cost* $a_i$, s.t. $\Sigma_{i=1}^{j} a_i \geq \Sigma_{i=1}^{j} t_i$.

**Banker's View**

If $c_i := a_i - t_i > 0$, it is called a *credit*.

**Physicist's View**

Data: $D_0, \ldots, D_i, \ldots$
A Potential Function $\Phi : D_i \mapsto \Upsilon_i \geq 0$.

$c_i = \Upsilon_i - \Upsilon_{i-1}$

# What is Amortization

- Given: a sequence of operations.
- Find: the cost of the entire sequence.
- Remark:
  - The *actual cost* $t_i$, not that important!
  - The *amortized cost* $a_i$, s.t. $\Sigma_{i=1}^{j} a_i \geq \Sigma_{i=1}^{j} t_i$.

## Banker's View

If $c_i := a_i - t_i > 0$, it is called a *credit*.

## Physicist's View

Data: $D_0, \ldots, D_i, \ldots$
A Potential Function $\Phi : D_i \mapsto \Upsilon_i \geq 0$.

$c_i = \Upsilon_i - \Upsilon_{i-1}$

# What is Amortization

- Given: a sequence of operations.
- Find: the cost of the entire sequence.
- Remark:
  - The *actual cost* $t_i$, not that important!
  - The *amortized cost* $a_i$, s.t. $\Sigma_{i=1}^{j} a_i \geq \Sigma_{i=1}^{j} t_i$.

**Banker's View**

If $c_i := a_i - t_i > 0$, it is called a *credit*.

**Physicist's View**

Data: $D_0, \ldots, D_i, \ldots$
A Potential Function $\Phi : D_i \mapsto \Upsilon_i \geq 0$.

$c_i = \Upsilon_i - \Upsilon_{i-1}$

# Amortization : fine computable(!) resource bounds, resource information in types

$$\texttt{f x} = \texttt{match}\,\texttt{x}\,\texttt{with} \quad \begin{array}{l} \texttt{Nil} \Rightarrow \texttt{cons}(1, \texttt{Nil}) \\ \mid \texttt{cons}(\texttt{h}, \texttt{t}) \Rightarrow \texttt{cons}(1, \texttt{cons}(2, \texttt{Nil})) \end{array}$$

The bound is: $T(length) = \begin{cases} 1, & length = 0 \\ 2, & length \geq 1, \end{cases}$

Typing: $\mathrm{L}(Int, k), 1 \rightarrow \mathrm{L}(Int, 0), 0$

We assign:

- 1 extra heap unit before the computation,
- An extra heap unit to the first element: $k(1) = 1$,
- Other elements do not need extras: $k(i) = 0$, $i \geq 2$.

# Amortization : fine computable(!) resource bounds, resource information in types

$$\mathtt{f\ x} = \mathtt{match\,x\,with}\ \ \mathtt{Nil} \Rightarrow \mathtt{cons}(1, \mathtt{Nil})$$
$$| \mathtt{cons}(\mathtt{h}, \mathtt{t}) \Rightarrow \mathtt{cons}(1, \mathtt{cons}(2, \mathtt{Nil}))$$

The bound is: $T(length) = \begin{cases} 1, & length = 0 \\ 2, & length \geq 1, \end{cases}$

Typing: $\mathrm{L}(Int, k),\ 1\ \rightarrow\ \mathrm{L}(Int, 0),\ 0$

We assign:

- 1 extra heap unit before the computation,
- An extra heap unit to the first element: $k(1) = 1$,
- Other elements do not need extras: $k(i) = 0$, $i \geq 2$.

## Amortization : fine computable(!) resource bounds, resource information in types

$$\texttt{f x} = \ \texttt{match x with} \ \ \texttt{Nil} \Rightarrow \texttt{cons}(1, \texttt{Nil}) \\ \qquad\qquad\qquad\qquad\qquad\qquad | \, \texttt{cons}(\texttt{h}, \texttt{t}) \Rightarrow \texttt{cons}(1, \texttt{cons}(2, \texttt{Nil}))$$

The bound is: $T(\mathit{length}) = \begin{cases} 1, & \mathit{length} = 0 \\ 2, & \mathit{length} \geq 1, \end{cases}$

Typing: $\mathrm{L}(\mathit{Int}, k),\ 1 \ \rightarrow \ \mathrm{L}(\mathit{Int}, 0),\ 0$

We assign:

- 1 extra heap unit before the computation,
- An extra heap unit to the first element: $k(1) = 1$,
- Other elements do not need extras: $k(i) = 0$, $i \geq 2$.

# Amortization (mainly for Time) - Reading in Progress

- Basic:
  - Cormen, Leiserson, Rivest - “Introduction to algorithms”
  - Okasaki - “Purely Functional Data Structures “

  fine treatment of recursive calls
  (binary increment in logarithm)
  Okasaki: lazy-eval. with suspesnions
- Schoenmakers - PhD thesis “Data Structures and Amortized Complexity in a Functional Setting”:
  - algebraic approach
  - linear usage
  - fi ne treatment of compositions/recursive calls
  - time

# Amortization (mainly for Time) - Reading in Progress

- Basic:
  - Cormen, Leiserson, Rivest - “Introduction to algorithms”
  - Okasaki - “Purely Functional Data Structures “

  fine treatment of recursive calls
  (binary increment in logarithm)
  Okasaki: lazy-eval. with suspesnions
- Schoenmakers - PhD thesis “Data Structures and Amortized Complexity in a Functional Setting”:
  - algebraic approach
  - linear usage
  - fi ne treatment of compositions/recursive calls
  - time

# Amortization (mainly for Time) - Reading in Progress

- Basic:
  - Cormen, Leiserson, Rivest - "Introduction to algorithms"
  - Okasaki - "Purely Functional Data Structures "

  fine treatment of recursive calls
  (binary increment in logarithm)
  Okasaki: lazy-eval. with suspesnions
- Schoenmakers - PhD thesis "Data Structures and Amortized Complexity in a Functional Setting":
  - algebraic approach
  - linear usage
  - fi ne treatment of compositions/recursive calls
  - time

# Amortization (mainly for Time) - Reading in Progress

- Basic:
  - Cormen, Leiserson, Rivest - “Introduction to algorithms”
  - Okasaki - “Purely Functional Data Structures “

  fine treatment of recursive calls
  (binary increment in logarithm)
  Okasaki: lazy-eval. with suspesnions
- Schoenmakers - PhD thesis “Data Structures and Amortized Complexity in a Functional Setting”:
  - algebraic approach
  - linear usage
  - fi ne treatment of compositions/recursive calls
  - time

# Problem: Fine Treatment of Recursive Calls

I can not type-check the increment-for-logarithm example in the presented type system!

The solution exists, but it leads to singleton types.

May be there are other solutions: later ...

# Hofmann-Jost System for Linear Heap Bounds

## Example

The program "copy"

```
copy x =   match x with
           Nil ⇒ Nil
           | cons(h, t) ⇒  let  y=copy t
                                in cons(h, y)
```

has typing: $\mathrm{L}(\mathit{Int}, 1),\ 0\ \rightarrow\ \mathrm{L}(\mathit{Int}, 0),\ 0$:
assign to each element of an input list - 1 extra heap unit.

## Semantics

Typing $\mathrm{L}(\mathit{Int}, k),\ k_0\ \rightarrow\ \mathrm{L}(\mathit{Int}, k'),\ k'_0$ means

- heap consumption $k\,l + k_0$,
- gain $k'\,l' + k'_0$

# Hofmann-Jost System for Linear Heap Bounds

## Example

The program “copy”

```
copy x =  match x with
          Nil ⇒ Nil
          | cons(h, t) ⇒  let  y=copy t
                               in cons(h, y)
```

has typing: $\mathrm{L}(\mathit{Int}, 1),\ 0 \ \rightarrow\ \mathrm{L}(\mathit{Int}, 0),\ 0$:
assign to each element of an input list - 1 extra heap unit.

## Semantics

Typing $\mathrm{L}(\mathit{Int}, k),\ k_0 \ \rightarrow\ \mathrm{L}(\mathit{Int}, k'),\ k'_0$ means

- heap consumption $k\, l + k_0$,
- gain $k'\, l' + k'_0$

# Hofmann-Jost System for Linear Heap Bounds

## Example

The program "copy"

```
copy x =  match x with
          Nil ⇒ Nil
          | cons(h, t) ⇒  let  y=copy t
                               in cons(h, y)
```

has typing: $\mathrm{L}(\mathit{Int}, 1),\ 0\ \to\ \mathrm{L}(\mathit{Int}, 0),\ 0$:
assign to each element of an input list - 1 extra heap unit.

## Semantics

Typing $\mathrm{L}(\mathit{Int}, k),\ k_0\ \to\ \mathrm{L}(\mathit{Int}, k'),\ k'_0$ means

- heap consumption $k\,l + k_0$,
- gain $k'\,l' + k'_0$

# What Amortization Brings to Types

Credits are type annotaions carrying resource information.

## Zero-Order, Sized and Unsized, Annotated Types

$T = Int \mid \mathrm{L}_l(T, k) \mid \mathrm{L}(T, k)$
$k : \mathbb{N} \to \mathbb{R}^+$
$k(i)$ is the *credit* of the $i$th $\mathtt{cons}$-cell.
$\sum_{i=1}^{l} k(i)$ is the *potential* of a list of integers
$k$ is a constant in HJ system.

## The hint for Type-checking

Unary Functions over Lists.
Let $F$ has a bounded on $[\alpha, \infty]$ derivaive, with $0 \leq \alpha < 1$.
Perform type-checking for input with $k(x) = F'(x)$.
Total consumption is $\sum_{i=1}^{l} k(i) \approx \int_{i=\alpha}^{l} k(x)\, d\, x = F(x) - F(\alpha)$

# What Amortization Brings to Types

Credits are type annotaions carrying resource information.

## Zero-Order, Sized and Unsized, Annotated Types

$T = Int \mid \mathrm{L}_l(T, k) \mid \mathrm{L}(T, k)$

$k : \mathbb{N} \to \mathbb{R}^+$

$k(i)$ is the *credit* of the $i$th cons-cell.

$\sum_{i=1}^{l} k(i)$ is the *potential* of a list of integers

$k$ is a constant in HJ system.

## The hint for Type-checking

Unary Functions over Lists.

Let $F$ has a bounded on $[\alpha, \infty]$ derivaive, with $0 \leq \alpha < 1$.

Perform type-checking for input with $k(x) = F'(x)$.

Total consumption is $\sum_{i=1}^{l} k(i) \approx \int_{i=\alpha}^{l} k(x)\, d\, x = F(x) - F(\alpha)$

# What Amortization Brings to Types

Credits are type annotaions carrying resource information.

## Zero-Order, Sized and Unsized, Annotated Types

$T = Int \mid \mathrm{L}_l(T, k) \mid \mathrm{L}(T, k)$
$k : \mathbb{N} \to \mathbb{R}^+$
$k(i)$ is the *credit* of the $i$th cons-cell.
$\sum_{i=1}^{l} k(i)$ is the *potential* of a list of integers
$k$ is a constant in HJ system.

## The hint for Type-checking

Unary Functions over Lists.
Let $F$ has a bounded on $[\alpha, \infty]$ derivaive, with $0 \leq \alpha < 1$.
Perform type-checking for input with $k(x) = F'(x)$.
Total consumption is $\sum_{i=1}^{l} k(i) \approx \int_{i=\alpha}^{l} k(x)\, d\, x = F(x) - F(\alpha)$

# What Amortization Brings to Types

Credits are type annotaions carrying resource information.

## Zero-Order, Sized and Unsized, Annotated Types

$T = Int \mid \mathrm{L}_l(T, k) \mid \mathrm{L}(T, k)$
$k : \mathbb{N} \to \mathbb{R}^+$
$k(i)$ is the *credit* of the $i$th cons-cell.
$\sum_{i=1}^{l} k(i)$ is the *potential* of a list of integers
$k$ is a constant in HJ system.

## The hint for Type-checking

Unary Functions over Lists.
Let $F$ has a bounded on $[\alpha, \infty]$ derivaive, with $0 \leq \alpha < 1$.
Perform type-checking for input with $k(x) = F'(x)$.
Total consumption is $\sum_{i=1}^{l} k(i) \approx \int_{i=\alpha}^{l} k(x)\, d\, x = F(x) - F(\alpha)$

# What Amortization Brings to Types

Credits are type annotaions carrying resource information.

## Zero-Order, Sized and Unsized, Annotated Types

$T = Int \mid \mathrm{L}_l(T, k) \mid \mathrm{L}(T, k)$
$k : \mathbb{N} \to \mathbb{R}^+$
$k(i)$ is the *credit* of the $i$th cons-cell.
$\sum_{i=1}^{l} k(i)$ is the *potential* of a list of integers
$k$ is a constant in HJ system.

## The hint for Type-checking

Unary Functions over Lists.
Let $F$ has a bounded on $[\alpha, \infty]$ derivaive, with $0 \leq \alpha < 1$.
Perform type-checking for input with $k(x) = F'(x)$.
Total consumption is $\sum_{i=1}^{l} k(i) \approx \int_{i=\alpha}^{l} k(x)\, d\, x = F(x) - F(\alpha)$

# Typing Judgement

## Judgement

$$\Gamma, n \vdash \left[\frac{e : T}{\Delta}\right], n'$$

$\Gamma$, $\Delta$ – annotated contexts,
$T$ – an annotated type, $n$, $n'$ – nonnegative numbers

## Example – destructive length

```
length' x =
 match x with  Nil ⇒ 0
              |cons(h, t)@_ ⇒  let  y=length' t
                                     in 1 + y
```

$$x : \mathrm{L}_l(Int, 0), 0 \vdash \left[\frac{\texttt{length}'\ \texttt{x} : Int}{x : \mathrm{L}_l(Int, 1)}\right], 0$$

# Typing Judgement

## Judgement

$$\Gamma, n \vdash \left[\frac{e : T}{\Delta}\right], n'$$

$\Gamma$, $\Delta$ – annotated contexts,
$T$ – an annotated type, $n$, $n'$ – nonnegative numbers

## Example – destructive length

$$\begin{aligned}&\texttt{length}'\ \texttt{x} = \\ &\ \texttt{match x with}\ \ \texttt{Nil} \Rightarrow 0 \\ &\qquad\qquad\qquad\ \mid \texttt{cons(h, t)@\_} \Rightarrow\ \texttt{let}\ \ \texttt{y}{=}\texttt{length}'\ \texttt{t} \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \texttt{in}\ 1 + \texttt{y}\end{aligned}$$

$$x : \mathrm{L}_l(Int, 0), 0 \vdash \left[\frac{\texttt{length}'\ \texttt{x} : Int}{x : \mathrm{L}_l(Int, 1)}\right], 0$$

# Typing Judgement

## Judgement

$$\Gamma, n \vdash \left[\frac{e : T}{\Delta}\right], n'$$

$\Gamma$, $\Delta$ – annotated contexts,
$T$ – an annotated type, $n$, $n'$ – nonnegative numbers

## Example – destructive length

$$\begin{array}{l}\mathtt{length'}\ \mathtt{x} = \\ \ \mathtt{match}\,\mathtt{x}\,\mathtt{with}\ \ \mathtt{Nil} \Rightarrow 0 \\ \qquad\qquad\qquad\ \ |\,\mathtt{cons(h,\ t)@\_} \Rightarrow\ \mathtt{let}\ \ \mathtt{y{=}length'\ t} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \mathtt{in}\,1 + \mathtt{y}\end{array}$$

$$x : \mathrm{L}_l(\mathit{Int}, 0), 0 \vdash \left[\frac{\mathtt{length'}\ \mathtt{x} : \mathit{Int}}{x : \mathrm{L}_l(\mathit{Int}, 1)}\right], 0$$

# Some Rules: Constructor

$$\frac{}{h : T,\ t : \mathrm{L}_l(T, k), k(l+1)+1 \vdash \left[\frac{\mathtt{cons(h,\ t)} : \mathrm{L}_{l+1}(T, k)}{h : Z(T),\ t : Z(\mathrm{L}_l(T, k))}\right], 0}$$

where zero-annotation map is efined incductively:

$Z(Int) := Int$,
$Z(\mathrm{L}_l(T, k)) := \mathrm{L}_l(Z(T), 0)$,
$[Z(\Gamma)](x) := Z(\Gamma(x))$.

# First-Order Types and Function Call

$$\frac{\mathrm{L}_l(T,\, k/k''),\, k_0 \;\to\; \mathrm{L}_{l'}(T',\, k'),\, k_0' \;\|\; \Psi(l,\, l',\, k,\, k'',\, k_0,\, k',\, k_0') \qquad \Psi(l,\, l',\, k,\, k'',\, k_0,\, k',\, k_0')}{x : \mathrm{L}_l(T,\, k),\, k_0 \vdash \left[\dfrac{f(x) : \mathrm{L}_{l'}(T',\, k')}{x : \mathrm{L}_l(T,\, k'')}\right],\, k_0'}$$

$l$ is the length of input,
$l'$ is the length of output

The predicate $\Psi$ manages mutual and recursive calls.
For type-checking may have, say, the form $l' = p(l)$.

HJ system: no need, because annotations are constants,
no dependency on the position of an element.

# First-Order Types and Function Call

$$\frac{\mathrm{L}_l(T,\, k/k''),\ k_0 \to \mathrm{L}_{l'}(T',\, k'),\ k'_0 \ \|\ \Psi(l,\, l',\, k,\, k'',\, k_0,\, k',\, k'_0) \qquad \Psi(l,\, l',\, k,\, k'',\, k_0,\, k',\, k'_0)}{x : \mathrm{L}_l(T,\, k),\ k_0 \vdash \left[\dfrac{f(x) : \mathrm{L}_{l'}(T',\, k')}{x : \mathrm{L}_l(T,\, k'')}\right],\ k'_0}$$

$l$ is the length of input,
$l'$ is the length of output

The predicate $\Psi$ manages mutual and recursive calls.
For type-checking may have, say, the form $l' = p(l)$.

HJ system: no need, because annotations are constants,
no dependency on the position of an element.

# First-Order Types and Function Call

$$\frac{\mathrm{L}_l(T,\, k/k''),\; k_0 \;\rightarrow\; \mathrm{L}_{l'}(T',\, k'),\; k'_0 \;\mid\mid\; \Psi(l,\, l',\, k,\, k'',\, k_0,\, k',\, k'_0) \qquad \Psi(l,\, l',\, k,\, k'',\, k_0,\, k',\, k'_0)}{x : \mathrm{L}_l(T,\, k),\, k_0 \vdash \left[\dfrac{f(x) : \mathrm{L}_{l'}(T',\, k')}{x : \mathrm{L}_l(T,\, k'')}\right],\; k'_0}$$

$l$ is the length of input,
$l'$ is the length of output

The predicate $\Psi$ manages mutual and recursive calls.
For type-checking may have, say, the form $l' = p(l)$.

HJ system: no need, because annotations are constants,
no dependency on the position of an element.

# First-Order Types and Function Call

$$\frac{\begin{array}{c}\mathrm{L}_l(T,\ k/k''),\ k_0\ \rightarrow\ \mathrm{L}_{l'}(T',\ k'),\ k_0'\ \ ||\ \Psi(l,\ l',\ k,\ k'',\ k_0,\ k',\ k_0')\\ \Psi(l,\ l',\ k,\ k'',\ k_0,\ k',\ k_0')\end{array}}{x\ :\ \mathrm{L}_l(T,\ k),\ k_0\ \vdash\ \left[\dfrac{f(x)\ :\ \mathrm{L}_{l'}(T',\ k')}{x\ :\ \mathrm{L}_l(T,\ k'')}\right],\ k_0'}$$

$l$ is the length of input,
$l'$ is the length of output

The predicate $\Psi$ manages mutual and recursive calls.
For type-checking may have, say, the form $l' = p(l)$.

HJ system: no need, because annotations are constants, no dependency on the position of an element.

# $\Psi$ is complex to infer

We want to use the type system for

"parametric type-checking"

.

E.g. : I expect that my program

- has something like quadratic heap consumption,
  the task: to obtain $?a\ x^2 + ?b\ x + ?c$ for heap,
- and has the length of the output is linear
  w.r.t. the length of an input,
  the task: to obtain $?d\ x + ?d'$ for output length.

# $\Psi$ is complex to infer

We want to use the type system for

"parametric type-checking"

.

E.g. : I expect that my program

- has something like quadratic heap consumption, the task: to obtain $?a\ x^2 + ?b\ x + ?c$ for heap,
- and has the length of the output is linear w.r.t. the length of an input, the task: to obtain $?d\ x + ?d'$ for output length.

# $\Psi$ is complex to infer

We want to use the type system for

"parametric type-checking"

.
E.g. : I expect that my program

- has something like quadratic heap consumption, the task: to obtain $?a\ x^2 + ?b\ x + ?c$ for heap,
- and has the length of the output is linear w.r.t. the length of an input, the task: to obtain $?d\ x + ?d'$ for output length.

# Skip it: Destructive match

$$\frac{\Gamma,\, n \vdash \left[\frac{e_1 \,:\, T'}{\Delta}\right],\, n' \qquad \Gamma,\, h : T,\, t : \mathrm{L}_{l-1}(T,\, k),\, n + 1 + k(l) \vdash \left[\frac{e_2 \,:\, T'}{\Delta,\, h : T,\, t : \mathrm{L}_{l-1}(T,\, k')}\right] \quad (\ast\text{the benign sharing for Match}\ast)}{\Gamma,\, t : \mathrm{L}_l(T,\, k),\, n \vdash \left[\frac{\begin{array}{l}\mathtt{match\ x\ with} \qquad : T \\ \mathtt{Nil} \Rightarrow e_1 \\ \mid \mathtt{cons(h,\ t)@\_} \Rightarrow e_2\end{array}}{\Delta,\, x : \mathrm{L}_l(T,\, k')}\right],\, n'}$$

# Let: Sharing is not a Monster

$$Int \oplus Int = Int$$

$$\mathrm{L}_l(T_1,\ k_1) \oplus \mathrm{L}_l(T_2,\ k_2) = \mathrm{L}_l(T_1 \oplus T_2,\ k_1 + k_2)$$

$$\mathrm{L}(T_1,\ k_1) \oplus \mathrm{L}(T_2,\ k_2) = \mathrm{L}(T_1 \oplus T_2,\ k_1 + k_2)$$

$$\left(\Gamma_1 \uplus \Gamma_2\right)(x) = \begin{array}{ll} \Gamma_1(x) & x \in dom(\Gamma_1) \setminus dom(\Gamma_2) \\ \Gamma_2(x) & x \in dom(\Gamma_2) \setminus dom(\Gamma_1) \\ \Gamma_1(x) + \Gamma_2(x) & x \in dom(\Gamma_1) \cap dom(\Gamma_2) \end{array}$$

# Skip it: Let

$$\dfrac{\begin{array}{l}\Gamma_1, n \vdash \left[\dfrac{e_1 : T_0}{\Delta_1}\right], n_0 \\ \Gamma_2, x : T_0, n_0 \vdash \left[\dfrac{e_2 : T}{\Delta_2, x : Z(T_0)}\right], n' \\ (\ast\text{the benign sharing for Let}\ast)\end{array}}{\Gamma_1 \uplus \Gamma_2, n \vdash \left[\dfrac{\begin{array}{c}\texttt{let}\quad \texttt{x}=e_1 \quad : T \\ \texttt{in}\; e_2\end{array}}{\Delta_1 \uplus \Delta_2}\right], n'}$$

# Some Rules: Budget

$$\frac{\Gamma, n \vdash \left[\frac{e\ :\ T}{\Delta}\right],\ n' \qquad n \le r \quad r' \le n}{\Gamma, r \vdash \left[\frac{e\ :\ T}{\Delta'}\right],\ r'}$$

$$\frac{r \ge 0 \qquad \Gamma, n \vdash \left[\frac{e\ :\ T}{\Delta}\right],\ n'}{\Gamma, n + r \vdash \left[\frac{e\ :\ T}{\Delta}\right],\ n' + r}$$

# Shuffle

$$\frac{\Gamma,\, x : \mathrm{L}_l(T,\, k),\, n \vdash \left[\dfrac{e : T'}{\Delta,\, x : \mathrm{L}_l(T,\, k')}\right],\, n' \quad k \geq k''}{\Gamma,\, x : \mathrm{L}_l(T,\, k - k''),\, n + \sum_{i=1}^{l} k''(i) \vdash \left[\dfrac{e : T'}{\Delta,\, x : \mathrm{L}_l(T,\, k')}\right],\, n'}$$

# Some Rules: Weakening

$$\frac{\Gamma, n \vdash \left[\dfrac{e : T}{\Delta}\right], n'}{\Gamma, \Theta, n \vdash \left[\dfrac{e : T}{\Delta, \Theta}\right], n'}$$

# Well-defined First-Order Signature

A first-order *signature* $\Sigma$ *is well-defined*
if for any function $f \in dom(\Sigma)$ with $\Sigma(f) =$
$\mathrm{L}_l(T, k/k''), k_0 \ \rightarrow \mathrm{L}_{l'}(T', k'), \ k'_0 \parallel \Psi(l, l', k, k'', k_0, k', k'_0)$
one can successfully type-check the body $e_{\mathtt{f}}$ of $\mathtt{f}$:

$$x : \mathrm{L}_l(T, k), k_0 \vdash \left[\frac{e_{\mathtt{f}}(x) : \mathrm{L}_{l'}(T', k')}{x : \mathrm{L}_l(T, k'')}\right], k'_0$$

provided that $\Psi(l, l', k, k'', k_0, k', k'_0)$ holds.

# Well-defined First-Order Signature

A first-order *signature* $\Sigma$ *is well-defined* if for any function $f \in dom(\Sigma)$ with $\Sigma(f) = \mathrm{L}_l(T, k/k''), k_0 \ \to \mathrm{L}_{l'}(T', k'), \ k'_0 \parallel \Psi(l, l', k, k'', k_0, k', k'_0)$ one can successfully type-check the body $e_{\mathtt{f}}$ of $\mathtt{f}$:

$$x : \mathrm{L}_l(T, k), k_0 \vdash \left[\frac{e_{\mathtt{f}}(x) : \mathrm{L}_{l'}(T', k')}{x : \mathrm{L}_l(T, k'')}\right], \ k'_0$$

provided that $\Psi(l, l', k, k'', k_0, k', k'_0)$ holds.

# Example: Destructive Half

leaves every 2nd element of an input list

```
half x =  match x with
          Nil => Nil
          | cons(h, t)@_ =>  match t with
                             Nil => Nil
                             | cons(hh, tt)@_ =>
                              let  y=half tt
                                   in cons(hh, y)
```

has typing $\mathrm{L}_l(T, k/k''), 0 \to \mathrm{L}_{l'}(T, k'), 0 \; || \; l' = p(l)$,
where $p(l) = \lfloor \frac{l}{2} \rfloor$ and $k = k' \equiv 0, k'' \equiv \frac{1}{2}$.

# Example: Destructive Half

## leaves every 2nd element of an input list

```
half x =   match x with
           Nil ⇒ Nil
           | cons(h, t)@_ ⇒  match t with
                              Nil ⇒ Nil
                              | cons(hh, tt)@_ ⇒
                               let  y=half tt
                                    in cons(hh, y)
```

has typing $\mathrm{L}_l(T, k/k''), 0 \to \mathrm{L}_{l'}(T, k'), 0 \;\|\; l' = p(l)$,
where $p(l) = \lfloor \frac{l}{2} \rfloor$ and $k = k' \equiv 0, k'' \equiv \frac{1}{2}$.

# Example: Logarithm

```
log x =
 let  y=half x
      in  match y with  Nil ⇒ Nil
                       | cons(h, t) ⇒  let  z=log y
                                            in cons(1, z)
```

If list $x$ has length $l$, then the program frees $l$ heap units but consumes $O(\log_2(l))$.

Type-checked the credit functions $k(x) = \dfrac{a}{x}$, $k''(x) \equiv 1$, have found that $a = 2$.

# The Problem(s): Is the Type System Refineable?

- merge the “budget rules” with the syntactical ones as much as possible, to reduce complexity of type-checking, find heuristics for the “shuffle rules”,
- non-strict sizes (if-rule is restrictive, ... )???
- add the number of recursive calls as a parameter for first-order types?
- (very) dependent types for the fine “if”-rule and recursive calls?
- verify calls “in-the-context” for fine treatment of compositions?

# The Problem(s): Is the Type System Refineable?

- merge the “budget rules” with the syntactical ones as much as possible, to reduce complexity of type-checking, find heuristics for the “shuffle rules”,
- non-strict sizes (if-rule is restrictive, ... )???
- add the number of recursive calls as a parameter for first-order types?
- (very) dependent types for the fine “if”-rule and recursive calls?
- verify calls “in-the-context” for fine treatment of compositions?

# The Problem(s): Is the Type System Refineable?

- merge the “budget rules” with the syntactical ones as much as possible, to reduce complexity of type-checking, find heuristics for the “shuffle rules”,
- non-strict sizes (if-rule is restrictive, ... )???
- add the number of recursive calls as a parameter for first-order types?
- (very) dependent types for the fine “if”-rule and recursive calls?
- verify calls “in-the-context” for fine treatment of compositions?

# The Problem(s): Is the Type System Refineable?

- merge the “budget rules” with the syntactical ones as much as possible, to reduce complexity of type-checking, find heuristics for the “shuffle rules”,
- non-strict sizes (if-rule is restrictive, ... )???
- add the number of recursive calls as a parameter for first-order types?
- (very) dependent types for the fine “if”-rule and recursive calls?
- verify calls “in-the-context” for fine treatment of compositions?

# The Problem(s): Is the Type System Refineable?

- merge the “budget rules” with the syntactical ones as much as possible, to reduce complexity of type-checking, find heuristics for the “shuffle rules”,
- non-strict sizes (if-rule is restrictive, ... )???
- add the number of recursive calls as a parameter for first-order types?
- (very) dependent types for the fine “if”-rule and recursive calls?
- verify calls “in-the-context” for fine treatment of compositions?

# Potential = the sum of the credits of all nodes

The list [ [10, 20, 30], [10] ]
of type $\mathrm{L}(\mathrm{L}(\mathit{Int}, k_1), k_2)$ with $k_1(x) = x$, $k_2(x) = 2x$
has the potential $2 \cdot 1 + (1) + 2 \cdot 2 + (1 + 2 + 3)$

# Potential = the sum of the credits of all nodes

The list [ [10, 20, 30], [10] ]
of type $\mathrm{L}(\mathrm{L}(Int, k_1), k_2)$ with $k_1(x) = x, \; k_2(x) = 2x$
has the potential $2 \cdot \; 1 \; + \; (1) \; + \; 2 \cdot \; 2 \; + \; (1 + 2 + 3)$

# Potential = the sum of the credits of all nodes

The list [ [10, 20, 30], [10] ]
of type $\mathrm{L}(\mathrm{L}(\mathit{Int}, k_1), k_2)$ with $k_1(x) = x,\ k_2(x) = 2x$
has the potential $2 \cdot\ 1\ +\ (1)\ +\ 2 \cdot\ 2\ +\ (1 + 2 + 3)$

# Potential is a dynamic notion

$\Phi : \mathit{Heap} \times \mathit{Val} \times T \longrightarrow \mathbb{R}^{+}$ is defined as

$$\Phi\Big(h,\ v,\ \mathit{Int}\Big) := 0,$$
$$\Phi\Big(h,\ \mathrm{null},\ \mathrm{L}_0(T,\ k)\Big) := 0,$$
$$\Phi\Big(h,\ \ell,\ \mathrm{L}_l(T,\ k)\Big) := \Phi\Big(h,\ h.\ell.\mathrm{HD},\ T\Big) + k(l) + \Phi\Big(h,\ h.\ell.\mathrm{TL},\ \mathrm{L}_{l-1}(T,\ k)\Big)$$
*for* $\ell \neq \mathrm{null}$,

$$\Phi\Big(h,\ \ell,\ \mathrm{L}(T,\ k)\Big) := \Phi\Big(h,\ \ell,\ \mathrm{L}_l(T,\ k)\Big),\ \mathit{where}\ l = D(h,\ \ell).$$

Extended to stack environments and typing contexts:
$\Phi\Big(h,\ E,\ \Gamma\Big) = \sum_{x \in dom(\Gamma)} \Phi\Big(h,\ E(x),\ \Gamma(x)\Big).$

# Potential is a dynamic notion

$\Phi : \mathit{Heap} \times \mathit{Val} \times T \longrightarrow \mathbb{R}^{+}$ is defined as

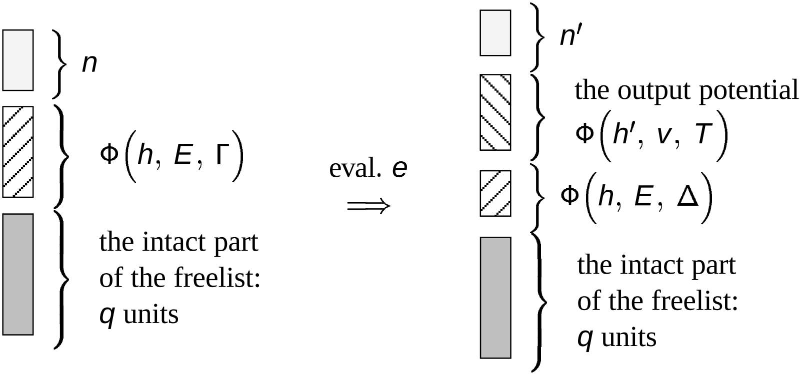
$$\Phi\Big(h,\ v,\ \mathit{Int}\Big) := 0,$$
$$\Phi\Big(h,\ \text{null},\ \mathrm{L}_0(T,\ k)\Big) := 0,$$
$$\Phi\Big(h,\ \ell,\ \mathrm{L}_l(T,\ k)\Big) := \Phi\Big(h,\ h.\ell.\mathrm{HD},\ T\Big) + k(l) +$$
$$\Phi\Big(h,\ h.\ell.\mathrm{TL},\ \mathrm{L}_{l-1}(T,\ k)\Big)$$
$$\textit{for } \ell \neq \text{null},$$

$$\Phi\Big(h,\ \ell,\ \mathrm{L}(T,\ k)\Big) := \Phi\Big(h,\ \ell,\ \mathrm{L}_l(T,\ k)\Big),\ \textit{where } l = D(h,\ \ell).$$

Extended to stack environments and typing contexts:

$$\Phi\Big(h,\ E,\ \Gamma\Big) = \sum_{x \in dom(\Gamma)} \Phi\Big(h,\ E(x),\ \Gamma(x)\Big).$$

# Soundness with the Feelist Model



# The Problem: Which Prover

I trust myself, but:
it would be more convenient to prove the soundness of the present system using a proof assistant,
proviso: the operational semantics was already incoded.
... and the things become more complicated...

General question:
If one needs to encode the gentleman's set:

- the syntax of the language,
- the operational semantics,
- the semantics of a typing judgement,
- the soundness proofs,

which prover to choose?

# The Problem: Which Prover

I trust myself, but:
it would be more convenient to prove the soundness of the present system using a proof assistant,
proviso: the operational semantics was already incoded.
... and the things become more complicated...

General question:
If one needs to encode the gentleman's set:

- the syntax of the language,
- the operational semantics,
- the semantics of a typing judgement,
- the soundness proofs,

which prover to choose?

# The Problem: Which Prover

I trust myself, but:
it would be more convenient to prove the soundness of the present system using a proof assistant,
proviso: the operational semantics was already incoded.
... and the things become more complicated...

General question:
If one needs to encode the gentleman's set:

- the syntax of the language,
- the operational semantics,
- the semantics of a typing judgement,
- the soundness proofs,

which prover to choose?

# The Problem: Which Prover

I trust myself, but:
it would be more convenient to prove the soundness of the present system using a proof assistant,
proviso: the operational semantics was already incoded.
... and the things become more complicated...

General question:
If one needs to encode the gentleman's set:

- the syntax of the language,
- the operational semantics,
- the semantics of a typing judgement,
- the soundness proofs,

which prover to choose?

# Summary

- We have designed
  the heap-space aware, amortization based, type system
  for first-order functional programs over polymorphic lists.
- It generalises Hofmann-Jost type system
  by making annotations variable.
- The system is sound.

## Future Work

- Conider other than lists data structures.
- Adjust the approach for an object-oriented setting
  (code structures which have funcional equivalents,
  (co)algebraic data types,...)

# Summary

- We have designed
  the heap-space aware, amortization based, type system
  for first-order functional programs over polymorphic lists.
- It generalises Hofmann-Jost type system
  by making annotations variable.
- The system is sound.

## Future Work

- Conider other than lists data structures.
- Adjust the approach for an object-oriented setting
  (code structures which have funcional equivalents,
  (co)algebraic data types,...)

# Summary

- We have designed
  the heap-space aware, amortization based, type system
  for first-order functional programs over polymorphic lists.
- It generalises Hofmann-Jost type system
  by making annotations variable.
- The system is sound.

## Future Work

- Conider other than lists data structures.
- Adjust the approach for an object-oriented setting
  (code structures which have funcional equivalents,
  (co)algebraic data types,...)

# Summary

- We have designed
  the heap-space aware, amortization based, type system
  for first-order functional programs over polymorphic lists.
- It generalises Hofmann-Jost type system
  by making annotations variable.
- The system is sound.

Future Work

- Conider other than lists data structures.
- Adjust the approach for an object-oriented setting
  (code structures which have funcional equivalents,
  (co)algebraic data types,...)

# Summary

- We have designed
  the heap-space aware, amortization based, type system
  for first-order functional programs over polymorphic lists.
- It generalises Hofmann-Jost type system
  by making annotations variable.
- The system is sound.

Future Work

- Conider other than lists data structures.
- Adjust the approach for an object-oriented setting
  (code structures which have funcional equivalents,
  (co)algebraic data types,...)