An Efficient Representation for Lazy Constructors using 64-bit Pointers

Georgios Fourtounis (gfour@softlab.ntua.gr)
Nikolaos Papaspyrou (nickie@softlab.ntua.gr)

National Technical University of Athens
School of Electrical and Computer Engineering
Software Engineering Laboratory

The 3rd ACM SIGPLAN Workshop on Functional High-Performance Computing (FHP’14)
Göteborg, Sweden, September 4, 2014

Work supported by the project Handling Uncertainty in Data Intensive Applications, co-financed by the European Union (European Social Fund — ESF) and Greek national funds, through the Operational Program “Education and Lifelong Learning”, under the program THALES.
In case you fall asleep...

What the *paper* is about

An efficient implementation technique based on tagged pointers on AMD64, used in a compiler for a subset of Haskell
In case you fall asleep...

What the **paper** is about

An efficient implementation technique based on tagged pointers on AMD64, used in a compiler for a subset of Haskell

What this **talk** will be about
In case you fall asleep...

What the paper is about

An efficient implementation technique based on tagged pointers on AMD64, used in a compiler for a subset of Haskell

What this talk will be about

AND NOW FOR SOMETHING COMPLETELY DIFFERENT
In case you fall asleep...

What the **paper** is about

An efficient implementation technique based on tagged pointers on AMD64, used in a compiler for a subset of Haskell

What this **talk** will be about

1. An outline of the techniques used in the **gic** compiler that enable this optimization on AMD64
In case you fall asleep...

What the **paper** is about

An efficient implementation technique based on tagged pointers on AMD64, used in a compiler for a subset of Haskell

What this **talk** will be about

1. An outline of the techniques used in the **gic** compiler that enable this optimization on AMD64
   - The intensional transformation
What the **paper** is about

An efficient implementation technique based on tagged pointers on AMD64, used in a compiler for a subset of Haskell

What this **talk** will be about

1. An outline of the techniques used in the *gic* compiler that enable this optimization on AMD64
   - The intensional transformation
   - Lazy activation records
In case you fall asleep...

What the paper is about

An efficient implementation technique based on tagged pointers on AMD64, used in a compiler for a subset of Haskell

What this talk will be about

1. An outline of the techniques used in the gic compiler that enable this optimization on AMD64
   - The intensional transformation
   - Lazy activation records

2. flip filter thePaper $
   \text{liftA2} (&&) \text{isInteresting canBePresentedHere}$
The intensional transformation

Alternative technique for implementing non-strict functional languages by transformation to dataflow programs

Dataflow programming languages

Dataflow programming:
- A program is a directed graph of data flowing through a network of processing units
- Quite popular in the 1980s due to its implicitly parallel nature

Figure from Joey Paquet's PhD thesis, “Intensional Scientific Programming” (1999)
Dataflow programming languages

Dataflow programming:
- A program is a directed graph of **data** flowing through a network of **processing units**
- Quite popular in the 1980s due to its implicitly parallel nature

Dataflow languages:
- Mostly **functional** in nature, encouraging **stream processing**
- **Examples**: Val, Id, Lucid, GLU, SISAL, etc.
Dataflow programming:

- A program is a directed graph of data flowing through a network of processing units.
- Quite popular in the 1980s due to its implicitly parallel nature.

Dataflow languages:

- Mostly functional in nature, encouraging stream processing.
- Examples: Val, Id, Lucid, GLU, SISAL, etc.

Dataflow machines:

- Specialized parallel architectures for executing dataflow programs, e.g. the MIT Tagged-Token Machine.
- Execution is determined by the availability of input arguments to operations.
The status of dataflow

In the 1990s:
- Interest started to decline
- Dataflow architectures 😞 vs. mainstream uniprocessors 🤣
The status of dataflow

In the 1990s:
- Interest started to decline
- Dataflow architectures 🔄 vs. mainstream uniprocessors ☑️

Today:
- Renewed interest
- Uniprocessors no longer follow Moore’s law
- Commodity parallel hardware on the rise
- A new generation of dataflow-esque languages/programming models: Dryad, Cluster, Hyrax, Map-Reduce, etc.
- Efficient implementation in mainstream multi-core architectures and reconfigurable hardware (FPGAs)
*Intensional transformation by example*

The input is a first-order functional program. The output is a program with parameterless definitions (intensional program).

**Example**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>result</code></td>
<td><code>= f 3 + f 5</code></td>
</tr>
<tr>
<td><code>f x</code></td>
<td><code>= g (x*x)</code></td>
</tr>
<tr>
<td><code>g y</code></td>
<td><code>= y+2</code></td>
</tr>
</tbody>
</table>
Intensional transformation by example

The input is a first-order functional program. The output is a program with parameterless definitions (intensional program).

Example

<table>
<thead>
<tr>
<th>result</th>
<th>=  f 3 + f 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>f x</td>
<td>= g (x*x)</td>
</tr>
<tr>
<td>g y</td>
<td>= y+2</td>
</tr>
</tbody>
</table>

Step 1: for all functions \( f \)

- Replace the \( i \)-th call of \( f \) by \( \text{call}_i(f) \)
- Remove formal parameters from function definitions
Intensional transformation by example

The input is a first-order functional program. The output is a program with parameterless definitions (intensional program).

Example

| f x = g (x*x) | f = call_0(g) |
| g y = y+2     | g = y+2      |
| result = f 3  + f 5 | result = call_0(f)+call_1(f) |

Step 1: for all functions $f$

- Replace the $i$-th call of $f$ by $\text{call}_i(f)$
- Remove formal parameters from function definitions
Intensional transformation by example

The input is a first-order functional program. The output is a program with parameterless definitions (intensional program).

Example

<table>
<thead>
<tr>
<th>result = f 3 + f 5</th>
<th>result = call_0(f)+call_1(f)</th>
</tr>
</thead>
<tbody>
<tr>
<td>f x = g (x*x)</td>
<td>f = call_0(g)</td>
</tr>
<tr>
<td>g y = y+2</td>
<td>g = y+2</td>
</tr>
</tbody>
</table>

Step 2: for all functions $f$, for all formal parameters $x$

- Find actual parameters corresponding to $x$ in all calls of $f$
- Introduce a new definition for $x$ with an `actuals` clause, listing the actual parameters in the order of the calls
Intensional transformation by example

The input is a first-order functional program. The output is a program with parameterless definitions (intensional program).

Example

\[
\begin{align*}
\text{result} &= f\ 3\ +\ f\ 5 \\
f\ x &= g\ (x*x) \\
g\ y &= y+2
\end{align*}
\]

\[
\begin{align*}
\text{result} &= \text{call}_0(f)+\text{call}_1(f) \\
f &= \text{call}_0(g) \\
g &= y+2 \\
x &= \text{actuals}(3, 5) \\
y &= \text{actuals}(x*x)
\end{align*}
\]

Step 2: for all functions \( f \), for all formal parameters \( x \)

- Find actual parameters corresponding to \( x \) in all calls of \( f \)
- Introduce a new definition for \( x \) with an \texttt{actuals} clause, listing the actual parameters in the order of the calls
Evaluation of expressions: $EVAL(e, w)$

- **Intensional**: with respect to a context $w$
- Evaluation contexts are lists of natural numbers
- The initial context is the empty list
Semantics of the target language

Evaluation of expressions: $EVAL(e, w)$

- **Intensional**: with respect to a context $w$
- Evaluation contexts are **lists** of natural numbers
- The **initial** context is the empty list

Context switching: call and actuals

$$EVAL(call_i(e), w) = EVAL(e, i : w)$$
$$EVAL(actuals(e_0, \ldots, e_{n-1}), i : w) = EVAL(e_i, w)$$
Example

Evaluation of the target program:

\[
\begin{align*}
\text{result} & = \text{call}_0(f) + \text{call}_1(f) \\
f & = \text{call}_0(g) \\
g & = y + 2 \\
x & = \text{actuals}(3, 5) \\
y & = \text{actuals}(x \times x)
\end{align*}
\]

\[
\begin{align*}
\text{result} &= \text{EVAL}(\text{result}, []) \\
&= \text{EVAL}(\text{call}_0(f) + \text{call}_1(f), []) \\
&= \text{EVAL}(\text{call}_0(f), []) + \text{EVAL}(\text{call}_1(f), []) \\
&= \text{EVAL}(f, [0]) + \text{EVAL}(f, [1]) \\
&= \text{EVAL}(\text{call}_0(g), [0]) + \text{EVAL}(\text{call}_0(g), [1]) \\
&= \text{EVAL}(g, [0, 0]) + \text{EVAL}(g, [0, 1]) \\
&= \text{EVAL}(y, [0, 0]) + \text{EVAL}(2, [0, 0]) + \text{EVAL}(y, [0, 1]) + \text{EVAL}(2, [0, 1]) \\
&= \text{EVAL}(\text{actuals}(x \times x), [0, 0]) + 2 + \text{EVAL}(\text{actuals}(x \times x), [0, 1]) + 2 \\
&= \text{EVAL}(x \times x, [0]) + 2 + \text{EVAL}(x \times x, [1]) + 2 \\
&= \text{EVAL}(x, [0]) \times \text{EVAL}(x, [0]) + 2 + \text{EVAL}(x, [1]) \times \text{EVAL}(x, [1]) + 2 \\
&= \text{EVAL}(\text{actuals}(3, 5), [0]) \times \text{EVAL}(\text{actuals}(3, 5), [0]) + 2 + \text{EVAL}(\text{actuals}(3, 5), [1]) \times \text{EVAL}(\text{actuals}(3, 5), [1]) + 2 \\
&= \text{EVAL}(3, []) \times \text{EVAL}(3, []) + 2 + \text{EVAL}(5, []) \times \text{EVAL}(5, []) + 2 \\
&= 3 \times 3 + 2 + 5 \times 5 + 2 \\
&= 38
\end{align*}
\]
Example

\[\text{result} = f \, 3 \, + \, f \, 5\]
\[f \, x \, = \, g \, (x \times x)\]
\[g \, y \, = \, y + 2\]

\[\text{result} \, = \, \text{call}_0(f) + \text{call}_1(f)\]
\[f \, = \, \text{call}_0(g)\]
\[g \, = \, y + 2\]
\[x \, = \, \text{actuals}(3, 5)\]
\[y \, = \, \text{actuals}(x \times x)\]
The idea

- A program with functions of order up to $n$ is translated to an intensional program using $n$-dimensional contexts
- Dimension $i$ has its own pair of call$^i$ and actuals$^i$ operators
- Transformation works in steps, each time decreasing the program’s order by one

Example (A second order program that can be transformed)

```plaintext
result = double inc 3
double f x = f (f x)
inc y = y + 1
```
Missing pieces

The original intensional transformation lacks:

1. User-defined data structures:

```plaintext
data List = Nil | Cons Int List

length ls =
    case ls of
        Nil → 0
        Cons x xs → 1 + length xs
```
The original intensional transformation lacks:

1. User-defined data structures:

   ```
   data List = Nil | Cons Int List
   length ls =
     case ls of
       Nil → 0
       Cons x xs → 1 + length xs
   ```

2. Partial application:

   ```
   result = double (add 1) 3
   double f x = f (f x)
   add a b = a + b
   ```
The original intensional transformation lacks:

1. **User-defined data structures:**
   ```haskell
data List = Nil | Cons Int List
length ls =
    case ls of
        Nil → 0
        Cons x xs → 1 + length xs
```

2. **Partial application:**
   ```haskell
result = double (add 1) 3
double f x = f (f x)
add a b = a + b
```

👍 Problem (2) reduced to (1) with **defunctionalization**
The complete picture: The gic compiler

https://github.com/gfour/gic

Key ideas:

- Compiles a large subset of Haskell 98
- Modular defunctionalization handles higher-order programs, supporting separate compilation
  [Fourtounis, Papaspyrou & Theofilopoulos, 2014]
- Generalized intensional transformation handles first-order programs with user-defined data types
  [Fourtounis, Papaspyrou & Rondogiannis, 2013]
- Currently supercombinator-based, using lambda-lifting
- Implementation on mainstream hardware using an adaptation of lazy activation records
  [Charalambidis, Grivas, Papaspyrou & Rondogiannis, 2008]
Implementation using lazy activation records (LARs)

- Target code is **low-level C**:
  - can be compiled using gcc, clang, icc
  - uses GNU extension: statement expressions
- Each Haskell function becomes a C function taking a LAR
- Each Haskell constructor $c$ becomes a C function taking a LAR $l$ and returning the pair $(c, l)$
- **Push/enter model**: Push LAR, then enter function
- Memory use:
  - the C stack is used for control (function entry-return)
  - LARs are usually placed on the heap, but a cheap analysis enables their placement on the stack
### Characteristics:

- LARs efficiently implement **intensional contexts** and are also used for storing actual parameters.
- LARs are **lazy**, i.e., the actual parameters are filled in upon demand.
- LAR construction is controlled by the presence of intensional operators in the target program; destruction may require **GC**.
The anatomy of a LAR

Characteristics:

- LARs efficiently implement **intensional contexts** and are also used for storing actual parameters.
- LARs are **lazy**, i.e. the actual parameters are filled in upon demand.
- LAR construction is controlled by the presence of intensional operators in the target program; destruction may require **GC**.

Contents:

Each LAR contains three fields:

- `prev` access link to parent LAR
- `vals` thunks for function parameters
- `nested` escaping LARs reached through pattern-matching, used to evaluate lazy constructor fields
The anatomy of a LAR (ii)

\[
f x y = \text{case } x \text{ of } [] \rightarrow [1] \\
a:as \rightarrow [a + y]
\]
The anatomy of a LAR (iii)

Representation of thunks:

A thunk must have three fields:

- **flag** set if the thunk is evaluated
- **code** pointer to code that must be run in order to evaluate the thunk
- **value** the already computed value of an evaluated thunk
Representation of thunks:

A thunk must have three fields:

- **flag**: set if the thunk is evaluated
- **code**: pointer to code that must be run in order to evaluate the thunk
- **value**: the already computed value of an evaluated thunk

Representation of values:

A value can be:

- a primitive integer $n$ (for enumerations, e.g. Bool and Int)
- a lazy constructor of the form $(c, l)$
Pointers in AMD64 leave a lot of space unused:

![Diagram showing sign-extended bit 47 and pointer body]

Remarks:
- This is true for pointers to the **heap** (using a suitable memory allocator, e.g. `malloc` or our own) and to the **stack**
- Code pointers can also be aligned (`gcc -falign-functions`)
- This permits a tagged-pointer implementation for **thunks**
Compact thunk representation

Unevaluated thunk

Lazy constructor

Primitive value

Thunk data  Thunk flag  Value flag  Unused
Benchmarks

What we’re comparing

- Comparison with a previous, non-compact representation
- Comparison with fully optimizing `ghc` (-03)
  - using the default `ghc` back-end
  - using three different `gic` back-ends: `gcc`, `clang` and `icc`
### Benchmarks

**What we’re comparing**

- Comparison with a previous, non-compact representation
- Comparison with fully optimizing `ghc (-O3)`
  - using the default `ghc` back-end
  - using three different `gic` back-ends: gcc, clang and icc

- We only use two basic variants of well-known optimizations (escape analysis, usage analysis), garbage collection was used
- (Micro/mini)-benchmarks, since we still don’t have a front-end for full Haskell (e.g. complex patterns, type classes, full library)
Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Runtime</th>
<th>Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>ghc-7.6.3, [-O3]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>gic/clang-3.3, [-O3]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>gic/gcc-4.7.2, [-O3]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>gic/icc-14.0.2, [-fast]</td>
</tr>
<tr>
<td>ack</td>
<td></td>
<td></td>
</tr>
<tr>
<td>collatz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>digits_of_e1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fib</td>
<td></td>
<td></td>
</tr>
<tr>
<td>htree</td>
<td></td>
<td></td>
</tr>
<tr>
<td>primes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>church</td>
<td></td>
<td></td>
</tr>
<tr>
<td>queens</td>
<td></td>
<td></td>
</tr>
<tr>
<td>queens_num</td>
<td></td>
<td></td>
</tr>
<tr>
<td>quicksort</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reverse</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tree-sort</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summary:

- Outline of \texttt{gic}, an alternative compiler for a subset of Haskell
- Efficient implementation for \texttt{gic} with tagged pointers on AMD64
So, in case you just woke up...

Summary:
- Outline of **gic**, an alternative compiler for a subset of Haskell
- Efficient implementation for **gic** with tagged pointers on AMD64

Concluding remarks:
- Promising performance; many optimizations still missing: code transformations (e.g. fusion, inlining), tail-call optimization, strictness analysis
- Overhead on math-heavy computations, pointers pass through the ALU
- Compact memory use (∼67% reduction)
- The technique can also be used with ARM, SPARC processors
Work in progress:

- The explicit pointer stack is expensive, working on a DWARF-based garbage collector for minimal overhead.
- Tail-call optimization: also useful for defunctionalization; delicate interplay between C and non-strictness.
- Add type classes using the concretization encoding of Pottier and Gauthier.
- Take out the lambda-lifter and support local definitions.
- Provide \texttt{gic} as a back-end for \texttt{ghc}.
So, in case you just woke up...

Work in progress:

- The explicit pointer stack is expensive, working on a DWARF-based garbage collector for minimal overhead
- Tail-call optimization: also useful for defunctionalization; delicate interplay between C and non-strictness
- Add type classes using the concretization encoding of Pottier and Gauthier
- Take out the lambda-lifter and support local definitions
- Provide `gic` as a back-end for `ghc`

Future work:

- More optimizations
- Parallel runtime
Thank you!
Example

Evaluation of the target program:

\[ \text{EVAL}(\text{result}, []) \]

result = call_0(f) + call_1(f)
f = call_0(g)
g = y + 2
x = actuals(3, 5)
y = actuals(x * x)
Example

Evaluation of the target program:

\[
EVAL(result, []) = EVAL(call_0(f) + call_1(f), [])
\]

result = call_0(f) + call_1(f)
f = call_0(g)
g = y + 2
x = actuals(3, 5)
y = actuals(x*x)
Example

Evaluation of the target program:

\[
EVAL(\text{result}, []) = EVAL(\text{call}_0(f) + \text{call}_1(f), []) = EVAL(\text{call}_0(f), []) + EVAL(\text{call}_1(f), [])
\]
Result of the target program:

\[
\text{EVAL}(\text{result}, [\ ])
= \text{EVAL}(\text{call}_0(f) + \text{call}_1(f), [\ ])
= \text{EVAL}(\text{call}_0(f), [\ ]) + \text{EVAL}(\text{call}_1(f), [\ ])
= \text{EVAL}(f, [0]) + \text{EVAL}(f, [1])
\]
Example

Evaluation of the target program:

\[
\text{EVAL}(\text{result}, []) = \text{EVAL}(\text{call}_0(f) + \text{call}_1(f), []) \\
= \text{EVAL}(\text{call}_0(f), []) + \text{EVAL}(\text{call}_1(f), []) \\
= \text{EVAL}(f, [0]) + \text{EVAL}(f, [1]) \\
= \text{EVAL}(\text{call}_0(g), [0]) + \text{EVAL}(\text{call}_0(g), [1])
\]

result = call_0(f) + call_1(f)
f = call_0(g)
g = y+2
x = actuals(3, 5)
y = actuals(x*x)
Example

Evaluation of the target program:

\[
EVAL(\text{result}, [ ]) = EVAL(\text{call}_0(f) + \text{call}_1(f), [ ]) = EVAL(\text{call}_0(f), [ ]) + EVAL(\text{call}_1(f), [ ]) = EVAL(f, [0]) + EVAL(f, [1]) = EVAL(\text{call}_0(g), [0]) + EVAL(\text{call}_0(g), [1]) = EVAL(g, [0, 0]) + EVAL(g, [0, 1])
\]

result = call_0(f) + call_1(f)
f = call_0(g)
g = y + 2
x = actuals(3, 5)
y = actuals(x*x)
Example

Evaluation of the target program:

\[
EVAL(\text{result}, []) = EVAL(\text{call}_0(f) + \text{call}_1(f), []) \\
= EVAL(\text{call}_0(f), []) + EVAL(\text{call}_1(f), []) \\
= EVAL(f, [0]) + EVAL(f, [1]) \\
= EVAL(\text{call}_0(g), [0]) + EVAL(\text{call}_0(g), [1]) \\
= EVAL(g, [0, 0]) + EVAL(g, [0, 1]) \\
= EVAL(y, [0, 0]) + EVAL(2, [0, 0]) + EVAL(y, [0, 1]) + EVAL(2, [0, 1])
\]

\[
\text{result} = \text{call}_0(f) + \text{call}_1(f) \\
f = \text{call}_0(g) \\
g = y + 2 \\
x = \text{actuals}(3, 5) \\
y = \text{actuals}(x \times x)
\]
Example

Evaluation of the target program:

\[
\begin{align*}
\text{EVAL}(\text{result}, [\ ]) &= \text{EVAL}(\text{call}_0(f) + \text{call}_1(f), [\ ]) \\
&= \text{EVAL}(\text{call}_0(f), [\ ]) + \text{EVAL}(\text{call}_1(f), [\ ]) \\
&= \text{EVAL}(f, [0]) + \text{EVAL}(f, [1]) \\
&= \text{EVAL}(\text{call}_0(g), [0]) + \text{EVAL}(\text{call}_0(g), [1]) \\
&= \text{EVAL}(g, [0, 0]) + \text{EVAL}(g, [0, 1]) \\
&= \text{EVAL}(y, [0, 0]) + \text{EVAL}(2, [0, 0]) + \text{EVAL}(y, [0, 1]) + \text{EVAL}(2, [0, 1]) \\
&= \text{EVAL}(\text{actuals}(x \times x), [0, 0]) + 2 + \text{EVAL}(\text{actuals}(x \times x), [0, 1]) + 2
\end{align*}
\]

result = call_0(f) + call_1(f)
f = call_0(g)
g = y + 2
x = actuals(3, 5)
y = actuals(x \times x)
Example

Evaluation of the target program:

\[
EVAL(\text{result}, [])
= EVAL(\text{call}_0(f) + \text{call}_1(f), [])
= EVAL(\text{call}_0(f), []) + EVAL(\text{call}_1(f), [])
= EVAL(f, [0]) + EVAL(f, [1])
= EVAL(\text{call}_0(g), [0]) + EVAL(\text{call}_0(g), [1])
= EVAL(g, [0, 0]) + EVAL(g, [0, 1])
= EVAL(y, [0, 0]) + EVAL(2, [0, 0]) + EVAL(y, [0, 1]) + EVAL(2, [0, 1])
= EVAL(\text{actuals}(x\times x), [0, 0]) + 2 + EVAL(\text{actuals}(x\times x), [0, 1]) + 2
= EVAL(x\times x, [0]) + 2 + EVAL(x\times x, [1]) + 2
\]

result = call_0(f) + call_1(f)
f = call_0(g)
g = y + 2
x = actuals(3, 5)
y = actuals(x\times x)
Example

Evaluation of the target program:

\[
\text{result} = \text{call}_0(f) + \text{call}_1(f) \\
f = \text{call}_0(g) \\
g = y + 2 \\
x = \text{actuals}(3, 5) \\
y = \text{actuals}(x \times x)
\]

\[
\text{EVAL}(\text{result}, []) = \text{EVAL}(\text{call}_0(f) + \text{call}_1(f), []) \\
= \text{EVAL}(\text{call}_0(f), []) + \text{EVAL}(\text{call}_1(f), []) \\
= \text{EVAL}(f, [0]) + \text{EVAL}(f, [1]) \\
= \text{EVAL}(\text{call}_0(g), [0]) + \text{EVAL}(\text{call}_0(g), [1]) \\
= \text{EVAL}(g, [0, 0]) + \text{EVAL}(g, [0, 1]) \\
= \text{EVAL}(y, [0, 0]) + \text{EVAL}(2, [0, 0]) + \text{EVAL}(y, [0, 1]) + \text{EVAL}(2, [0, 1]) \\
= \text{EVAL}(\text{actuals}(x \times x), [0, 0]) + 2 + \text{EVAL}(\text{actuals}(x \times x), [0, 1]) + 2 \\
= \text{EVAL}(x \times x, [0]) + 2 + \text{EVAL}(x \times x, [1]) + 2 \\
= \text{EVAL}(x, [0]) \times \text{EVAL}(x, [0]) + 2 + \text{EVAL}(x, [1]) \times \text{EVAL}(x, [1]) + 2
\]
Example

Evaluation of the target program:

\[
EVAL(\text{result}, []) \\
= EVAL(\text{call}_0(f) + \text{call}_1(f), []) \\
= EVAL(\text{call}_0(f), []) + EVAL(\text{call}_1(f), []) \\
= EVAL(f, [0]) + EVAL(f, [1]) \\
= EVAL(\text{call}_0(g), [0]) + EVAL(\text{call}_0(g), [1]) \\
= EVAL(g, [0, 0]) + EVAL(g, [0, 1]) \\
= EVAL(y, [0, 0]) + EVAL(2, [0, 0]) + EVAL(y, [0, 1]) + EVAL(2, [0, 1]) \\
= EVAL(\text{actuals}(x*x), [0, 0]) + 2 + EVAL(\text{actuals}(x*x), [0, 1]) + 2 \\
= EVAL(x*x, [0]) + 2 + EVAL(x*x, [1]) + 2 \\
= EVAL(x, [0]) \times EVAL(x, [0]) + 2 + EVAL(x, [1]) \times EVAL(x, [1]) + 2 \\
= EVAL(\text{actuals}(3, 5), [0]) \times EVAL(\text{actuals}(3, 5), [0]) + 2 + \\
EVAL(\text{actuals}(3, 5), [1]) \times EVAL(\text{actuals}(3, 5), [1]) + 2
\]

\[
\text{result} = \text{call}_0(f) + \text{call}_1(f) \\
f = \text{call}_0(g) \\
g = y + 2 \\
x = \text{actuals}(3, 5) \\
y = \text{actuals}(x*x)
\]
Example

Evaluation of the target program:

\[
\begin{align*}
EVAL(\text{result}, [\ ])
&= EVAL(\text{call}_0(f) + \text{call}_1(f), [\ ]) \\
&= EVAL(\text{call}_0(f), [\ ]) + EVAL(\text{call}_1(f), [\ ]) \\
&= EVAL(f, [0]) + EVAL(f, [1]) \\
&= EVAL(\text{call}_0(g), [0]) + EVAL(\text{call}_0(g), [1]) \\
&= EVAL(g, [0, 0]) + EVAL(g, [0, 1]) \\
&= EVAL(y, [0, 0]) + EVAL(2, [0, 0]) + EVAL(y, [0, 1]) + EVAL(2, [0, 1]) \\
&= EVAL(\text{actuals}(x*x), [0, 0]) + 2 + EVAL(\text{actuals}(x*x), [0, 1]) + 2 \\
&= EVAL(x*x, [0]) + 2 + EVAL(x*x, [1]) + 2 \\
&= EVAL(x, [0]) * EVAL(x, [0]) + 2 + EVAL(x, [1]) * EVAL(x, [1]) + 2 \\
&= EVAL(\text{actuals}(3, 5), [0]) * EVAL(\text{actuals}(3, 5), [0]) + 2 + \\
&\quad EVAL(\text{actuals}(3, 5), [1]) * EVAL(\text{actuals}(3, 5), [1]) + 2 \\
&= EVAL(3, [\ ]) * EVAL(3, [\ ]) + 2 + EVAL(5, [\ ]) * EVAL(5, [\ ]) + 2
\end{align*}
\]

result = call_0(f)+call_1(f)
f = call_0(g)
g = y+2
x = actuals(3, 5)
y = actuals(x*x)
Example

Evaluation of the target program:

\[ \text{result} = \text{call}_0(f) + \text{call}_1(f) \]
\[ f = \text{call}_0(g) \]
\[ g = y + 2 \]
\[ x = \text{actuals}(3, 5) \]
\[ y = \text{actuals}(x \times x) \]
Example

Evaluation of the target program:

\[
\text{EVAL} (\text{result}, [])
\]

\[
= \text{EVAL} (\text{call}_0 (f) + \text{call}_1 (f), [])
\]

\[
= \text{EVAL} (\text{call}_0 (f), []) + \text{EVAL} (\text{call}_1 (f), [])
\]

\[
= \text{EVAL} (f, [0]) + \text{EVAL} (f, [1])
\]

\[
= \text{EVAL} (\text{call}_0 (g), [0]) + \text{EVAL} (\text{call}_0 (g), [1])
\]

\[
= \text{EVAL} (g, [0, 0]) + \text{EVAL} (g, [0, 1])
\]

\[
= \text{EVAL} (y, [0, 0]) + \text{EVAL} (2, [0, 0]) + \text{EVAL} (y, [0, 1]) + \text{EVAL} (2, [0, 1])
\]

\[
= \text{EVAL} (\text{actuals} (x*x), [0, 0]) + 2 + \text{EVAL} (\text{actuals} (x*x), [0, 1]) + 2
\]

\[
= \text{EVAL} (x*x, [0]) + 2 + \text{EVAL} (x*x, [1]) + 2
\]

\[
= \text{EVAL} (x, [0]) * \text{EVAL} (x, [0]) + 2 + \text{EVAL} (x, [1]) * \text{EVAL} (x, [1]) + 2
\]

\[
= \text{EVAL} (\text{actuals} (3, 5), [0]) * \text{EVAL} (\text{actuals} (3, 5), [0]) + 2 + \text{EVAL} (\text{actuals} (3, 5), [1]) * \text{EVAL} (\text{actuals} (3, 5), [1]) + 2
\]

\[
= \text{EVAL} (3, []) * \text{EVAL} (3, []) + 2 + \text{EVAL} (5, []) * \text{EVAL} (5, []) + 2
\]

\[
= 3 * 3 + 2 + 5 * 5 + 2
\]

\[
= 38
\]
Garbage collection

- Representation compatible with a semi-space garbage collector
- LARs are the only memory objects managed, used for both function calls and constructors
- The `prev` field is a pointer:
  - Each LAR keep its layout description in the high bits of `prev`
  - The field is reused for forwarded addresses
- The `root set` contains LARs passed as arguments to functions still running
  - We currently use an explicit pointer stack (a level of indirection) to keep track of the roots without traversing the C stack
  - This results in at least 2x overhead over the no GC case (but see work in progress at the end of the talk)
Garbage collection interface

- \( a \): arity
- \( n \): nesting
- \( \text{prev} \): access link

\[ \begin{array}{c|c|c|c}
63 & 32 \\
\hline
a: \text{arity} & n: \text{nesting} & \text{prev: access link} \\
\hline
31 & 32 \\
\hline
000 & 000 \\
\end{array} \]

\( \text{prev during runtime} \)

- Forwarded LAR address

\[ \begin{array}{c}
63 \\
\hline
32 \\
\hline
1 \\
\end{array} \]

\( \text{prev of copied LAR} \)

- GC data
- LAR pointer body
- Unused
- GC bit