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 (FHPC'14)
Göteborg, Sweeden, 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

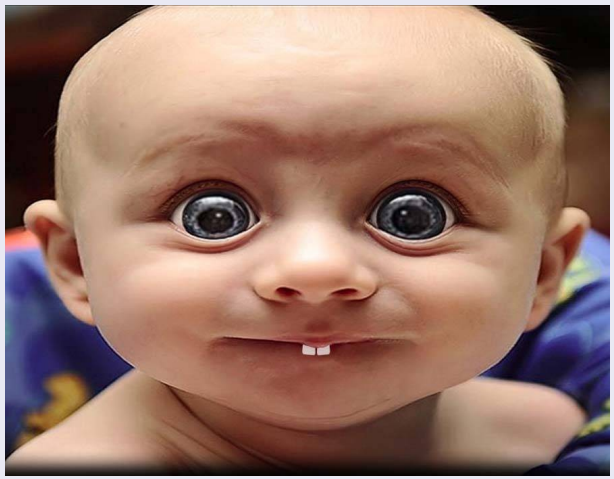
What this **talk** will be about

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



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

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

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 $`
   `liftA2 (&&) isInteresting canBePresentedHere`

The intensional transformation

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

- [Yaghi, 1984] The intensional implementation technique for functional languages.
- [Arvind & Nikhil, 1990] The "coloring" technique for implementing functions on the MIT Dataflow Machine.
- [Rondogiannis & Wadge, 1997, 1999] A formalization of the intensional transformation and its extension for a class of higher-order 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

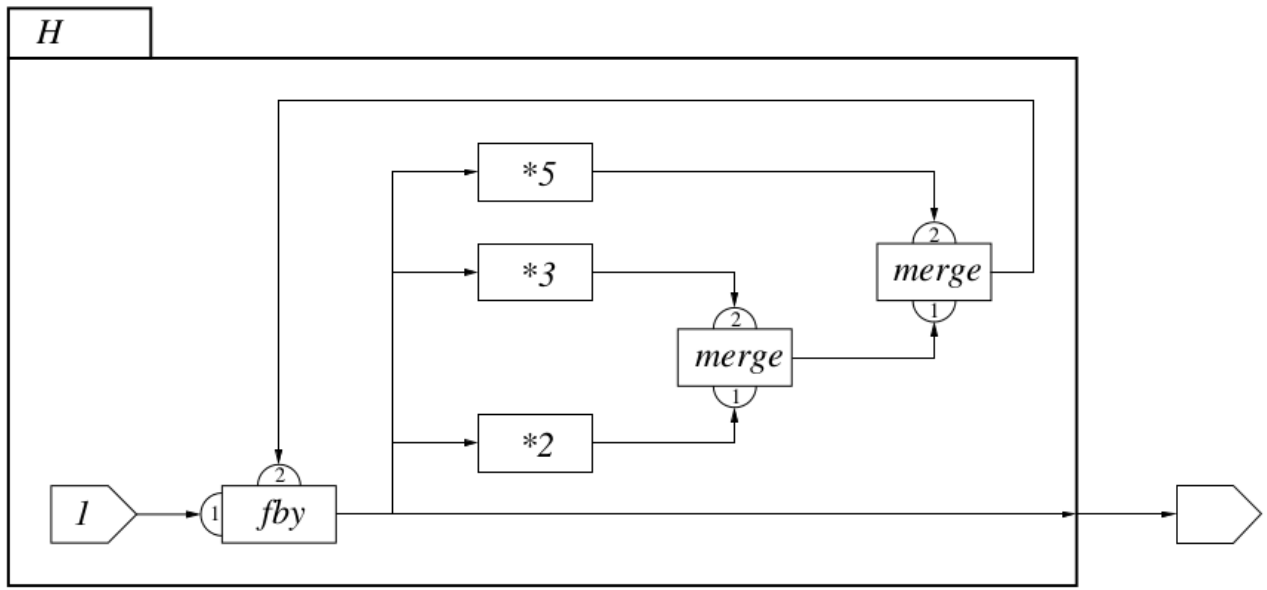


Figure from Joey Paquet's PhD thesis, "Intensional Scientific Programming" (1999)

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 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 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, Clustera, 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

```
result  =  f 3 + f 5
f x     =  g (x*x)
g y     =  y+2
```

Intensional transformation by example

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

Example

```
result  =  f 3 + f 5
f x     =  g (x*x)
g y     =  y+2
```

Step 1: for all functions f

- Replace the i-th call of f by $\texttt{call}_i(f)$
- 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

```
result  =  f 3 + f 5
f x     =  g (x*x)
g y     =  y+2
```

```
result  =  call0(f)+call1(f)
f       =  call0(g)
g       =  y+2
```

Step 1: for all functions f

- Replace the i-th call of f by $\texttt{call}_i(f)$
- 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

```
result  =  f 3 + f 5
f x     =  g (x*x)
g y     =  y+2
```

```
result  =  call0(f)+call1(f)
f       =  call0(g)
g       =  y+2
```

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

```
result  =  f 3 + f 5
f x     =  g (x*x)
g y     =  y+2
```

```
result  =  call0(f)+call1(f)
f       =  call0(g)
g       =  y+2
x       =  actuals(3,5)
y       =  actuals(x*x)
```

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

Semantics of the target language

Evaluation of expressions: $\mathit{EVAL}(e, w)$

- **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: $\mathit{EVAL}(e, w)$

- **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`

$$\begin{aligned} \mathit{EVAL}(\texttt{call}_i(e), w) &= \mathit{EVAL}(e, i : w) \\ \mathit{EVAL}(\texttt{actuals}(e_0, \ldots, e_{n-1}), i : w) &= \mathit{EVAL}(e_i, w) \end{aligned}$$

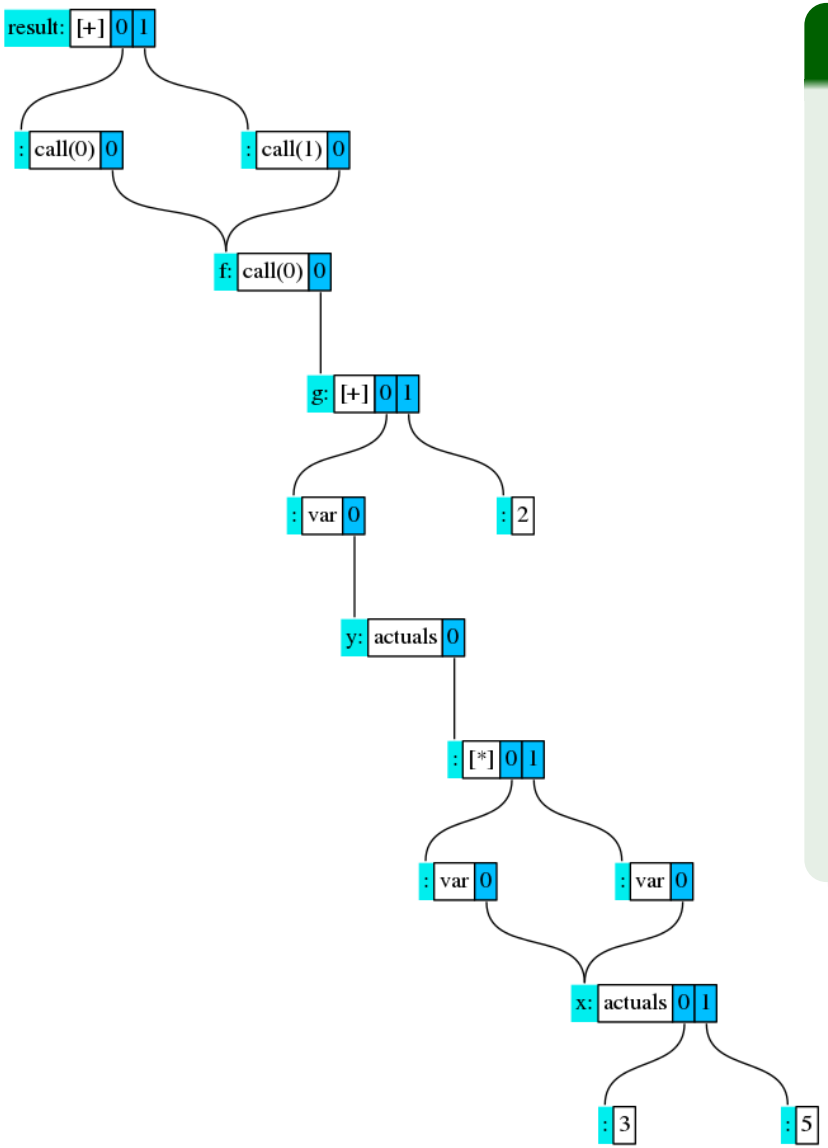
Example

Evaluation of the target program:

$$
\begin{array}{rl}
 & \mathit{EVAL}(\texttt{result}, [\,]) \\
= & \mathit{EVAL}(\texttt{call}_0\texttt{(f)+ call}_1\texttt{(f)}, [\,]) \\
= & \mathit{EVAL}(\texttt{call}_0\texttt{(f)}, [\,]) + \mathit{EVAL}(\texttt{call}_1\texttt{(f)}, [\,]) \\
= & \mathit{EVAL}(\texttt{f}, [0]) + \mathit{EVAL}(\texttt{f}, [1]) \\
= & \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [0]) + \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [1]) \\
= & \mathit{EVAL}(\texttt{g}, [0,0]) + \mathit{EVAL}(\texttt{g}, [0,1]) \\
= & \mathit{EVAL}(\texttt{y}, [0,0]) + \mathit{EVAL}(2, [0,0]) + \mathit{EVAL}(\texttt{y}, [0,1]) + \mathit{EVAL}(2, [0,1]) \\
= & \mathit{EVAL}(\texttt{actuals(x*x)}, [0,0]) + 2 + \mathit{EVAL}(\texttt{actuals(x*x)}, [0,1]) + 2 \\
= & \mathit{EVAL}(\texttt{x*x}, [0]) + 2 + \mathit{EVAL}(\texttt{x*x}, [1]) + 2 \\
= & \mathit{EVAL}(\texttt{x}, [0]) * \mathit{EVAL}(\texttt{x}, [0]) + 2 + \mathit{EVAL}(\texttt{x}, [1]) * \mathit{EVAL}(\texttt{x}, [1]) + 2 \\
= & \mathit{EVAL}(\texttt{actuals(3, 5)}, [0]) * \mathit{EVAL}(\texttt{actuals(3, 5)}, [0]) + 2 + \\
 & \mathit{EVAL}(\texttt{actuals(3, 5)}, [1]) * \mathit{EVAL}(\texttt{actuals(3, 5)}, [1]) + 2 \\
= & \mathit{EVAL}(3, [\,]) * \mathit{EVAL}(3, [\,]) + 2 + \mathit{EVAL}(5, [\,]) * \mathit{EVAL}(5, [\,]) + 2 \\
= & 3 * 3 + 2 + 5 * 5 + 2 \\
= & 38
\end{array}
$$

```
result = call0(f)+call1(f)
f      = call0(g)
g      = y+2
x      = actuals(3, 5)
y      = actuals(x*x)
```

Dataflow graph



Example

```
result  =  f 3 + f 5
f x     =  g (x*x)
g y     =  y+2
```

$$
\begin{aligned}
\texttt{result} &= \texttt{call}_0\texttt{(f)+call}_1\texttt{(f)} \\
\texttt{f} &= \texttt{call}_0\texttt{(g)} \\
\texttt{g} &= \texttt{y+2} \\
\texttt{x} &= \texttt{actuals(3, 5)} \\
\texttt{y} &= \texttt{actuals(x*x)}
\end{aligned}
$$

Higher-order functions: Multiple dimensions

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 $\texttt{call}^i$ and $\texttt{actuals}^i$ operators
- Transformation works in steps, each time decreasing the program's order by one

Example (A second order program that can be transformed)

```
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:

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

Missing pieces

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
```

Missing pieces

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
```

👍 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

The anatomy of a LAR (i)

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 (i)

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**

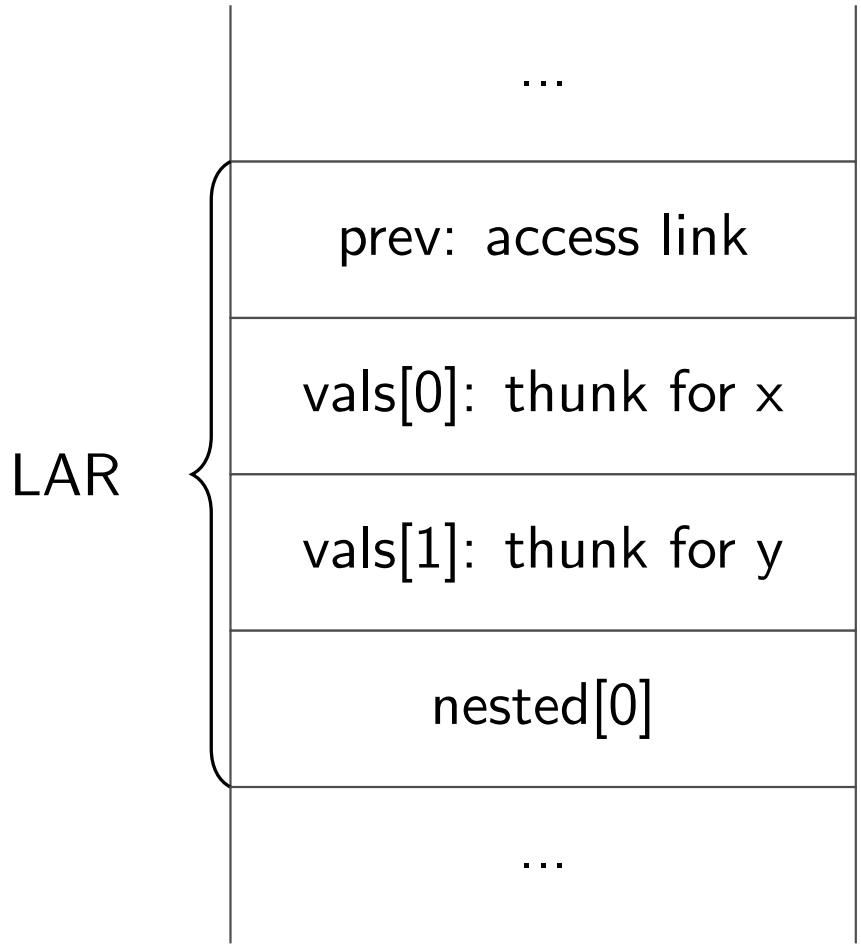
Contents:

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 = case x of [] -> [1]
                  a:as -> [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

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

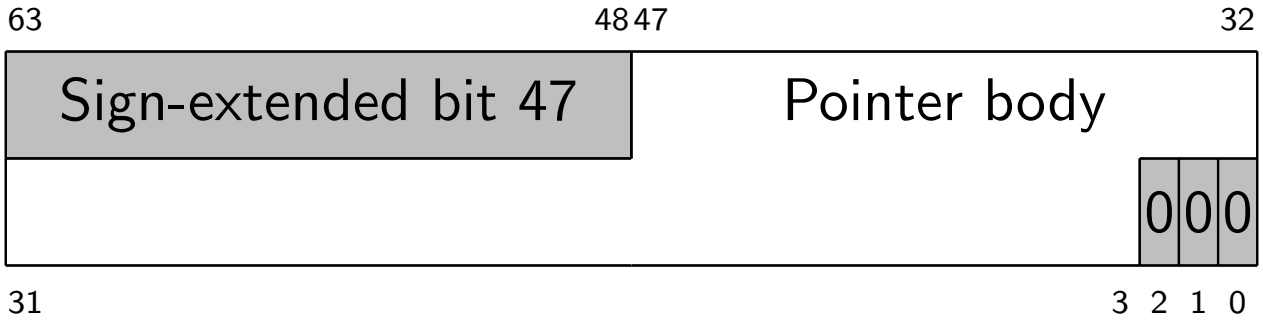
Representation of values:

A value can be:

- a primitive integer n (for enumerations, e.g. `Bool` and `Int`)
- a lazy constructor of the form (c, l)

AMD64 pointers

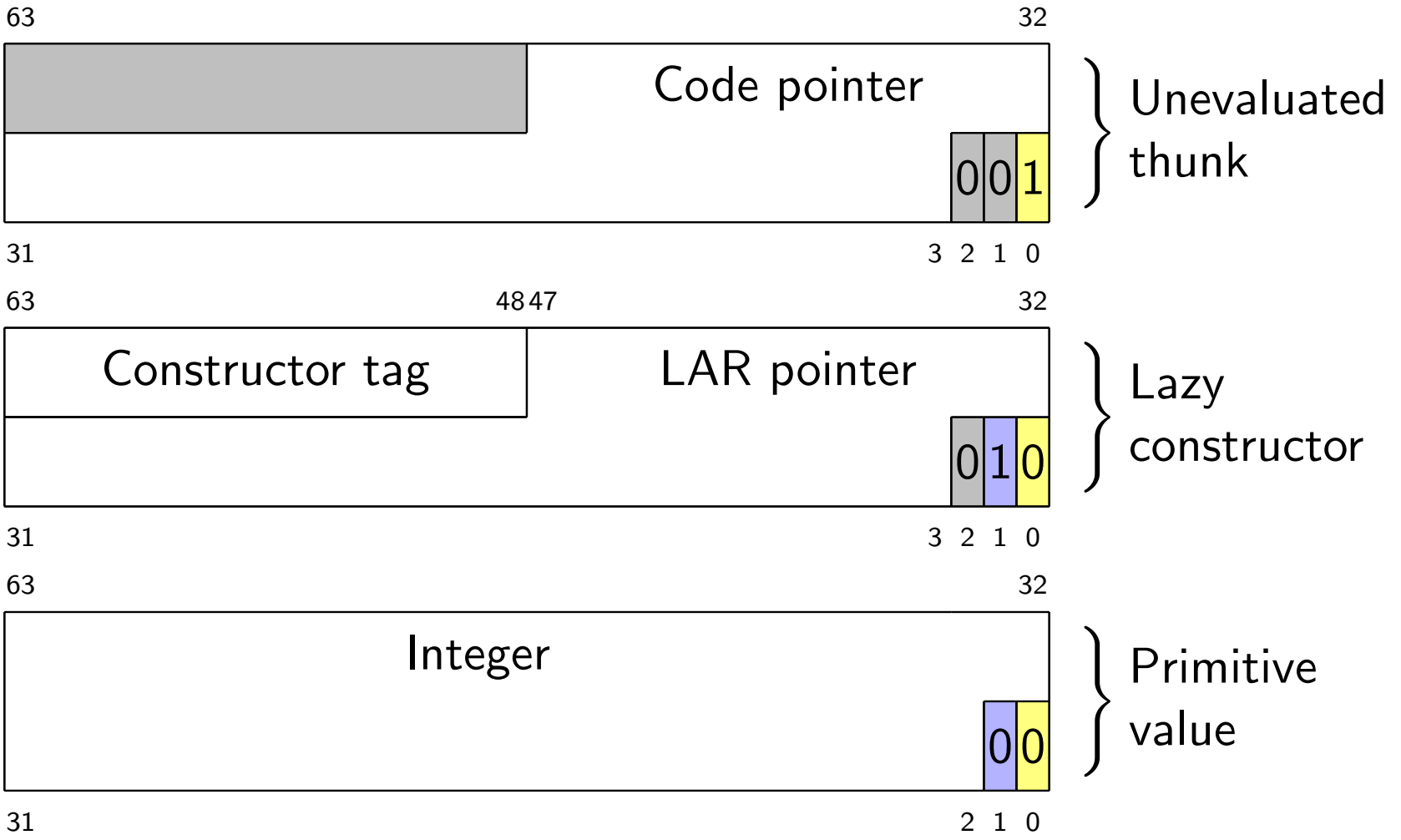
Pointers in AMD64 leave a lot of space unused:



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



Benchmarks (i)

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`

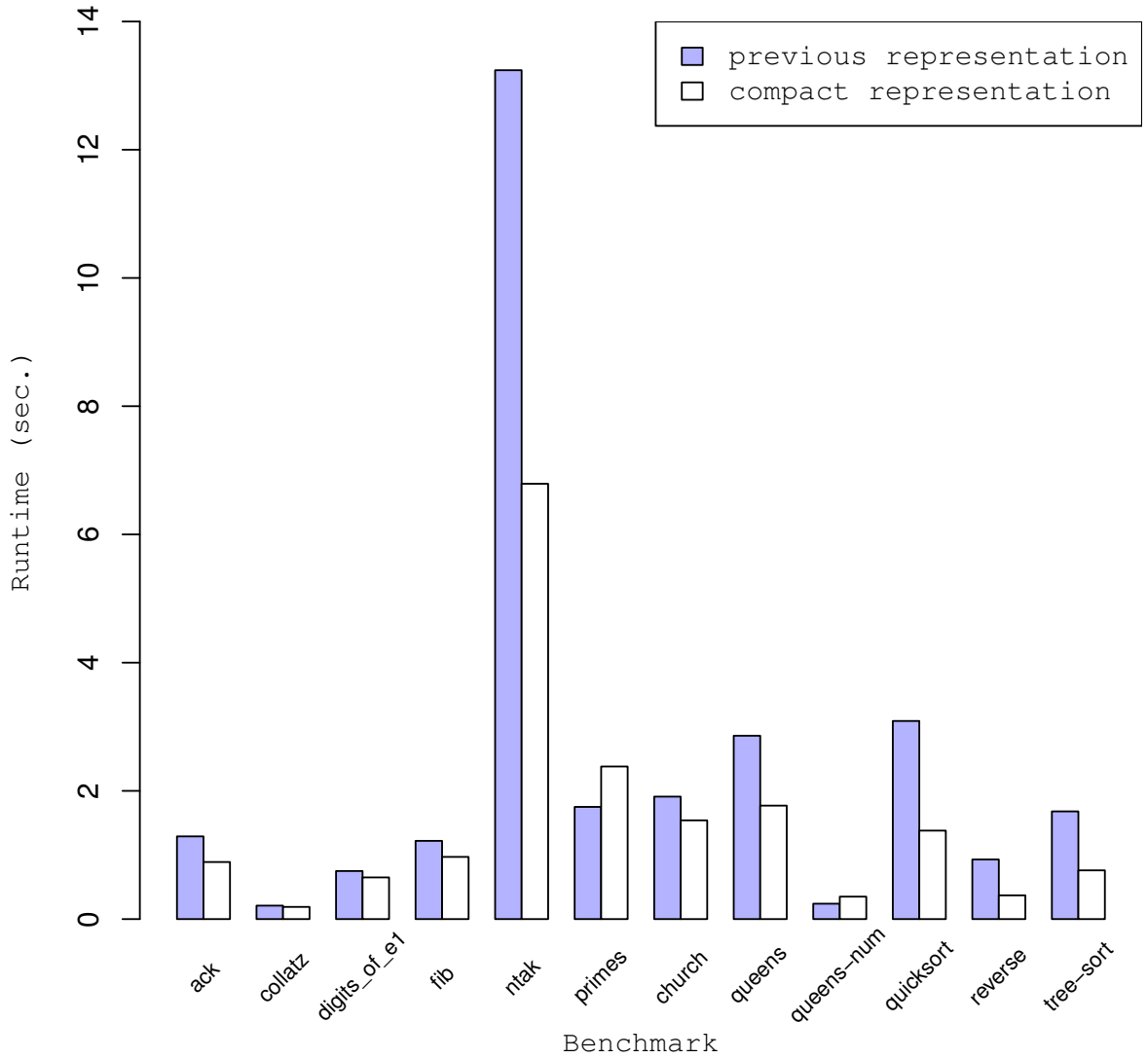
Benchmarks (i)

What we're comparing

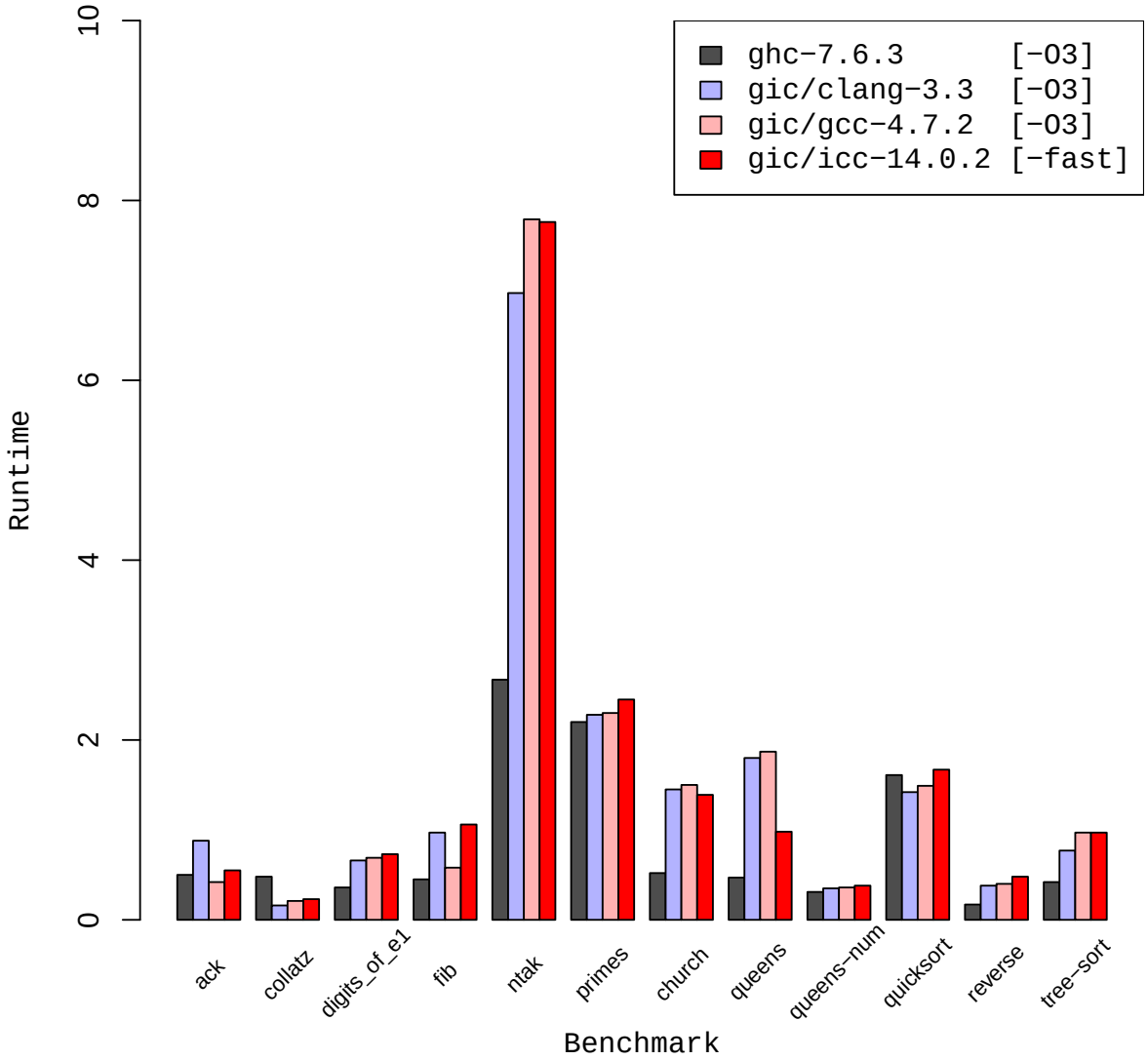
- Comparison with a previous, non-compact representation
- Comparison with fully optimizing **ghc** (-O3)
  - using the default **ghc** back-end
  - using three different **gic** back-ends: `gcc`, `clang` and `icc`

- 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 (ii)



Benchmarks (iii)



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

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 ($\sim$67% reduction)
- The technique can also be used with ARM, SPARC processors

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**

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:

$$\mathit{EVAL}(\texttt{result}, [\,])$$

$$\begin{aligned}
\texttt{result} &= \texttt{call}_0\texttt{(f)+call}_1\texttt{(f)} \\
\texttt{f} &= \texttt{call}_0\texttt{(g)} \\
\texttt{g} &= \texttt{y+2} \\
\texttt{x} &= \texttt{actuals(3, 5)} \\
\texttt{y} &= \texttt{actuals(x*x)}
\end{aligned}$$

Example

Evaluation of the target program:

$$
\begin{aligned}
& \mathit{EVAL}(\texttt{result}, [\,]) \\
= \quad & \mathit{EVAL}(\texttt{call}_0\texttt{(f)+ call}_1\texttt{(f)}, [\,])
\end{aligned}
$$

```
result = call0(f)+call1(f)
f      = call0(g)
g      = y+2
x      = actuals(3,5)
y      = actuals(x*x)
```

Example

Evaluation of the target program:

$$
\begin{aligned}
& \mathit{EVAL}(\texttt{result}, [\,]) \\
= \;& \mathit{EVAL}(\texttt{call}_0\texttt{(f)+ call}_1\texttt{(f)}, [\,]) \\
= \;& \mathit{EVAL}(\texttt{call}_0\texttt{(f)}, [\,]) + \mathit{EVAL}(\texttt{call}_1\texttt{(f)}, [\,])
\end{aligned}
$$

```
result = call0(f)+call1(f)
f      = call0(g)
g      = y+2
x      = actuals(3, 5)
y      = actuals(x*x)
```

Example

Evaluation of the target program:

$$
\begin{aligned}
& \mathit{EVAL}(\texttt{result}, [\,]) \\
= \;& \mathit{EVAL}(\texttt{call}_0\texttt{(f)+ call}_1\texttt{(f)}, [\,]) \\
= \;& \mathit{EVAL}(\texttt{call}_0\texttt{(f)}, [\,]) + \mathit{EVAL}(\texttt{call}_1\texttt{(f)}, [\,]) \\
= \;& \mathit{EVAL}(\texttt{f}, [0]) + \mathit{EVAL}(\texttt{f}, [1])
\end{aligned}
$$

```
result = call0(f)+call1(f)
f      = call0(g)
g      = y+2
x      = actuals(3,5)
y      = actuals(x*x)
```

Example

Evaluation of the target program:

$$
\begin{aligned}
& \mathit{EVAL}(\texttt{result}, [\,]) \\
= \;& \mathit{EVAL}(\texttt{call}_0\texttt{(f)+ call}_1\texttt{(f)}, [\,]) \\
= \;& \mathit{EVAL}(\texttt{call}_0\texttt{(f)}, [\,]) + \mathit{EVAL}(\texttt{call}_1\texttt{(f)}, [\,]) \\
= \;& \mathit{EVAL}(\texttt{f}, [0]) + \mathit{EVAL}(\texttt{f}, [1]) \\
= \;& \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [0]) + \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [1])
\end{aligned}
$$

```
result = call0(f)+call1(f)
f      = call0(g)
g      = y+2
x      = actuals(3,5)
y      = actuals(x*x)
```

Example

Evaluation of the target program:

$$
\begin{aligned}
& \mathit{EVAL}(\texttt{result}, [\,]) \\
= \;& \mathit{EVAL}(\texttt{call}_0\texttt{(f)+ call}_1\texttt{(f)}, [\,]) \\
= \;& \mathit{EVAL}(\texttt{call}_0\texttt{(f)}, [\,]) + \mathit{EVAL}(\texttt{call}_1\texttt{(f)}, [\,]) \\
= \;& \mathit{EVAL}(\texttt{f}, [0]) + \mathit{EVAL}(\texttt{f}, [1]) \\
= \;& \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [0]) + \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [1]) \\
= \;& \mathit{EVAL}(\texttt{g}, [0,0]) + \mathit{EVAL}(\texttt{g}, [0,1])
\end{aligned}
$$

```
result = call0(f)+call1(f)
f      = call0(g)
g      = y+2
x      = actuals(3,5)
y      = actuals(x*x)
```

Example

Evaluation of the target program:

$$
\begin{aligned}
& \mathit{EVAL}(\texttt{result}, [\,]) \\
= \;& \mathit{EVAL}(\texttt{call}_0\texttt{(f)+ call}_1\texttt{(f)}, [\,]) \\
= \;& \mathit{EVAL}(\texttt{call}_0\texttt{(f)}, [\,]) + \mathit{EVAL}(\texttt{call}_1\texttt{(f)}, [\,]) \\
= \;& \mathit{EVAL}(\texttt{f}, [0]) + \mathit{EVAL}(\texttt{f}, [1]) \\
= \;& \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [0]) + \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [1]) \\
= \;& \mathit{EVAL}(\texttt{g}, [0,0]) + \mathit{EVAL}(\texttt{g}, [0,1]) \\
= \;& \mathit{EVAL}(\texttt{y}, [0,0]) + \mathit{EVAL}(2, [0,0]) + \mathit{EVAL}(\texttt{y}, [0,1]) + \mathit{EVAL}(2, [0,1])
\end{aligned}
$$

```
result = call0(f)+call1(f)
f      = call0(g)
g      = y+2
x      = actuals(3, 5)
y      = actuals(x*x)
```

Example

Evaluation of the target program:

$$
\begin{aligned}
& \mathit{EVAL}(\texttt{result}, [\,]) \\
= \;& \mathit{EVAL}(\texttt{call}_0\texttt{(f)+ call}_1\texttt{(f)}, [\,]) \\
= \;& \mathit{EVAL}(\texttt{call}_0\texttt{(f)}, [\,]) + \mathit{EVAL}(\texttt{call}_1\texttt{(f)}, [\,]) \\
= \;& \mathit{EVAL}(\texttt{f}, [0]) + \mathit{EVAL}(\texttt{f}, [1]) \\
= \;& \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [0]) + \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [1]) \\
= \;& \mathit{EVAL}(\texttt{g}, [0,0]) + \mathit{EVAL}(\texttt{g}, [0,1]) \\
= \;& \mathit{EVAL}(\texttt{y}, [0,0]) + \mathit{EVAL}(2, [0,0]) + \mathit{EVAL}(\texttt{y}, [0,1]) + \mathit{EVAL}(2, [0,1]) \\
= \;& \mathit{EVAL}(\texttt{actuals(x*x)}, [0,0]) + 2 + \mathit{EVAL}(\texttt{actuals(x*x)}, [0,1]) + 2
\end{aligned}
$$

```
result = call0(f)+call1(f)
f      = call0(g)
g      = y+2
x      = actuals(3,5)
y      = actuals(x*x)
```

Example

Evaluation of the target program:

$$
\begin{array}{rl}
 & \mathit{EVAL}(\texttt{result}, [\,]) \\
= & \mathit{EVAL}(\texttt{call}_0\texttt{(f)+ call}_1\texttt{(f)}, [\,]) \\
= & \mathit{EVAL}(\texttt{call}_0\texttt{(f)}, [\,]) + \mathit{EVAL}(\texttt{call}_1\texttt{(f)}, [\,]) \\
= & \mathit{EVAL}(\texttt{f}, [0]) + \mathit{EVAL}(\texttt{f}, [1]) \\
= & \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [0]) + \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [1]) \\
= & \mathit{EVAL}(\texttt{g}, [0,0]) + \mathit{EVAL}(\texttt{g}, [0,1]) \\
= & \mathit{EVAL}(\texttt{y}, [0,0]) + \mathit{EVAL}(2, [0,0]) + \mathit{EVAL}(\texttt{y}, [0,1]) + \mathit{EVAL}(2, [0,1]) \\
= & \mathit{EVAL}(\texttt{actuals(x*x)}, [0,0]) + 2 + \mathit{EVAL}(\texttt{actuals(x*x)}, [0,1]) + 2 \\
= & \mathit{EVAL}(\texttt{x*x}, [0]) + 2 + \mathit{EVAL}(\texttt{x*x}, [1]) + 2
\end{array}
$$

$$
\begin{array}{ll}
\texttt{result} & = \texttt{call}_0\texttt{(f)+call}_1\texttt{(f)} \\
\texttt{f} & = \texttt{call}_0\texttt{(g)} \\
\texttt{g} & = \texttt{y+2} \\
\texttt{x} & = \texttt{actuals(3, 5)} \\
\texttt{y} & = \texttt{actuals(x*x)}
\end{array}
$$

Example

Evaluation of the target program:

$$
\begin{array}{rl}
 & \mathit{EVAL}(\texttt{result}, [\,]) \\
= & \mathit{EVAL}(\texttt{call}_0\texttt{(f)+ call}_1\texttt{(f)}, [\,]) \\
= & \mathit{EVAL}(\texttt{call}_0\texttt{(f)}, [\,]) + \mathit{EVAL}(\texttt{call}_1\texttt{(f)}, [\,]) \\
= & \mathit{EVAL}(\texttt{f}, [0]) + \mathit{EVAL}(\texttt{f}, [1]) \\
= & \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [0]) + \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [1]) \\
= & \mathit{EVAL}(\texttt{g}, [0,0]) + \mathit{EVAL}(\texttt{g}, [0,1]) \\
= & \mathit{EVAL}(\texttt{y}, [0,0]) + \mathit{EVAL}(2, [0,0]) + \mathit{EVAL}(\texttt{y}, [0,1]) + \mathit{EVAL}(2, [0,1]) \\
= & \mathit{EVAL}(\texttt{actuals(x*x)}, [0,0]) + 2 + \mathit{EVAL}(\texttt{actuals(x*x)}, [0,1]) + 2 \\
= & \mathit{EVAL}(\texttt{x*x}, [0]) + 2 + \mathit{EVAL}(\texttt{x*x}, [1]) + 2 \\
= & \mathit{EVAL}(\texttt{x}, [0]) * \mathit{EVAL}(\texttt{x}, [0]) + 2 + \mathit{EVAL}(\texttt{x}, [1]) * \mathit{EVAL}(\texttt{x}, [1]) + 2
\end{array}
$$

```
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:

$$
\begin{array}{rl}
 & \mathit{EVAL}(\texttt{result}, [\,]) \\
= & \mathit{EVAL}(\texttt{call}_0\texttt{(f)+ call}_1\texttt{(f)}, [\,]) \\
= & \mathit{EVAL}(\texttt{call}_0\texttt{(f)}, [\,]) + \mathit{EVAL}(\texttt{call}_1\texttt{(f)}, [\,]) \\
= & \mathit{EVAL}(\texttt{f}, [0]) + \mathit{EVAL}(\texttt{f}, [1]) \\
= & \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [0]) + \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [1]) \\
= & \mathit{EVAL}(\texttt{g}, [0,0]) + \mathit{EVAL}(\texttt{g}, [0,1]) \\
= & \mathit{EVAL}(\texttt{y}, [0,0]) + \mathit{EVAL}(2, [0,0]) + \mathit{EVAL}(\texttt{y}, [0,1]) + \mathit{EVAL}(2, [0,1]) \\
= & \mathit{EVAL}(\texttt{actuals(x*x)}, [0,0]) + 2 + \mathit{EVAL}(\texttt{actuals(x*x)}, [0,1]) + 2 \\
= & \mathit{EVAL}(\texttt{x*x}, [0]) + 2 + \mathit{EVAL}(\texttt{x*x}, [1]) + 2 \\
= & \mathit{EVAL}(\texttt{x}, [0]) * \mathit{EVAL}(\texttt{x}, [0]) + 2 + \mathit{EVAL}(\texttt{x}, [1]) * \mathit{EVAL}(\texttt{x}, [1]) + 2 \\
= & \mathit{EVAL}(\texttt{actuals(3, 5)}, [0]) * \mathit{EVAL}(\texttt{actuals(3, 5)}, [0]) + 2 + \\
 & \mathit{EVAL}(\texttt{actuals(3, 5)}, [1]) * \mathit{EVAL}(\texttt{actuals(3, 5)}, [1]) + 2
\end{array}
$$

```
result = call0(f)+call1(f)
f      = call0(g)
g      = y+2
x      = actuals(3, 5)
y      = actuals(x*x)
```

Example

Evaluation of the target program:

$$
\begin{aligned}
& \mathit{EVAL}(\texttt{result}, [\,]) \\
= \;& \mathit{EVAL}(\texttt{call}_0\texttt{(f)+ call}_1\texttt{(f)}, [\,]) \\
= \;& \mathit{EVAL}(\texttt{call}_0\texttt{(f)}, [\,]) + \mathit{EVAL}(\texttt{call}_1\texttt{(f)}, [\,]) \\
= \;& \mathit{EVAL}(\texttt{f}, [0]) + \mathit{EVAL}(\texttt{f}, [1]) \\
= \;& \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [0]) + \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [1]) \\
= \;& \mathit{EVAL}(\texttt{g}, [0,0]) + \mathit{EVAL}(\texttt{g}, [0,1]) \\
= \;& \mathit{EVAL}(\texttt{y}, [0,0]) + \mathit{EVAL}(2, [0,0]) + \mathit{EVAL}(\texttt{y}, [0,1]) + \mathit{EVAL}(2, [0,1]) \\
= \;& \mathit{EVAL}(\texttt{actuals(x*x)}, [0,0]) + 2 + \mathit{EVAL}(\texttt{actuals(x*x)}, [0,1]) + 2 \\
= \;& \mathit{EVAL}(\texttt{x*x}, [0]) + 2 + \mathit{EVAL}(\texttt{x*x}, [1]) + 2 \\
= \;& \mathit{EVAL}(\texttt{x}, [0]) * \mathit{EVAL}(\texttt{x}, [0]) + 2 + \mathit{EVAL}(\texttt{x}, [1]) * \mathit{EVAL}(\texttt{x}, [1]) + 2 \\
= \;& \mathit{EVAL}(\texttt{actuals(3, 5)}, [0]) * \mathit{EVAL}(\texttt{actuals(3, 5)}, [0]) + 2 + \\
& \mathit{EVAL}(\texttt{actuals(3, 5)}, [1]) * \mathit{EVAL}(\texttt{actuals(3, 5)}, [1]) + 2 \\
= \;& \mathit{EVAL}(3, [\,]) * \mathit{EVAL}(3, [\,]) + 2 + \mathit{EVAL}(5, [\,]) * \mathit{EVAL}(5, [\,]) + 2
\end{aligned}
$$

```
result = call0(f)+call1(f)
f      = call0(g)
g      = y+2
x      = actuals(3, 5)
y      = actuals(x*x)
```

Example

```
result = call0(f)+call1(f)
f      = call0(g)
g      = y+2
x      = actuals(3, 5)
y      = actuals(x*x)
```

Evaluation of the target program:

$$
\begin{aligned}
& \mathit{EVAL}(\texttt{result}, [\,]) \\
= \;& \mathit{EVAL}(\texttt{call}_0\texttt{(f)+ call}_1\texttt{(f)}, [\,]) \\
= \;& \mathit{EVAL}(\texttt{call}_0\texttt{(f)}, [\,]) + \mathit{EVAL}(\texttt{call}_1\texttt{(f)}, [\,]) \\
= \;& \mathit{EVAL}(\texttt{f}, [0]) + \mathit{EVAL}(\texttt{f}, [1]) \\
= \;& \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [0]) + \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [1]) \\
= \;& \mathit{EVAL}(\texttt{g}, [0,0]) + \mathit{EVAL}(\texttt{g}, [0,1]) \\
= \;& \mathit{EVAL}(\texttt{y}, [0,0]) + \mathit{EVAL}(2, [0,0]) + \mathit{EVAL}(\texttt{y}, [0,1]) + \mathit{EVAL}(2, [0,1]) \\
= \;& \mathit{EVAL}(\texttt{actuals(x*x)}, [0,0]) + 2 + \mathit{EVAL}(\texttt{actuals(x*x)}, [0,1]) + 2 \\
= \;& \mathit{EVAL}(\texttt{x*x}, [0]) + 2 + \mathit{EVAL}(\texttt{x*x}, [1]) + 2 \\
= \;& \mathit{EVAL}(\texttt{x}, [0]) * \mathit{EVAL}(\texttt{x}, [0]) + 2 + \mathit{EVAL}(\texttt{x}, [1]) * \mathit{EVAL}(\texttt{x}, [1]) + 2 \\
= \;& \mathit{EVAL}(\texttt{actuals(3, 5)}, [0]) * \mathit{EVAL}(\texttt{actuals(3, 5)}, [0]) + 2 + \\
& \mathit{EVAL}(\texttt{actuals(3, 5)}, [1]) * \mathit{EVAL}(\texttt{actuals(3, 5)}, [1]) + 2 \\
= \;& \mathit{EVAL}(3, [\,]) * \mathit{EVAL}(3, [\,]) + 2 + \mathit{EVAL}(5, [\,]) * \mathit{EVAL}(5, [\,]) + 2 \\
= \;& 3 * 3 + 2 + 5 * 5 + 2
\end{aligned}
$$

Example

```
result = call0(f)+call1(f)
f      = call0(g)
g      = y+2
x      = actuals(3, 5)
y      = actuals(x*x)
```

Evaluation of the target program:

$$
\begin{aligned}
& \mathit{EVAL}(\texttt{result}, [\,]) \\
= \;& \mathit{EVAL}(\texttt{call}_0\texttt{(f)+ call}_1\texttt{(f)}, [\,]) \\
= \;& \mathit{EVAL}(\texttt{call}_0\texttt{(f)}, [\,]) + \mathit{EVAL}(\texttt{call}_1\texttt{(f)}, [\,]) \\
= \;& \mathit{EVAL}(\texttt{f}, [0]) + \mathit{EVAL}(\texttt{f}, [1]) \\
= \;& \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [0]) + \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [1]) \\
= \;& \mathit{EVAL}(\texttt{g}, [0,0]) + \mathit{EVAL}(\texttt{g}, [0,1]) \\
= \;& \mathit{EVAL}(\texttt{y}, [0,0]) + \mathit{EVAL}(2, [0,0]) + \mathit{EVAL}(\texttt{y}, [0,1]) + \mathit{EVAL}(2, [0,1]) \\
= \;& \mathit{EVAL}(\texttt{actuals(x*x)}, [0,0]) + 2 + \mathit{EVAL}(\texttt{actuals(x*x)}, [0,1]) + 2 \\
= \;& \mathit{EVAL}(\texttt{x*x}, [0]) + 2 + \mathit{EVAL}(\texttt{x*x}, [1]) + 2 \\
= \;& \mathit{EVAL}(\texttt{x}, [0]) * \mathit{EVAL}(\texttt{x}, [0]) + 2 + \mathit{EVAL}(\texttt{x}, [1]) * \mathit{EVAL}(\texttt{x}, [1]) + 2 \\
= \;& \mathit{EVAL}(\texttt{actuals(3, 5)}, [0]) * \mathit{EVAL}(\texttt{actuals(3, 5)}, [0]) + 2 + \\
& \mathit{EVAL}(\texttt{actuals(3, 5)}, [1]) * \mathit{EVAL}(\texttt{actuals(3, 5)}, [1]) + 2 \\
= \;& \mathit{EVAL}(3, [\,]) * \mathit{EVAL}(3, [\,]) + 2 + \mathit{EVAL}(5, [\,]) * \mathit{EVAL}(5, [\,]) + 2 \\
= \;& 3 * 3 + 2 + 5 * 5 + 2 \\
= \;& 38
\end{aligned}
$$

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