Counting Immutable Beans
Reference Counting Optimized for Purely Functional Programming

Sebastian Ullrich - KIT - Germany
Leonardo de Moura - MSR - USA
• Pure functional language; Strict; Dependent types
• Meta programming: extend Lean using Lean
• Applications:
  • Formal Abstracts Project - Tom Hales
  • Perfectoid Spaces Project
    Kevin Buzzard, Johan Commelin, and Patrick Massot
  • Education (CMU, Imperial College, …)
  • Lean Forward - Jasmin Blanchette
  • Protocol Verification (Galois)
  • SQL query equivalence (UW)
  • IMO Grand Challenge (MSR)
  • AliveInLean (MSR)
• 6 papers at ITP 2019
an extensible compiler

Lean3 users write metaprograms/tactics in Lean
Examples: ring solver, conductive predicates, superposition prover, transfer tactic, …

We are implementing Lean4 in Lean itself.
All subsystems can be extended: parser, elaborator, compiler, …

New compiler is already outperforming Haskell and OCaml.

Proofs for performance and profit.
A better value proposition: use proofs for obtaining more efficient code.
The return of reference counting

• Most compilers for functional languages (OCaml, GHC, …) use tracing GC
• RC is simple to implement.
• Easy to support multi-threading programs.
• Destructive updates when reference count = 1.
  • It is a known optimization for big objects (e.g., arrays).
    Array.set : Array a -> Index -> a -> Array a
  • We demonstrate it is also relevant for small objects.
• In languages like Coq and Lean, we do not have cycles.
• Easy to interface with C, C++ and Rust.
Resurrection hypothesis

Many objects die just before the creation of an object of the same kind.

Examples:

• List.map : List a -> (a -> b) -> List b

• Compiler applies transformations to expressions.

• Proof assistant rewrites/simplifies formulas.

• Updates to functional data structures such as red black trees.

• List zipper

\[
\text{goForward} ([], bs) = ([], bs) \\
\text{goForward} (x : xs, bs) = (xs, x : bs)
\]
Contributions

- Approach for reusing memory: **small** and big values. Big values are often nested into small ones.
- Inference procedure for borrowed references (à la Swift)
- Simple and efficient scheme for performing atomic RC updates in multi-threaded programs.
- Implementation and experimental evaluation.
- https://github.com/leanprover/lean4
Reference counts

- Each heap-allocated object has a reference count.
- We can view the counter as a collection of tokens.
- The `inc` instruction creates a new token.
- The `dec` instruction consumes a token.
- When a function takes an argument as an `owned` reference, it must consume one of its tokens.
- A function may consume an owned reference by using `dec`, passing it to another function, or storing it in a newly allocated value.
Owned references: examples

\[ id \ x = \text{ret} \ x \]

\[ \text{mkPairOf} \ x = \text{inc} \ x; \text{let} \ p = \text{Pair} \ x \ x; \text{ret} \ p \]

\[ \text{fst} \ x \ y = \text{dec} \ y; \text{ret} \ x \]
Borrowed references

• If $xs$ is an owned reference

\[ isNil \, xs = \text{case} \, xs \, \text{of} \]

\[ (\text{Nil} \rightarrow \text{dec} \, xs; \text{ret} \, \text{true}) \]

\[ (\text{Cons} \rightarrow \text{dec} \, xs; \text{ret} \, \text{false}) \]

• If $xs$ is a borrowed reference

\[ isNil \, xs = \text{case} \, xs \, \text{of} \, (\text{Nil} \rightarrow \text{ret} \, \text{true}) \, (\text{Cons} \rightarrow \text{ret} \, \text{false}) \]
Owned vs Borrowed

• Transformers and constructors own references.

• Inspectors and visitors borrow references.

• Remark: it is not safe to destructively update borrowed references even when RC = 1
Reusing small objects

\[
\begin{align*}
map f \ [\ ] &= \ [\ ] \\
map f \ (x : xs) &= (f \ x) : (\map f \ xs)
\end{align*}
\]

First attempt

\[
\begin{align*}
\map f \ xs &= \case \ xs \ of \\
&\quad (\ret \ xs) \\
&\quad (\let \ x = \proj_1 \ xs; \inc \ x; \let \ s = \proj_2 \ xs; \inc \ s; \\
&\quad \let \ y = f \ x; \let \ ys = \map f \ s; \\
&\quad \let \ r = (\reuse \ xs \ inctor_2 \ y \ ys); \ret \ r)
\end{align*}
\]
Reusing small objects

\[ map \ f \ xs = \text{case} \ xs \ of \]
\[
\quad (\text{ret} \ xs)
\quad (\text{let} \ x = \text{proj}_1 \ xs; \ \text{inc} \ x; \ \text{let} \ s = \text{proj}_2 \ xs; \ \text{inc} \ s; \]
\quad \quad \text{let} \ y = f \ x; \ \text{let} \ ys = map \ f \ s;
\quad \quad \text{let} \ r = (\text{reuse} \ xs \ \text{inctor}_2 \ y \ ys); \ \text{ret} \ r)
\]

\[ f \rightarrow \text{trim} \]

\[ xs \rightarrow 1 \quad 1 \quad \ldots \rightarrow 1 \quad 1 \quad \ldots \rightarrow \ldots \]

\[ 1 \quad \text{“hello”} \rightarrow 1 \quad \text{“world”} \]
Reusing small objects

\[ \text{map } f \, xs = \text{case } xs \text{ of} \]

\[
(\text{ret } xs) \\
(\text{let } x = \text{proj}_1 \, xs; \text{inc } x; \text{let } s = \text{proj}_2 \, xs; \text{inc } s; \\
\text{let } y = f \, x; \text{let } ys = \text{map } f \, s; \\
\text{let } r = (\text{reuse } xs \, \text{ctor}_2 \, y \, ys); \text{ret } r)
\]

\[ f \rightarrow \text{trim} \]

\[ x \rightarrow \begin{array}{c}
1 \\
2 \quad \text{“hello”} \\
2 \\
1 \quad \text{“world”}
\end{array} \]
Reusing small objects

\[
\text{map } f \text{ } xs = \text{case } xs \text{ of }
\]
\[
(\text{ret } xs)
\]
\[
(\text{let } x = \text{proj}_1 \text{ } xs; \text{inc } x; \text{let } s = \text{proj}_2 \text{ } xs; \text{inc } s; \\
\text{let } y = f \text{ } x; \text{let } ys = \text{map } f \text{ } s; \\
\text{let } r = (\text{reuse } xs \text{ inctor}_2 \text{ } y \text{ } ys); \text{ret } r)
\]

\[
f \rightarrow \text{trim}
\]

\[
x \rightarrow \begin{array}{c}1\text{ “hello” } \end{array} \rightarrow \begin{array}{c}1\text{ “world” } \end{array} \rightarrow \ldots
\]

\[
xs \rightarrow \begin{array}{c}1\text{ “hello” } \end{array} \rightarrow \begin{array}{c}2\text{ “hello” } \end{array} \rightarrow \ldots
\]

\[
s \rightarrow \begin{array}{c}1\text{ “hello” } \end{array}
\]
Reusing small objects

\[
\text{map } f \; xs = \text{case } xs \text{ of } \\
(\text{ret } xs) \\
(\text{let } x = \text{proj}_1 \; xs; \text{inc } x; \text{let } s = \text{proj}_2 \; xs; \text{inc } s; \\
\text{let } y = f \; x; \text{let } ys = \text{map } f \; s; \\
\text{let } r = (\text{reuse } xs \; \text{inctor}_2 \; y \; ys); \text{ret } r)
\]

\[
f \rightarrow \text{trim}
\]

\[
\begin{array}{c}
\text{xs} \\
\downarrow \\
1 \rightarrow \text{“hello”} \\
\downarrow \\
1 \rightarrow \text{“world”} \\
\downarrow \\
1 \rightarrow \text{“world”}
\end{array}
\]
Reusing small objects

\[ \text{map } f \, xs = \text{case } xs \text{ of} \]
\[ \quad (\text{ret } xs) \]
\[ \quad (\text{let } x = \text{proj}_1 \, xs; \text{inc } x; \text{let } s = \text{proj}_2 \, xs; \text{inc } s; \]
\[ \quad \quad \text{let } y = f \, x; \text{let } ys = \text{map } f \, s; \]
\[ \quad \quad \text{let } r = (\text{reuse } xs \text{ inctor}_2 \, y \, ys); \text{ret } r) \]

BAD. We only reused the one memory cell. We can do better!
Reusing small objects

\[
\begin{align*}
map f \; [] &= [] \\
map f \; (x : xs) &= (f \; x) : (map f \; xs)
\end{align*}
\]

\text{Second attempt}

\[
\begin{align*}
map f \; xs &= \text{case} \; xs \, \text{of} \\
&\quad (\text{ret} \; xs) \\
&\quad (\text{let} \; x = \text{proj}_1 \; xs; \text{inc} \; x; \text{let} \; s = \text{proj}_2 \; xs; \text{inc} \; s; \\
&\quad \quad \text{let} \; w = \text{reset} \; xs; \\
&\quad \quad \text{let} \; y = f \; x; \text{let} \; ys = map f \; s; \\
&\quad \quad \text{let} \; r = (\text{reuse} \; w \; \text{in} \; \text{ctor}_2 \; y \; ys); \text{ret} \; r)
\end{align*}
\]
Reusing small objects

\[
\text{map } f \text{ xs} = \text{case } xs \text{ of }
\]
\[
\text{(ret } xs) \\
\text{(let } x = \text{proj}_1 \text{ xs}; \text{inc } x; \text{let } s = \text{proj}_2 \text{ xs}; \text{inc } s; \\
\text{let } w = \text{reset } xs; \\
\text{let } y = f x; \text{let } ys = \text{map } f \text{ s}; \\
\text{let } r = (\text{reuse } w \text{ in } \text{ctor}_2 \text{ y ys}; \text{ret } r)
\]

\[f \rightarrow \text{trim}\]

\[xs \rightarrow 1 \rightarrow 1 \rightarrow \ldots \]

\[1 \quad \text{“hello”} \quad 1 \quad \text{“world”} \]
Reusing small objects

\[
\text{map } f \, xs = \text{case } xs \text{ of } \\
(\text{ret } xs) \\
(\text{let } x = \text{proj}_1 \, xs; \text{inc } x; \text{let } s = \text{proj}_2 \, xs; \text{inc } s; \\
\text{let } w = \text{reset } xs; \\
\text{let } y = f \, x; \text{let } ys = \text{map } f \, s; \\
\text{let } r = (\text{reuse } w \text{ in ctor}_2 \, y \, ys); \text{ret } r)
\]

\[f \rightarrow \text{trim}\]

\[
\begin{array}{c}
\text{xs} \\
1 \\
\text{2 “hello”} \\
\text{1 “world”}
\end{array}
\]
Reusing small objects

\[ \text{map } f \text{ xs} = \text{case } \text{xs of} \]

\[
\begin{align*}
(\text{let } x &= \text{proj}_1 \text{ xs}; \text{inc } x; \text{let } s &= \text{proj}_2 \text{ xs}; \text{inc } s; \\
&\quad \text{let } w = \text{reset } \text{xs}; \\
&\quad \text{let } y = f x; \text{let } ys = \text{map } f s; \\
&\quad \text{let } r = (\text{reuse } w \text { in } \text{ctor}_2 \ y \ ys); \text{ret } r)
\end{align*}
\]

\[ f \longrightarrow \text{trim} \]

\[ s \]

\[ \text{xs} \]
Reusing small objects

\[
\text{map } f \text{ } xs = \text{ case } xs \text{ of }
\]
\[
(\text{ret } xs)
\]
\[
(\text{let } x = \text{proj}_1 \text{ } xs; \text{inc } x; \text{let } s = \text{proj}_2 \text{ } xs; \text{inc } s;
\]
\[
\text{let } w = \text{reset } xs;
\]
\[
\text{let } y = f x; \text{let } ys = \text{map } f \text{ } s;
\]
\[
\text{let } r = (\text{reuse } w \text{ in } \text{ctor}_2 \text{ } y \text{ } ys); \text{ret } r)
\]
Reusing small objects

\[
\text{map } f \; xs = \text{ case } xs \text{ of }
\]
\[
\quad \text{(ret } xs) \\
\quad \text{(let } x = \text{proj}_1 \; xs; \text{inc } x; \text{let } s = \text{proj}_2 \; xs; \text{inc } s; \\
\qquad \text{let } w = \text{reset } xs; \\
\qquad \text{let } y = f \; x; \text{let } ys = \text{map } f \; s; \\
\qquad \text{let } r = (\text{reuse } w \text{ in ctor}_2 \; y \; ys); \text{ret } r)
\]

\[
f \rightarrow \text{trim}
\]

\[
x \quad \begin{array}{c}
\text{1} \\
\text{w}
\end{array} \quad \begin{array}{c}
\text{1} \\
\text{x}
\end{array} \quad \begin{array}{c}
\text{1} \\
\text{y}
\end{array} \quad \begin{array}{c}
\text{“hello”} \\
\text{“world”}
\end{array} \quad \begin{array}{c}
\text{1} \\
\text{s} \\
\text{ys}
\end{array} \quad \begin{array}{c}
\text{...}
\end{array}
\]
Reusing small objects

\[
map f xs = \text{case } xs \text{ of }
\]
\[
(\text{return } xs)
\]
\[
(let \ x = \text{proj}_1 xs; \text{inc } x; \text{let } s = \text{proj}_2 xs; \text{inc } s; \text{let } w = \text{reset } xs; \text{let } y = f x; \text{let } ys = \text{map } f s; \text{let } r = (\text{reuse } w \text{ in ctor}_2 y ys); \text{return } r)
\]

\(f \rightarrow \text{trim}\)

The whole list was destructively updated!
The compiler

- Lean \( \Rightarrow \) Lambda Pure
- Insert reset/reuse instructions
- Infer borrowed annotations
- Insert inc/dec instructions
- Additional optimizations

\[
\begin{align*}
\text{Var} & \ni w, x, y, z \\
\text{Const} & \ni c \\
\text{Expr} & ::= c \ y \mid \text{pap} \ c \ y \mid x \ y \mid \text{ctor} \ i \ y \mid \text{proj} \ i \ x \\
\text{FnBody} & ::= \text{ret} \ x \mid \text{let} \ x = e; \ F \mid \text{case} \ x \text{ of} \ F \\
\text{Fn} & ::= \lambda \ y. \ F \\
\text{Program} & = \text{Const} \to \text{Fn}
\end{align*}
\]
Inserting reset/reuse

For each (case \(x\) of \(F_1 \ldots F_n\)), for each branch \(F_i\), if \(F_i\) is of form \((P; S; \text{let } y := \text{ctor}_i zs; K)\) where

1. \(#zs\) is equal to the number of fields of \(x\) at branch \(F_i\)
2. \(x\) is dead at \((S; \text{let } y := \text{ctor}_i zs; K)\)

then replace with

\[P; \text{let } w := \text{reset } x; S; \text{let } y := \text{reuse } w \text{ in } \text{ctor}_i zs; K\]

\[
\text{swap } xs = \text{case } xs \text{ of }
\quad (\text{ret } xs) \\
\quad (\text{let } t_1 = \text{proj}_2 xs; \text{case } t_1 \text{ of }
\quad 
\qquad (\text{ret } xs) \\
\qquad (\text{let } h_1 = \text{proj}_1 xs; \\
\qquad \quad \text{let } h_2 = \text{proj}_1 t_1; \text{let } t_2 = \text{proj}_2 t_1; \\
\qquad \quad \text{let } r_1 = \text{ctor}_2 h_1 t_2; \text{let } r_2 = \text{ctor}_2 h_2 r_1; \text{ret } r_2))
\]

\[
\begin{align*}
\text{swap } [] &= [] \\
\text{swap } [x] &= [x] \\
\text{swap } (x: y: zs) &= y: x: zs
\end{align*}
\]
Inserting reset/reuse

For each (case \( x \) of \( F_1 \ldots F_n \)), for each branch \( F_i \), if \( F_i \) is of form

\( (P; S; \text{let } y := \text{ctor}_i zs; K) \) where

1. \#\( zs \) is equal to the number of fields of \( x \) at branch \( F_i \)
2. \( x \) is dead at \((S; \text{let } y := \text{ctor}_i zs; K)\)

then replace with

\( P; \text{let } w := \text{reset } x; S; \text{let } y := \text{reuse } w \text{ in ctor}_i zs; K \)

\[
\begin{align*}
\text{swap } xs &= \text{case } xs \text{ of} \\
&(\text{ret } xs) \\
&(\text{let } t_1 = \text{proj}_2 xs; \text{case } t_1 \text{ of} \\
&(\text{ret } xs) \\
&(\text{let } h_1 = \text{proj}_1 t_1; \text{let } t_2 = \text{proj}_2 t_1; \\
& \text{let } r_1 = \text{ctor}_2 h_1 t_2; \text{let } r_2 = \text{ctor}_2 h_2 r_1; \text{ret } r_2))
\end{align*}
\]

\[
\begin{align*}
\text{swap } xs &= \text{case } xs \text{ of} \\
&(\text{ret } xs) \\
&(\text{let } t_1 = \text{proj}_2 xs; \text{case } t_1 \text{ of} \\
&(\text{ret } xs) \\
&(\text{let } h_1 = \text{proj}_1 t_1; \text{let } t_2 = \text{proj}_2 t_1; \\
& \text{let } w_2 = \text{reset } t_1; \text{let } r_1 = \text{reuse } w_2 \text{ in ctor}_2 h_1 t_2; \\
& \text{let } r_2 = \text{reuse } w_1 \text{ in ctor}_2 h_2 r_1; \text{ret } r_2))
\end{align*}
\]
Inferring borrowed annotations

• Heuristic based on the fact that when we mark a parameter as borrowed
  • We reduce the number of RC operations needed, but we prevent reset/reuse and primitive operations from reusing memory cells.

• We also want to preserve tail calls.

• Our approach: collect variables that must be owned.
  • $x$ or one of its projections is used in a reset.
  • $x$ is passed to a function that takes an owned reference.
  • By marking $x$ as borrowed we destroy a tail call.
Tail call preservation

\[ f \ x = \text{case } x \text{ of} \]
\[ (\text{let } r = \text{proj}_1 \ x; \text{ret } r) \]
\[ (\text{let } y_1 = \text{ctor}_1; \text{let } y_2 = \text{ctor}_1 \ y_1; \text{let } r = f \ y_2 ; \text{ret } r) \]

If we mark \( x \) as borrowed, we do not preserve tail calls

\[ f \ x = \text{case } x \text{ of} \]
\[ (\text{let } r = \text{proj}_1 \ x; \text{inc } r; \text{ret } r) \]
\[ (\text{let } y_1 = \text{ctor}_1; \text{let } y_2 \ \text{ctor}_1 \ y_1; \]
\[ \text{let } r = f \ y_2 ; \text{dec } y_2 ; \text{ret } r) \]
Multi-threading support

\[ Task.mk : (Unit \rightarrow \alpha) \rightarrow Task \alpha \]
\[ Task.bind : Task \alpha \rightarrow (\alpha \rightarrow Task \beta) \rightarrow Task \beta \]
\[ Task.get : Task \alpha \rightarrow \alpha \]

- We store in the object header whether an object is multi-thread or not.
- New objects are not multi-threaded.
- We don’t need memory fences for updating RC if an object is not multi-thread.
- The runtime has a \textit{markMT(o)} primitive.
  \[(Task.mk f) \Rightarrow markMT(f)\]
  \[(Task.bind x f) \Rightarrow markMT(x) \text{ and } markMT(f)\]
Simple Optimizations

- Our compiler expands \texttt{reset} and \texttt{reuse} using lower level instructions: \texttt{isShared }x, \texttt{set }x[i] v, \ldots

- The lower level instructions generate new optimization opportunities for many common IR sequences. Example: \texttt{reset} immediately followed by \texttt{reuse}.

- Minimizes the amount of copying and RC operations.
Comparison with Linear/Uniqueness Types

• Values of types marked as linear/unique can be destructively updated.

• Compiler statically checks whether values are being used linearly or not.

• Pros: no runtime checks; compatible with tracing GCs.

• Cons: awkward to use; complicates a dependent type system even more.

• Big cons: all or nothing. A function $f$ that takes non-shared values most of the time cannot perform destructive updates.
Persistent Arrays

Reusing big and small objects.
Persistent arrays will often be shared.
What about cycles?

- Inductive datatypes in Lean are acyclic.
- We can implement co-inductive datatypes without creating cycles.
- Only unsafe code in Lean can create cycles.
- **Cycles are overrated.**
- What about graphs? How do you represent them in Lean?
  - Use arrays like in Rust.
  - We have destructive updates in Lean.
  - Persistent arrays are also quite fast.
## Experimental evaluation

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Lean</th>
<th>del[%]</th>
<th>Cache Misses [1M/s]</th>
<th>GHC</th>
<th>GC</th>
<th>CM</th>
<th>OCaml</th>
<th>GC</th>
<th>CM</th>
</tr>
</thead>
<tbody>
<tr>
<td>binarytrees</td>
<td>1.0</td>
<td>40</td>
<td>37</td>
<td>3.09</td>
<td>72</td>
<td>120</td>
<td>1.20</td>
<td>N/A</td>
<td>186</td>
</tr>
<tr>
<td>deriv</td>
<td>1.0</td>
<td>24</td>
<td>32</td>
<td>1.89</td>
<td>51</td>
<td>31</td>
<td>1.42</td>
<td>76</td>
<td>59</td>
</tr>
<tr>
<td>const_fold</td>
<td>1.0</td>
<td>11</td>
<td>83</td>
<td>2.23</td>
<td>64</td>
<td>51</td>
<td>4.66</td>
<td>91</td>
<td>107</td>
</tr>
<tr>
<td>qsort</td>
<td>1.0</td>
<td>9</td>
<td>0</td>
<td>1.63</td>
<td>1</td>
<td>0</td>
<td>1.37</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>rbmap</td>
<td>1.0</td>
<td>2</td>
<td>6</td>
<td>2.41</td>
<td>39</td>
<td>23</td>
<td>1.00</td>
<td>31</td>
<td>27</td>
</tr>
<tr>
<td>rbmap_10</td>
<td>1.48</td>
<td>15</td>
<td>34</td>
<td>16.37</td>
<td>88</td>
<td>47</td>
<td>1.93</td>
<td>60</td>
<td>59</td>
</tr>
<tr>
<td>rbmap_1</td>
<td>5.1</td>
<td>27</td>
<td>55</td>
<td>16.41</td>
<td>88</td>
<td>48</td>
<td>9.83</td>
<td>88</td>
<td>89</td>
</tr>
</tbody>
</table>
Conclusion

• It is feasible to implement functional languages using RC.

• We barely scratched the surface of the design space.

• We are implementing Lean4 in Lean.

• Compiler generates C code.

• Compiler source code and all experiments are available online. http://github.com/leanprover/lean4

• We are working on new optimizations.