Reusing Thunks for Recursive Data Structures in Lazy Functional Programs

Yasunao TAKANO (Coma-systems Co., Ltd.)
Hideya IWASAKI (The University of Electro-Communications)
Tomoharu UGAWA (The University of Electro-Communications)
Thunk (promise, suspension)

- A thunk is created to delay the evaluation of an expression
  - A thunk contains the expression and the environment
    (a collection of pairs of bound variables and values)

- The process of evaluating the expression in a thunk is called "forcing"

\[
\begin{align*}
n + 1 & \rightarrow T\{n+1\}|_{n=2} \rightarrow 3 \\
\text{delay} & \quad \text{force}
\end{align*}
\]
Our idea - *Thunk Reuse*

- Lazy evaluation has significant run-time overheads
  - Allocating many thunks (space-consuming task)

- We suppress thunk allocations by reusing the thunk that has been just forced
  - Our target is a thunk at the tail part of cons cell
  - We destructively update the environment of the thunk
The data constructor Cons ":" delays its arguments

\[\text{ints } n = n : \text{ints } (n + 1)\]

\[\text{ints } 1 \Rightarrow 1 : T_1\{\text{ints } (n+1)}\{n=1}\]
Without thunk reuse

- The data constructor Cons ":" delays its arguments

\[ \text{ints } n = n : \text{ints } (n + 1) \]

\[ \text{ints } 1 \Rightarrow 1 : T_1\{\text{ints } (n+1)\}\{n=1\} \]
Without thunk reuse

The data constructor Cons ":" delays its arguments

\[ \text{ints } n = n : \text{ints } (n + 1) \]

\[ \text{ints } 1 \Rightarrow 1 : T_1\{\text{ints } (n+1)\}[n=1] \]
\[ \Rightarrow 1 : 2 : T_2\{\text{ints } (n+1)\}[n=2] \]
Without thunk reuse

- The data constructor Cons "::" delays its arguments
  
  \[
  \text{ints } n = n : \text{ints } (n + 1) \\
  \text{ints } 1 \Rightarrow 1 : T_1\{\text{ints } (n+1)\}{n=1} \Rightarrow 1 : 2 : T_2\{\text{ints } (n+1)\}{n=2}
  \]

  Structures of $T_1$ and $T_2$ are almost the same.
Thunk reuse

\[ \text{ints } n = n : \text{ints } (n+1) \]
\[ \text{ints } 1 \Rightarrow 1 : RT \{ \text{ints } (n+1) \} \{ n=1 \} \]
Thunk reuse

\[
\text{ints } n = n : \text{ints } (n+1)\\
\text{ints } 1 \Rightarrow 1 : RT_1\{\text{ints } (n+1)\}\{n=1\}
\]
Thunk reuse

\[
\text{ints } n = n : \text{ints } (n+1)
\]

\[
\text{ints } 1 \Rightarrow 1 : \text{RT}_1\{\text{ints } (n+1)\}\{n=1\}
\]
Thunk reuse

\[ \text{ints } n = n : \text{ints } (n+1) \]

\[ \text{ints } 1 \Rightarrow 1 : \text{RT}_1\{\text{ints } (n+1)\}[n=1] \]
\[ \Rightarrow 1 : 2 : \text{RT}_1\{\text{ints}(n+1)\}[n=2] \]

- Destructively updates the environment
Thunk reuse

\[\text{ints } n = n : \text{ints } (n+1)\]

\[\text{ints } 1 \Rightarrow 1 : \text{RT}_1\{\text{ints } (n+1)\}[n=1] \Rightarrow 1 : 2 : \text{RT}_1\{\text{ints}(n+1)\}[n=2]\]

Makes \(C_1\) point to \(C_2\)

Destructively updates the environment

C1

C2

RT1

ints \(n+1\)

n=2
Thunk reuse

\[ \text{ints } n = n : \text{ints } (n+1) \]

Suppresses the allocation of a new thunk

Makes C1 point to C2

Destructively updates the environment
Singly referred condition

\[ \text{ints } n = n : \text{ints} \ (n+1) \]

C\(_1\) \quad C\(_2\)
\begin{align*}
1 & \quad \text{Makes C}_1 \text{ point to } \text{C}_2 \\
\end{align*}

RT\(_1\)
\begin{align*}
\text{ints} \ (n+1) \\
\text{n=2} & \quad \text{Destructively updates the environment} \\
\end{align*}

\textbf{Singly referred condition}

RT\(_1\) should be referred to only by the tail part of C\(_2\)
Remembering the reference of $C_1$

\[
\text{ints } n = n : \text{ints } (n+1)
\]

Before forcing $RT_1$, we have to remember the reference of $C_1$, because we are going to destructively update the $C_1$'s tail.
Pattern matching can increase the number of references to a thunk

case (ints 1) of
  x:xs -> .. xs ..
We replace each occurrence of \( \text{xs} \) with \((\text{tail\#} \ \text{xxs})\) to avoid the duplication of references.

\[
\text{case (ints 1) of } \\
\ x:\text{xs} \rightarrow \ldots \text{xs} \ldots \\
\]

\[
\rightarrow \\
\text{case (ints 1) of } \\
\ xxs@(x:_@) \rightarrow \ldots (\text{tail\#} \ \text{xxs}) \ldots \\
\]

Evaluation of \((\text{tail\#} \ \text{xxs})\) leads to forcing \(\text{RT}_1\). 

\((\text{tail\#} \ \text{xxs})\) is almost the same as \((\text{tail} \ \text{xxs})\) except that \((\text{tail\#} \ \text{xxs})\) remembers the address of \(\text{xxs}\).
Implementing our Idea to GHC

Haskell Source

Compiler

Core Language

Transforming pattern-matches

CoreToCore

STG Language

Finding reusable thunks

Generates code for thunk reuse

STG To STG

C--

Target Code

Runtime System

Storage Manager

Garbage Collection
dealing reusable thunks on generational-copying GC

Scheduler

Object Definitions
Adding object type for reusable thunks

Execution Model

Registers

Stacks

Updates Thunks

Reuses Thunks

Target Code
This process resembles updating thunks.
Thunk reuse in GHC execution model

case (ints 1) of
    xxs@(x:_) -> .. (tail# xxs) ..

```
Stack
  reuse_frame
```

```
1
```

C1

RT1

```text
ints (n+1)
n=1
```

`tail#` pushes `xxs` and `reuse_frame` onto the stack.
Thunk reuse in GHC execution model

case (ints 1) of
  xxs@(x:_) -> .. (tail# xxs) ..

forcing RT₁

reuse_frame

Stack

RT₁ is forced
Thunk reuse in GHC execution model

case (ints 1) of
  xxs@(x:_ ) -> .. (tail# xxs) ..

Stack

RT₁ is forced and as a result C₂ is obtained.

RT₁ is reused as the delayed computation at the tail of C₂
case (ints 1) of
  xxs@(x:_) -> .. (tail# xxs) ..

Reuse_frame overwrites the tail of $C_1$ with a pointer to $C_2$. $C_1$'s address can be obtained from the stack.
case (ints 1) of
  xxs@(x:_) -> .. (tail# xxs) ..
Experiments

- nofib benchmark
  - imaginary, spectral, real
- GHC 7.0.3
- AMD Opteron CPU, 8GB main memory, Linux 2.6.32
- Compiled with -O2 flag
- Measured by GHC's statistic option -S
Total memory allocations

Geometric mean : 90.7 %
Execution time

Geometric mean : 111.0 %
Result

- Total memory allocations
  - Thunk reuse is effective in many programs except programs which allocate thunks for `tail`
- Execution time
  - In many programs, the execution time is between 100% and 110%, compared to the original GHC
Analysis on execution time

- **Advantage**
  - Time for memory allocations
  - The number of GC cycles

- **Disadvantage**
  - Overhead of tail#
  - Overhead of checking reusability of thunks
We have proposed a new implementation technique to suppress memory allocations by reusing thunks.

On current our implementation, total allocation is reduced in many case, while extra execution time is necessary.
We need advices

- We should improve execution time
  - Elimination of the overhead of \texttt{tail#}
  - Can we use the technique of \textit{pointer tagging} instead of allocating a thunk for \texttt{tail#}?
- Further optimization for self recursive functions such as \texttt{map}
  
  \begin{verbatim}
  map f [] = []
  map f (x:xs) = f x : \texttt{map'} f xs
  \end{verbatim}
  
  where \texttt{map'} f [] = []
  
  \begin{verbatim}
  map' f (x:xs) = f x : map' f xs
  \end{verbatim}

- We have to add new functions in STGtoSTG path, but we don't know how to do that

- Modifying GHC is a very hard task for me

\texttt{takano@coma-systems.com}