A Comparison of Program Transformation Systems

M. Dever & G. W. Hamilton
{mdever, hamilton}@computing.dcu.ie

Dublin City University

July 8, 2012
Outline

Background
  Program Transformation
  Language

Supercompilation
  Overview
  Termination

Distillation
  Overview
  Termination

Correctness & Efficiency

Benchmarking
  Overview
  Results
  Automating Benchmarking

Future Work
Program Transformation: Why?

- Functional programming typically makes heavy use of intermediate data, higher order functions and lazy evaluation.
  - Often results in more readable, elegant solutions to problems\(^1\).
    \[
    \text{reduce } (+) \ (\text{map square } xs)
    \]
  - However, these features can often lead to inefficiencies in the final program.
    - Heavy use of intermediate data, resulting in a negative impact on both execution time and memory usage.
  - How can we remove these inefficiencies?
    - Transform initial program into an equivalent program, with these inefficiencies removed.

---
Program Transformation: How?

- **Fold/Unfold** transformation techniques, first introduced by Burstall & Darlington\(^2\)

  Folding  Replacing an instance of a function body with its corresponding function call.

  Unfolding  Replacing a function call with a corresponding instance of its function body.

Program Transformation: How?

- A popular transformation technique that is used in many transformation systems
  - Partial Evaluation
  - Deforestation
  - Supercompilation
  - Distillation
Language Syntax

\[ e ::= x \]
\[ \ | c \ e_1 \ldots e_k \]
\[ \ | f \]
\[ \ | \lambda x. e \]
\[ \ | e_0 \ e_1 \]
\[ \ | \text{case } e_0 \ \text{of } p_1 \Rightarrow e_1 \ | \cdots | p_k \Rightarrow e_k \]
\[ \ | e_0 \ \text{where } f_1 = e_1 \ldots f_k = e_k \]

\[ p ::= c \ x_1 \ldots x_k \]
Example Program

\[
\text{loop } n \text{ Succ}(\text{Zero})
\]

\[
\text{where}
\]

\[
\text{loop } = \lambda n. \lambda \text{sum}. \text{case } n \text{ of }
\]
\[
\text{Zero } \Rightarrow \text{sum}
\]
\[
| \text{Succ}(n') \Rightarrow \text{loop'} n \text{ Succ}(\text{Zero}) n' \text{ sum}
\]

\[
\text{loop'} = \lambda i. \lambda \text{prod}. \lambda n. \lambda \text{sum}. \text{case } i \text{ of }
\]
\[
\text{Zero } \Rightarrow \text{loop } n (\text{add sum prod})
\]
\[
| \text{Succ}(i') \Rightarrow \text{loop'} i' (\text{mult i prod}) n \text{ sum}
\]

\[
\text{add } = \lambda m. \lambda n. \text{case } m \text{ of }
\]
\[
\text{Zero } \Rightarrow n
\]
\[
| \text{Succ}(m) \Rightarrow \text{Succ}(\text{add } m n)
\]

\[
\text{mult } = \lambda m. \lambda n. \text{case } m \text{ of }
\]
\[
\text{Zero } \Rightarrow \text{Zero}
\]
\[
| \text{Succ}(m) \Rightarrow \text{add } n (\text{mult } m n)
\]
Reduction Rules

\[
\begin{align*}
  f &= e \\
  f &\varrightarrow e
\end{align*}
\]
Reduction Rules

\[ f = e \]

\[ \begin{align*}
  f & \sim^f e \\
  ((\lambda x.e_0)\ e_1) & \beta \rightarrow (e_0\{x := e_1\})
\end{align*} \]
Reduction Rules

\[
\begin{align*}
f = e \\
f \xrightarrow{f} e & \quad \left((\lambda x.e_0) e_1\right) \xrightarrow{\beta} (e_0\{x := e_1\}) \\
\end{align*}
\]

\[
\begin{align*}
p_i = c \ x_1 \ldots x_n \\
\text{case} \ (c \ e_1 \ldots e_n) \text{ of } p_1 : e_1' | \ldots | p_k : e_k' & \xrightarrow{c} (e_i'\{x_1 := e_1, \ldots, x_n := e_n\}) \\
\end{align*}
\]
Reduction Rules

\[
\frac{f = e}{f \overset{f}{\Rightarrow} e}
\]

\[
\frac{(\lambda x.e_0) e_1 \overset{\beta}{\Rightarrow} (e_0 \{x := e_1\})}{e_0 \overset{r}{\Rightarrow} e_0'}
\]

\[
\frac{(e_0 e_1) \overset{r}{\Rightarrow} (e_0' e_1)}{p_i = c \; x_1 \ldots x_n}
\]

\[
\frac{(\text{case} \; (c \; e_1 \ldots e_n) \; \text{of} \; p_1 : e_1' \mid \ldots \mid p_k : e_k') \overset{\text{c}}{\Rightarrow} (e_i' \{x_1 := e_1, \ldots, x_n := e_n\})}{(\text{case} \; (c \; e_1' \ldots e_n') \; \text{of} \; p_1 : e_1' \mid \ldots \mid p_k : e_k') \overset{\text{c}}{\Rightarrow} (e_i' \{x_1 := e_1, \ldots, x_n := e_n\})}
\]
Reduction Rules

\[
\begin{align*}
f &= e \\
f \xrightarrow{f} e &\quad (\lambda x.e_0) e_1 \xrightarrow{\beta} (e_0\{x := e_1\}) \\
&\quad e_0 \xrightarrow{r} e'_0 \\
(e_0 \ e_1) &\xrightarrow{r} (e'_0 \ e_1) \\
p_i = c \ x_1 \ldots x_n &\quad (\text{case} \ (c \ e_1 \ldots e_n) \ \text{of} \ p_1 : e'_1 | \ldots | p_k : e'_k) \xrightarrow{c} (e'_i\{x_1 := e_1, \ldots, x_n := e_n\}) \\
&\quad e_0 \xrightarrow{r} e'_0 \\
&\quad (\text{case} \ e_0 \ \text{of} \ p_1 : e_1 | \ldots | p_k : e_k) \xrightarrow{r} (\text{case} \ e'_0 \ \text{of} \ p_1 : e_1 | \ldots | p_k : e_k)
\end{align*}
\]
Labelled Transition Systems

States $s_0, s_1, s_2, s_3$ (start state: $s_0$)
Labelled Transition Systems

\[ \alpha_4 \]

States \( s_0, s_1, s_2, s_3 \) (start state: \( s_0 \))

Actions \( \alpha_1, \alpha_2, \alpha_3, \alpha_4 \)
Labelled Transition Systems

States \( s_0, s_1, s_2, s_3 \) (start state: \( s_0 \))

Actions \( \alpha_1, \alpha_2, \alpha_3, \alpha_4 \)

Transitions \( s_0 \xrightarrow{\alpha_1} s_1, s_1 \xrightarrow{\alpha_2} s_2, s_2 \xrightarrow{\alpha_3} s_3, s_3 \xrightarrow{\alpha_4} s_0 \)
Labelled Transition Systems for Language

Labelled Transition System Actions

$\lambda x$: Abstraction over variable $x$

case $p$: Case selector

$\# i$: $i^{th}$ argument in an application

tau $\tau_f$: Function unfolding

tau $\tau_\beta$: $\beta$-reduction

tau $\tau_c$: Constructor elimination

Variable
Constructor
Labelled Transition Systems for Language

Labelled Transition System States

0

$Exp$

Stop State

Expression
Labelled Transition Systems for Language
Labelled Transition Systems for Language

\[ x e_1 \ldots e_k \]

\[ 0 \quad e_1 \quad \ldots \quad e_k \]

\[ c e_1 \ldots e_k \]

\[ 0 \quad e_1 \quad \ldots \quad e_k \]
Labelled Transition Systems for Language
Labelled Transition Systems for Language

\[ x \ e_1 \ldots e_k \]

\[ 0 \quad e_1 \ldots e_k \]

\[ f \]

\[ f = e \]

\[ e \]

\[ \lambda x . e \]

\[ \lambda x \]

\[ e \]
Labelled Transition Systems for Language

\[
x e_1 \ldots e_k
\]

\[
0 \quad e_1 \cdots e_k
\]

\[
f \quad \tau_f \ (f = e)
\]

\[
\lambda x. e
\]

\[
(\lambda x. e_0) \ e_1
\]

\[
e_0 \{x := e_1\}
\]
Labelled Transition Systems for Language

\[ x \cdot e_1 \ldots \cdot e_k \]

\[ f \]

\[ \tau_f \ (f = e) \]

\[ e \]

\[ \lambda x.e \]

\[ \tau_\beta \]

\[ e_0 \{ x := e \} \]

\[ \text{case } x \text{ of } p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k \]
Labelled Transition Systems for Language

\[
x e_1 \ldots e_k
\]

\[
c e_1 \ldots e_k
\]

\[
\text{case } x \text{ of } p_1 \Rightarrow e_1 | \ldots | p_k \Rightarrow e_k
\]

\[
\text{case } (c e_1 \ldots e_n) \text{ of } p_1 \Rightarrow e'_1 | \ldots | p_k \Rightarrow e'_k
\]
Supercompilation

- Introduced by Turchin\textsuperscript{3} but not really known outside Russia until later.
- Became more well known via \textit{positive supercompilation}\textsuperscript{4}.
  - A simplified algorithm retaining positive information propagation.
  - Defined using a more common functional language.

\textsuperscript{3}Turchin, V.F.: The concept of a supercompiler. ACM Transactions on Programming Languages and Systems (1986)
Positive Supercompilation: How?

**Driving** is performed on the input program to construct a labelled transition system, representing the symbolic computation of the program by normal order reduction.

Positive information propagation maintains known information about variables.

**Folding** is performed on encountering a **renaming** of a previously encountered **term**.

**Generalization**

is performed on encountering an **embedding** of a previously encountered **term** to ensure termination of the transformation.

**Residualization**

is performed to extract a (hopefully) more efficient program from the folded and generalized labelled transition system.
Termination

- An important issue associated with positive supercompilation is that of termination.
- The size of terms encountered during reduction can diverge, in which case a renaming will never be encountered and the transformation will not terminate.
- Termination can be ensured through the use of generalization.
- To represent the result of generalization, LTS’s can represent the result of generalization via generalized states which have the following form:
Generalization: When?

- A **whistle** is required to stop driving due to potential divergence, and to indicate that generalization should be performed.
- The **homeomorphic embedding relation** provides a suitable such whistle:

  - **Variable** $x \subseteq y$
    - $e \subseteq e_i$ for some $i \in \{1..n\}$
    - Diving $\frac{e \subseteq \phi(e_1 \ldots e_n)}{e_i \subseteq e'_i}$ for all $i \in \{1..n\}$
    - Coupling $\frac{\phi(e_1 \ldots e_n) \subseteq \phi(e'_1 \ldots e'_n)}$
Generalization: How?

- A **generalization** of expressions \(e\) and \(e'\) is a triple \((e_g, \theta, \theta')\) where \(\theta\) and \(\theta'\) are substitutions such that \(e_g\theta \equiv e\) and \(e_g\theta' \equiv e'\).

- A **most specific generalization** of expressions \(e\) and \(e'\) is a generalization \((e_g, \theta, \theta')\) such that for every other generalization \((e'_g, \theta'', \theta''')\) of \(e\) and \(e'\), \(e_g\) is an instance of \(e'_g\).
Positive Supercompilation: Summary

- **Strictly more powerful**, according to Sørensen\(^5\), than both partial evaluation and deforestation.

---


Positive Supercompilation: Summary

- Strictly **more powerful**, according to Sørensen\(^5\), than both partial evaluation and deforestation.
- Performs both **specialization** and **symbolic computation**.
  - Can specialize a naive pattern matcher to give a KMP pattern matcher.
  - This relies on positive information propagation, which is not done in partial evaluation or deforestation.

---


Positive Supercompilation: Summary

- Strictly more powerful, according to Sørensen\(^5\), than both partial evaluation and deforestation.
- Performs both specialization and symbolic computation.
  - Can specialize a naive pattern matcher to give a KMP pattern matcher.
  - This relies on positive information propagation, which is not done in partial evaluation or deforestation.
- Positive supercompilation can only produce a linear speedup\(^6\) in programs


Distillation

- Distillation, introduced by Hamilton\(^7\), is another program transformation technique
  - Like positive supercompilation, *driving* is used to perform a symbolic computation of a program, which constructs a potentially infinite labelled transition system.
  - **Positive information propagation** is also performed during driving.
  - Generalization and folding are performed with respect to the *labelled transition system* at each node, rather than just the expression it contains.

Distillation

- **Generalization** is performed on encountering an embedding of a previously encountered labelled transition system to ensure termination of the transformation.
- **Folding** is performed on encountering a renaming of a previously encountered labelled transition system.
Termination

- As with positive supercompilation, termination is an important issue associated with distillation.
  - Distillation has an alternate approach to termination.
  - Distillation compares LTSs to determine whether to fold or generalize.
  - There is an obviously difficulty in this as an LTS may be infinite.
  - However, it is acceptable to compare just the core component of an LTS from its root to where an unfolding of a previously encountered function is detected.
    - This core component will always be finite.
Generalization: How?

- Performed **incrementally** from roots of two LTSs.
- Increment is interval between **function unfoldings**.
- Corresponding states with **different transitions** are extracted using **lets**.
- Identical extractions are identified.
- These **lets** will be **distributed** through the generalized LTS:
Result of Distillation on Example Program

\( f \ n \ \text{Zero} \)

\[
f = \lambda n. \lambda x. \text{case } n \text{ of } \\
\text{Zero} \Rightarrow \text{Succ}(x) \\
\text{Succ}(n') \Rightarrow f \ n' \ \text{Succ}(\text{add } x \ (\text{mult } n' \ \text{Succ}(x)))
\]

\[
\text{add} = \lambda m. \lambda n. \text{case } m \text{ of } \\
\text{Zero} \Rightarrow n \\
| \text{Succ}(m) \Rightarrow \text{Succ}(\text{add } m \ n)
\]

\[
\text{mult} = \lambda m. \lambda n. \text{case } m \text{ of } \\
\text{Zero} \Rightarrow \text{Zero} \\
| \text{Succ}(m) \Rightarrow \text{add } n \ (\text{mult } m \ n)
\]
Distillation: Summary

- Strictly **more powerful** than positive supercompilation.
  - Therefore strictly more powerful, via Sørensen\(^8\), than partial evaluation and deforestation.

\(^8\)Sørensen, M.H.: Turchin’s supercompiler revisited - an operational theory of positive information propagation (1996)
Distillation: Summary

- Strictly **more powerful** than positive supercompilation.
  - Therefore strictly more powerful, via Sørensen\(^8\), than partial evaluation and deforestation.
- Performs **all** optimizations that positive supercompilation performs.

---

\(^8\)Sørensen, M.H.: Turchin’s supercompiler revisited - an operational theory of positive information propagation (1996)
Distillation: Summary

- Strictly **more powerful** than positive supercompilation.
  - Therefore strictly more powerful, via Sørensen\(^8\), than partial evaluation and deforestation.
- Performs **all** optimizations that positive supercompilation performs.
- Distillation is capable of obtaining a **superlinear speedup** in programs.

---
\(^8\)Sørensen, M.H.: Turchin’s supercompiler revisited - an operational theory of positive information propagation (1996)
Correctness

• **Partial correctness** of both positive supercompilation and distillation can be proved by showing that there is a **bisimulation** between the LTS corresponding to a program before transformation, and the LTS resulting from transformation.
Correctness

- **Partial correctness** of both positive supercompilation and distillation can be proved by showing that there is a **bisimulation** between the LTS corresponding to a program before transformation, and the LTS resulting from transformation.

- **Total correctness** of both positive supercompilation and distillation also requires showing that they **terminate**.
Correctness

- **Partial correctness** of both positive supercompilation and distillation can be proved by showing that there is a **bisimulation** between the LTS corresponding to a program before transformation, and the LTS resulting from transformation.

- **Total correctness** of both positive supercompilation and distillation also requires showing that they **terminate**.
  - This involves showing that there is a **size bound** on the core components which are encountered during transformation (expressions in supercompilation and LTSs in distillation).
Correctness

- **Partial correctness** of both positive supercompilation and distillation can be proved by showing that there is a **bisimulation** between the LTS corresponding to a program before transformation, and the LTS resulting from transformation.

- **Total correctness** of both positive supercompilation and distillation also requires showing that they **terminate**.
  - This involves showing that there is a **size bound** on the core components which are encountered during transformation (expressions in supercompilation and LTSs in distillation).
  - If there is such a bound, then a **renaming** must eventually be encountered, and folding can be performed.
Bisimulation

Strong Bisimulation
Bisimulation

Strong Bisimulation
Bisimulation

Strong Bisimulation
Bisimulation

Strong Bisimulation

\[ e_0 \xrightarrow{\alpha_1} e_1 \xrightarrow{\tau} e_2 \xrightarrow{\alpha_2} e_3 \]

\[ e_4 \xrightarrow{\alpha_1} e_5 \xrightarrow{\tau} e_6 \xrightarrow{\alpha_2} e_7 \]
Bisimulation

Strong Bisimulation

\[ \begin{align*}
&\quad e_0 \xrightarrow{\alpha_1} e_1 \xrightarrow{\tau} e_2 \xrightarrow{\alpha_2} e_3 \\
&\quad e_4 \xrightarrow{\alpha_1} e_5 \xrightarrow{\tau} e_6 \xrightarrow{\alpha_2} e_7
\end{align*} \]
Bisimulation

Weak Bisimulation

\[ \alpha_1 \quad \tau \quad \alpha_2 \]

\[ \tau \quad \alpha_2 \]

\[ \tau \quad \alpha_1 \quad \tau \]

\[ e_0 \quad e_1 \quad e_2 \quad e_3 \quad e_4 \quad e_5 \quad e_6 \quad e_7 \]
Bisimulation

Weak Bisimulation

\[ e_0 \xrightarrow{\alpha_1} e_1 \xrightarrow{\tau} e_2 \xrightarrow{\alpha_2} e_3 \]

\[ e_4 \xrightarrow{\tau} e_5 \xrightarrow{\alpha_1} e_6 \xrightarrow{\tau} e_7 \]
Bisimulation

Weak Bisimulation

\[ \begin{align*}
  e_0 & \xrightarrow{\alpha_1} e_1 \\
  e_1 & \xrightarrow{\tau} e_2 \\
  e_2 & \xrightarrow{\alpha_2} e_3 \\
  e_4 & \xrightarrow{\tau} e_5 \\
  e_5 & \xrightarrow{\alpha_1} e_6 \\
  e_6 & \xrightarrow{\tau} e_7
\end{align*} \]
Bisimulation

Weak Bisimulation
Efficiency

- In positive supercompilation, there can only be a **constant** number of silent transitions between each recursive call of a function.
Efficiency

- In positive supercompilation, there can only be a constant number of silent transitions between each recursive call of a function.
  - Removing these will therefore only give a linear speedup.
Efficiency

- In positive supercompilation, there can only be a constant number of silent transitions between each recursive call of a function.
  - Removing these will therefore only give a linear speedup.
- In distillation, the number of silent transitions between each recursive call of a function can be increasing.
Efficiency

- In positive supercompilation, there can only be a constant number of silent transitions between each recursive call of a function.
  - Removing these will therefore only give a linear speedup.
- In distillation, the number of silent transitions between each recursive call of a function can be increasing.
  - These can still be collapsed down and identified, thus giving a superlinear speedup.
Efficiency

- In positive supercompilation, there can only be a **constant** number of silent transitions between each recursive call of a function.
  - Removing these will therefore only give a **linear** speedup.
- In distillation, the number of silent transitions between each recursive call of a function can be **increasing**.
  - These can still be collapsed down and identified, thus giving a **superlinear** speedup.
- Essentially, the key difference between the two is that positive supercompilation looks at code fragments before they have been evaluated, and distillation looks at them after.
How do these transformation systems compare?

- As we have seen, these are both theoretically powerful transformation systems
- Part of the focus of this paper is on seeing whether reality lives up to the theory.
- There are a number of things we need to compare these transformation systems:
  - A suite of programs to benchmark and evaluate
  - A means to obtain necessary benchmark information about the runtime of benchmarked programs
  - For good measure, another transformation system, not implemented by us
What are we going to benchmark?

sumsquares
A program that calculates the sum of the squares of two lists

(word|line|char)count
Programs that respectively count the number of words, lines and characters in a given input

exp3_8
A program that calculates 3 raised to the power of a given number

nrev
A program that performs a naive list reversal

Other programs from previous\(^9\) works and the nofib benchmark suite

How do we obtain benchmark information?

65,896 bytes allocated in the heap
3,512 bytes copied during GC
44,416 bytes maximum residency (1 sample(s))
17,024 bytes maximum slop
  1 MB total memory in use (0 MB lost due to fragmentation)

<table>
<thead>
<tr>
<th></th>
<th>Tot time (elapsed)</th>
<th>Avg pause</th>
<th>Max pause</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gen 0</td>
<td>0 colls, 0 par</td>
<td>0.00s</td>
<td>0.00s</td>
</tr>
<tr>
<td>Gen 1</td>
<td>1 colls, 0 par</td>
<td>0.00s</td>
<td>0.00s</td>
</tr>
<tr>
<td>INIT</td>
<td>time</td>
<td>0.00s (0.00s elapsed)</td>
<td></td>
</tr>
<tr>
<td>MUT</td>
<td>time</td>
<td>0.00s (0.00s elapsed)</td>
<td></td>
</tr>
<tr>
<td>GC</td>
<td>time</td>
<td>0.00s (0.00s elapsed)</td>
<td></td>
</tr>
<tr>
<td>EXIT</td>
<td>time</td>
<td>0.00s (0.00s elapsed)</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>time</td>
<td>0.00s (0.00s elapsed)</td>
<td></td>
</tr>
<tr>
<td>%GC</td>
<td>time</td>
<td>9.6% (23.8% elapsed)</td>
<td></td>
</tr>
</tbody>
</table>

Alloc rate  47,135,908 bytes per MUT second
Productivity 82.3% of total user, 137.8% of total elapsed
HOSC: Another Supercompiler

- We had intended on benchmarking against two-level supercompilation\textsuperscript{10}
  - Like distillation, capable of obtaining a superlinear speed up
- However, we had difficulties getting this supercompiler working
- We opted to benchmark against the HOSC single level supercompiler instead

\textsuperscript{10}Klyuchnikov, I.G.: Towards effective two-level supercompilation (2010)
# Execution Time Comparisons

<table>
<thead>
<tr>
<th>Name</th>
<th>Unoptimized</th>
<th>Supercompilation</th>
<th>HOSC</th>
<th>Distillation</th>
</tr>
</thead>
<tbody>
<tr>
<td>nrev</td>
<td>62.5</td>
<td>53.3</td>
<td>68.7</td>
<td>0.1</td>
</tr>
<tr>
<td>charcount</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>exp3_8</td>
<td>45.9</td>
<td>32.4</td>
<td>52.1</td>
<td>-</td>
</tr>
<tr>
<td>factorial</td>
<td>2.6</td>
<td>2.5</td>
<td>2.8</td>
<td>-</td>
</tr>
<tr>
<td>linecount</td>
<td>28.7</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>primes</td>
<td>79.2</td>
<td>75.9</td>
<td>104.5</td>
<td>-</td>
</tr>
<tr>
<td>raytracer</td>
<td>12.7</td>
<td>10.0</td>
<td>10.4</td>
<td>10.0</td>
</tr>
<tr>
<td>rfib</td>
<td>57.7</td>
<td>35.3</td>
<td>37.7</td>
<td>-</td>
</tr>
<tr>
<td>sumsquare</td>
<td>81.9</td>
<td>72.7</td>
<td>76.9</td>
<td>-</td>
</tr>
<tr>
<td>treeflip</td>
<td>51.2</td>
<td>29.9</td>
<td>32.2</td>
<td>-</td>
</tr>
<tr>
<td>wordcount</td>
<td>29.8</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
</tbody>
</table>
Execution Time Comparisons

- Perhaps most interesting is the naive list reversal program.
  - Original: 62.5 seconds
  - Supercompiled: 53.3 seconds - 14.72% decrease in execution time
  - HOSC: 68.7 seconds - 9.92% increase in execution time
  - Distillation: 0.1 seconds - 99.84% decrease in execution time
## Memory Usage Comparisons

<table>
<thead>
<tr>
<th>Name</th>
<th>Unoptimized</th>
<th>Supercompilation</th>
<th>HOSC</th>
<th>Distillation</th>
</tr>
</thead>
<tbody>
<tr>
<td>nrev</td>
<td>8</td>
<td>6</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>charcount</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>exp3_8</td>
<td>6</td>
<td>4</td>
<td>6</td>
<td>-</td>
</tr>
<tr>
<td>factorial</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>linecount</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>primes</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>raytracer</td>
<td>1011</td>
<td>730</td>
<td>732</td>
<td>732</td>
</tr>
<tr>
<td>rfib</td>
<td>2073</td>
<td>1061</td>
<td>1047</td>
<td>-</td>
</tr>
<tr>
<td>sumsquare</td>
<td>2313</td>
<td>2391</td>
<td>2221</td>
<td>-</td>
</tr>
<tr>
<td>treeflip</td>
<td>2176</td>
<td>1083</td>
<td>1069</td>
<td>-</td>
</tr>
<tr>
<td>wordcount</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Memory Usage Comparisons

- Again, perhaps most interesting is the naive list reversal program.
  - Original: 8 MB
  - Supercompiled: 6 MB - 25% decrease in memory usage
  - HOSC: 11 MB - 11% increase in memory usage
  - Distillation: 3 MB - 62.5% decrease in memory usage
Automating Benchmarking

- One of the tedious and time consuming tasks associated with implementing program transformers is that of benchmarking.
- Automating program transformation is obviously very important, but what about implementing the benchmarking of such transformations?
A (somewhat) automatic benchmarking system

- Upload two files: an input file to be transformed, and an arguments file to be used during benchmarking.
- Files are tested for compilation, if this fails then receive the compilation error.
- If the files compile, then:
  - They are saved to a database.
  - A task is sent to a benchmarking machine.
  - Input program is transformed (currently only positive supercompilation).
  - Input and transformed programs are then benchmarked.
A (somewhat) automatic benchmarking system

- How are these programs benchmarked?
  - Via three user inputs, a number indicating the amount of benchmark points, a number indicating the number of runs and the arguments file.
  - For each benchmark point, each program is run the specified number of times.
  - Benchmark data for each point is saved, and averages for each point are displayed for each program.
A (somewhat) automatic benchmarking system

- This benchmark data is publicly viewable
- As are the input program, and each transformation result
- Users have ability to view benchmark data via benchmark point or transformation technique
```haskell
module Main where

import System.Environment (getArgs)
import Arguments

main = do
  args <- getArgs
  let level = read (head args) :: Integer
  print $ root (randomXS level)

  root = \xs -> nrev xs

  nrev = \xs -> case xs of
          [] -> []
          (y:ys) -> app (nrev ys) [y]

  app = \xs ys -> case xs of
          [] -> ys
          (z:zs) -> (z:app zs ys)
```

```
module Arguments where

randomXS = \level -> case level of
  1 -> [1..10]
  2 -> [10..100]
  3 -> [100..1000]
  4 -> [1000..10000]
  5 -> [10000..30000]
```

**Run Information**

<table>
<thead>
<tr>
<th>Level Number</th>
<th>GHC</th>
<th>GHC -O2</th>
<th>Super</th>
<th>Super -O2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>2</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>4</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
Some Links

http://github.com/distillation/distiller - Distillation Source code

http://github.com/distillation/distill_web - Benchmarking Website Source
Future Work

- Automate the parallelization of functional programs:
  - Aim to target Nvidia GPGPU architecture initially
  - Use skeletons to guide parallelization process

- Finish and expand the benchmarking site:
  - We welcome any collaboration and/or suggestions
  - Having somewhere to run benchmarks against many transformation tools would be quite beneficial