A Comparison of Program Transformation Systems

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

Dublin City University

July 8, 2012

Outline

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].

    reduce (+) (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.

[1] Hughes, J.: Why Functional Programming Matters. Computer Journal (1989)

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.

[2] Burstall, R.M., Darlington, J.: A transformation system for developing recursive programs. Journal of the Association for Computing Machinery (1977)

Program Transformation: How?

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

Language Syntax

$e ::= x$	Variable
$\mid\ c\ e_1 \ldots e_k$	Constructor Application
$\mid\ f$	Function Call
$\mid\ \lambda x.e$	λ-Abstraction
$\mid\ e_0\ e_1$	Application
$\mid\ \textbf{case}\ e_0\ \textbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k$	Case Expression
$\mid\ e_0\ \textbf{where}\ f_1 = e_1 \ldots f_k = e_k$	Local Function Definition
$p ::= c\ x_1 \ldots x_k$	Pattern

Example Program

$$
\begin{array}{lcl}
\multicolumn{3}{l}{loop\ n\ Succ(Zero)} \\
\multicolumn{3}{l}{\textbf{where}} \\
loop & = & \lambda n.\lambda sum.\textbf{case}\ n\ \textbf{of} \\
 & & Zero \Rightarrow sum \\
 & & \mid Succ(n') \Rightarrow loop'\ n\ Succ(Zero)\ n'\ sum \\
 & & \\
loop' & = & \lambda i.\lambda prod.\lambda n.\lambda sum.\textbf{case}\ i\ \textbf{of} \\
 & & Zero \Rightarrow loop\ n\ (add\ sum\ prod) \\
 & & \mid Succ(i') \Rightarrow loop'\ i'(mult\ i\ prod)\ n\ sum \\
 & & \\
add & = & \lambda m.\lambda n.\textbf{case}\ m\ \textbf{of} \\
 & & Zero \Rightarrow n \\
 & & \mid Succ(m) \Rightarrow Succ(add\ m\ n) \\
 & & \\
mult & = & \lambda m.\lambda n.\textbf{case}\ m\ \textbf{of} \\
 & & Zero \Rightarrow Zero \\
 & & \mid Succ(m) \Rightarrow add\ n\ (mult\ m\ n)
\end{array}
$$

Reduction Rules

$$\frac{f = e}{f \overset{f}{\rightsquigarrow} e}$$

Reduction Rules

$$\frac{f = e}{f \overset{f}{\leadsto} e} \qquad ((\lambda x.e_0)\ e_1) \overset{\beta}{\leadsto} (e_0\{x := e_1\})$$

Reduction Rules

$$\frac{f = e}{f \overset{f}{\rightsquigarrow} e} \qquad ((\lambda x.e_0)\ e_1) \overset{\beta}{\rightsquigarrow} (e_0\{x := e_1\})$$

$$\frac{p_i = c\ x_1 \ldots x_n}{(\mathbf{case}\ (c\ e_1 \ldots e_n)\ \mathbf{of}\ p_1 : e'_1 | \ldots | p_k : e'_k) \overset{c}{\rightsquigarrow} (e'_i\{x_1 := e_1, \ldots, x_n := e_n\})}$$

Reduction Rules

$$\frac{f = e}{f \overset{f}{\rightsquigarrow} e} \qquad ((\lambda x.e_0)\ e_1) \overset{\beta}{\rightsquigarrow} (e_0\{x := e_1\}) \qquad \frac{e_0 \overset{r}{\rightsquigarrow} e_0'}{(e_0\ e_1) \overset{r}{\rightsquigarrow} (e_0'\ e_1)}$$

$$\frac{p_i = c\ x_1 \ldots x_n}{(\textbf{case}\ (c\ e_1 \ldots e_n)\ \textbf{of}\ p_1 : e_1' | \ldots | p_k : e_k') \overset{c}{\rightsquigarrow} (e_i'\{x_1 := e_1, \ldots, x_n := e_n\})}$$

Reduction Rules

$$\frac{f = e}{f \overset{f}{\rightsquigarrow} e} \qquad ((\lambda x.e_0)\ e_1) \overset{\beta}{\rightsquigarrow} (e_0\{x := e_1\}) \qquad \frac{e_0 \overset{r}{\rightsquigarrow} e_0'}{(e_0\ e_1) \overset{r}{\rightsquigarrow} (e_0'\ e_1)}$$

$$\frac{p_i = c\ x_1 \ldots x_n}{(\textbf{case}\ (c\ e_1 \ldots e_n)\ \textbf{of}\ p_1 : e_1' | \ldots | p_k : e_k') \overset{c}{\rightsquigarrow} (e_i'\{x_1 := e_1, \ldots, x_n := e_n\})}$$

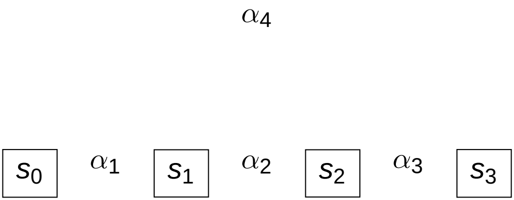
$$\frac{e_0 \overset{r}{\rightsquigarrow} e_0'}{(\textbf{case}\ e_0\ \textbf{of}\ p_1 : e_1 | \ldots p_k : e_k) \overset{r}{\rightsquigarrow} (\textbf{case}\ e_0'\ \textbf{of}\ p_1 : e_1 | \ldots p_k : e_k)}$$

Labelled Transition Systems

$\boxed{s_0}$ $\boxed{s_1}$ $\boxed{s_2}$ $\boxed{s_3}$

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

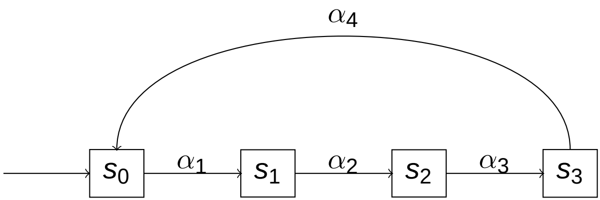
Labelled Transition Systems



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

Actions $\alpha_1, \alpha_2, \alpha_3, \alpha_4$

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

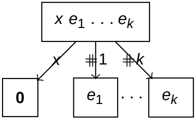
x	Variable
c	Constructor
$\#i$	i^{th} argument in an application
λx	Abstraction over variable x
case	Case selector
p	Case branch pattern
τ_f	Function unfolding
τ_β	β-reduction
τ_c	Constructor elimination

Labelled Transition Systems for Language

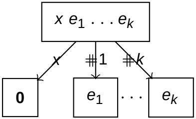
Labelled Transition System States

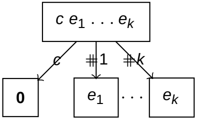
0	Stop State
Exp	Expression

Labelled Transition Systems for Language

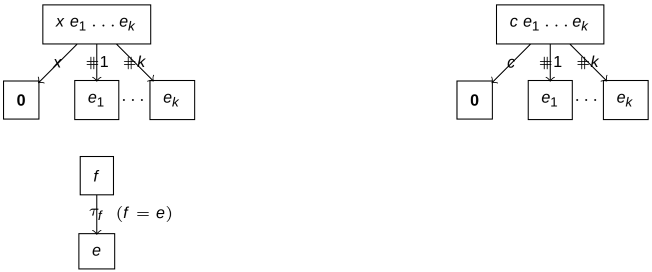


Labelled Transition Systems for Language

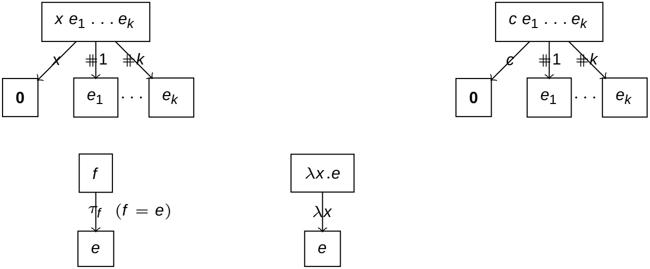




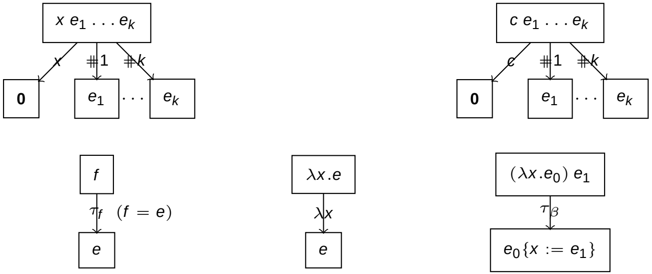
Labelled Transition Systems for Language



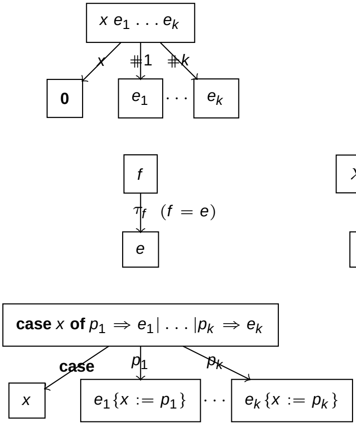
Labelled Transition Systems for Language



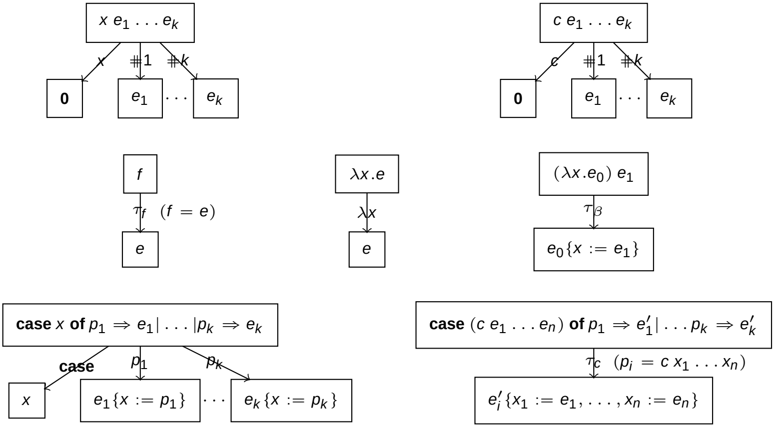
Labelled Transition Systems for Language



Labelled Transition Systems for Language



Labelled Transition Systems for Language



Supercompilation

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

[3] Turchin, V.F.: The concept of a supercompiler. ACM Transactions on Programming Languages and Systems (1986)

[4] Sørensen, M., Glück, R., Jones, N.: A positive supercompiler. Journal of Functional Programming (1993)

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.

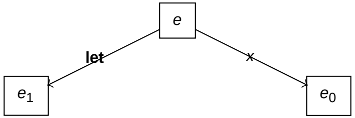
Folding is performed on encountering a **renaming** of a previously encountered **term**.

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 $x \trianglelefteq y$

Diving $$\frac{e \trianglelefteq e_i \text{ for some } i \in \{1..n\}}{e \trianglelefteq \phi(e_1 \ldots e_n)}$$

Coupling $$\frac{e_i \trianglelefteq e'_i \text{ for all } i \in \{1..n\}}{\phi(e_1 \ldots e_n) \trianglelefteq \phi(e'_1 \ldots e'_n)}$$

Generalization: How?

- A **generalization** of expressions e and e' is a triple (e_g, θ, θ') where θ and θ' 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, θ, θ') 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.

[5] Sørensen, M., Glück, R., Jones, N.: A positive supercompiler. Journal of Functional Programming (1993)

[6] Sørensen, M., Glück, R., Jones, N.: A positive supercompiler. Journal of Functional Programming (1993)

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.

[5] Sørensen, M., Glück, R., Jones, N.: A positive supercompiler. Journal of Functional Programming (1993)

[6] Sørensen, M., Glück, R., Jones, N.: A positive supercompiler. Journal of Functional Programming (1993)

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

[5] Sørensen, M., Glück, R., Jones, N.: A positive supercompiler. Journal of Functional Programming (1993)

[6] Sørensen, M., Glück, R., Jones, N.: A positive supercompiler. Journal of Functional Programming (1993)

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.

[7] Hamilton, G.W.: Distillation: Extracting the essence of programs. Proceedings of the ACM Workshop on Partial Evaluation and Program Manipulation (2007)

Distillation

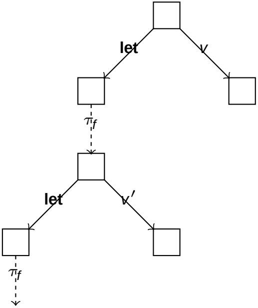
- **Generalization** is performed on encountering an **embedding** of a previously encountered **labelled transition system** to ensure termination of the transformation.
- **Folding** is performed on encountering a **renaming** of a previously encountered labelled transition system.

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 **let**s.
- Identical extractions are **identified**.
- These **let**s will be **distributed** through the generalized LTS:



Result of Distillation on Example Program

$$
\begin{array}{l}
f\ n\ Zero \\
\textbf{where} \\
\begin{array}{lcl}
f & = & \lambda n.\lambda x.\textbf{case}\ n\ \textbf{of} \\
 & & Zero \Rightarrow Succ(x) \\
 & & Succ(n') \Rightarrow f\ n'\ Succ(add\ x\ (mult\ n'\ Succ(x))) \\
 & & \\
add & = & \lambda m.\lambda n.\textbf{case}\ m\ \textbf{of} \\
 & & Zero \Rightarrow n \\
 & & \mid Succ(m) \Rightarrow Succ(add\ m\ n) \\
 & & \\
mult & = & \lambda m.\lambda n.\textbf{case}\ m\ \textbf{of} \\
 & & Zero \Rightarrow Zero \\
 & & \mid Succ(m) \Rightarrow add\ n\ (mult\ m\ n)
\end{array}
\end{array}
$$

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

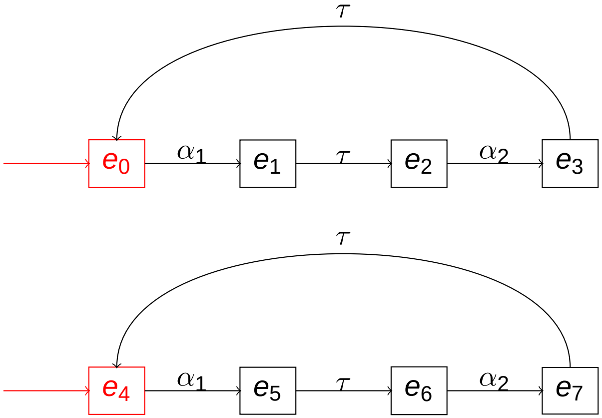
- **Partial correctness** of both positive supercompilation and distillation can be proved by showing that there is a **bisimulation** between the LTS corresponding to a program before transformation, and the LTS resulting from transformation.
- **Total correctness** of both positive supercompilation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

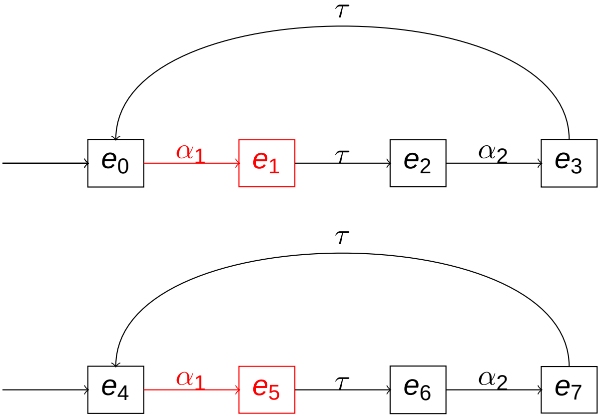
Bisimulation

Strong Bisimulation



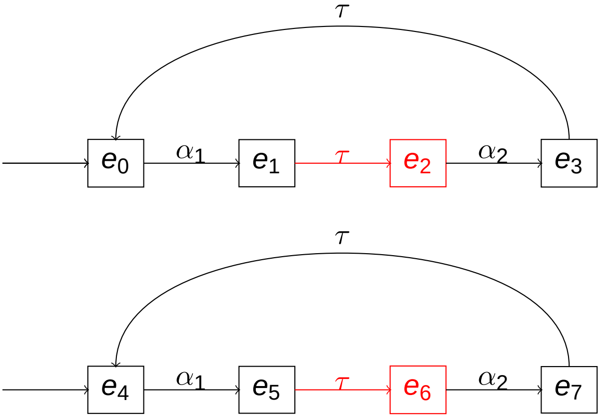
Bisimulation

Strong Bisimulation



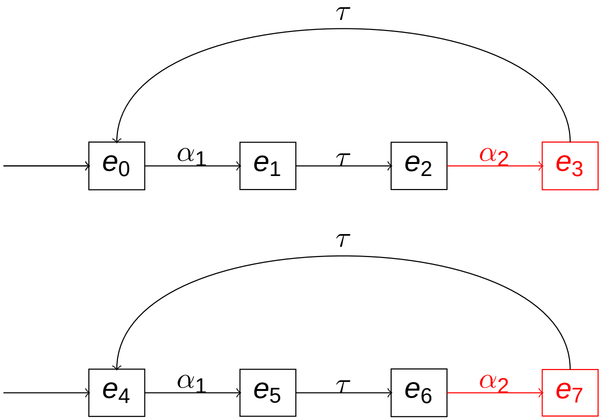
Bisimulation

Strong Bisimulation



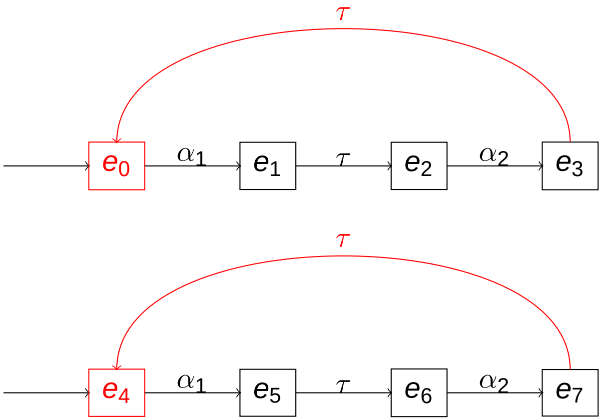
Bisimulation

Strong Bisimulation



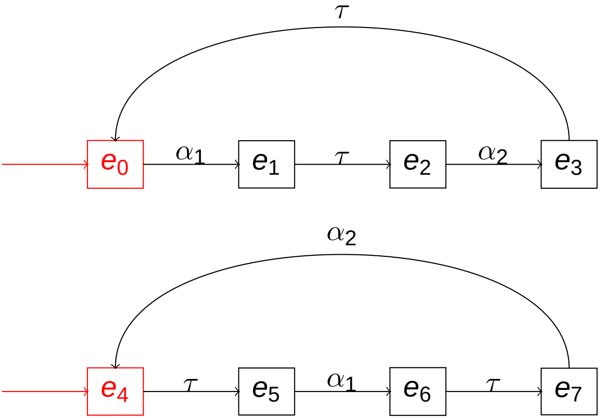
Bisimulation

Strong Bisimulation



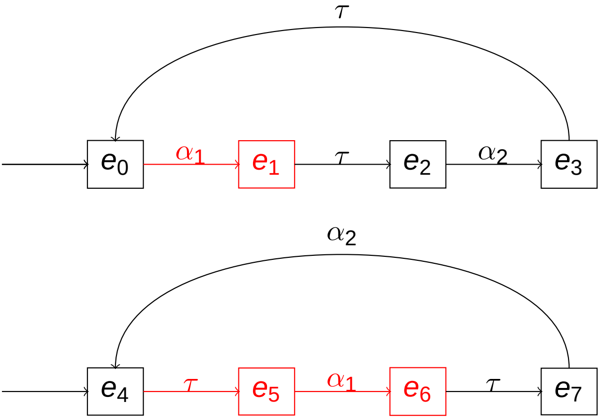
Bisimulation

Weak Bisimulation



Bisimulation

Weak Bisimulation



Bisimulation

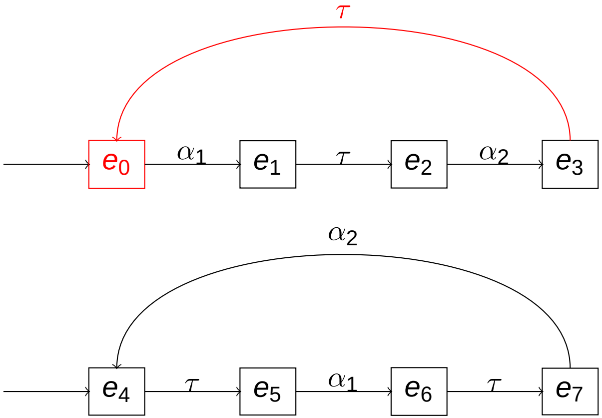
Weak Bisimulation

$$\begin{tikzcd}
\arrow[r] & e_0 \arrow[r, "\alpha_1"] & e_1 \arrow[r, "\tau"] & e_2 \arrow[r, "\alpha_2"] & e_3 \arrow[lll, bend right, "\tau"']
\end{tikzcd}$$

$$\begin{tikzcd}
\arrow[r] & e_4 \arrow[r, "\tau"] & e_5 \arrow[r, "\alpha_1"] & e_6 \arrow[r, "\tau"] & e_7 \arrow[lll, bend right, "\alpha_2"']
\end{tikzcd}$$

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

[9] Mitchell, N., Runciman, C.: A supercompiler for core Haskell. In: IFL 2007 (2008)

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)

                                   Tot time (elapsed)  Avg pause  Max pause
Gen  0         0 colls,     0 par    0.00s    0.00s     0.0000s    0.0000s
Gen  1         1 colls,     0 par    0.00s    0.00s     0.0002s    0.0002s

INIT    time    0.00s  (  0.00s elapsed)
MUT     time    0.00s  (  0.00s elapsed)
GC      time    0.00s  (  0.00s elapsed)
EXIT    time    0.00s  (  0.00s elapsed)
Total   time    0.00s  (  0.00s elapsed)

%GC     time       9.6%  (23.8% elapsed)

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[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

[10] Klyuchnikov, I.G.: Towards effective two-level supercompilation (2010)

Execution Time Comparisons

Name	Unoptimized	Supercompilation	HOSC	Distillation
nrev	62.5	53.3	68.7	0.1
charcount	0.01	0.01	0.01	0.01
exp3_8	45.9	32.4	52.1	-
factorial	2.6	2.5	2.8	-
linecount	28.7	0.01	0.01	0.01
primes	79.2	75.9	104.5	-
raytracer	12.7	10.0	10.4	10.0
rfib	57.7	35.3	37.7	-
sumsquare	81.9	72.7	76.9	-
treeflip	51.2	29.9	32.2	-
wordcount	29.8	0.01	0.01	0.01

Execution Time Comparisions

- 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

Name	Unoptimized	Supercompilation	HOSC	Distillation
nrev	8	6	11	3
charcount	3	3	3	3
exp3_8	6	4	6	-
factorial	3	3	3	-
linecount	6	1	1	1
primes	2	2	2	-
raytracer	1011	730	732	732
rfib	2073	1061	1047	-
sumsquare	2313	2391	2221	-
treeflip	2176	1083	1069	-
wordcount	6	1	1	1

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

Distillr Programs Login Regist

Input Supercompiled

```
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)
```

Arguments

```
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

Time Memory

Level Number	GHC	GHC -O2	Super	Super -O2
1	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0
4	1.0	1.0	1.0	1.0

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