Program Transformation in a Labelled Transition Systems Framework

G.W. Hamilton

School of Computing
Dublin City University
Dublin 9, Ireland
hamilton@computing.dcu.ie

September 12, 2016
Outline

1. Introduction
2. Language
3. Labelled Transition Systems
4. Positive Supercompilation
5. Distillation
6. Conclusion
The use of intermediate structures, lazy evaluation and higher-order functions in functional programming facilitates a more elegant and readable style of programming:

\[
\text{sum}\ (\text{squares}\ (\text{upto}\ 1\ n))\ 0
\]

where

\[
\text{sum} = \lambda x.\lambda a.\text{case}\ x\ of
\]
\[
\begin{array}{l}
\[] \Rightarrow a \\
| \ x' : x' \Rightarrow \text{sum}\ x'\ (a + x')
\end{array}
\]

\[
\text{squares} = \lambda x.\text{case}\ x\ of
\]
\[
\begin{array}{l}
\[] \Rightarrow [] \\
| \ x' : x' \Rightarrow x'^2 : (\text{squares}\ x')
\end{array}
\]

\[
\text{upto} = \lambda m.\lambda n.\text{case}\ (m > n)\ of
\]
\[
\begin{array}{l}
\text{True} \Rightarrow [] \\
| \text{False} \Rightarrow m : \text{upto}\ (m + 1)\ n
\end{array}
\]
However, this is inefficient.

One solution to this problem is to transform these programs into more efficient equivalent programs.

The unfold/fold program transformation framework\(^a\) is one approach to doing this.

- **Unfolding**: replacing a function call with a corresponding instance of the function body.
- **Folding**: replacing an instance of a function body with the corresponding function call.

Introduction

Unfold/Fold Program Transformation


- **Supercompilation**: V.F. Turchin, "The Concept of a Supercompiler", *ACM Transactions on Programming Languages and Systems* 8(3): 90–121, 1986

Introduction

Partial Evaluation

- Theoretical possibility of partial evaluation was established many years ago in recursive function theory as Kleene’s s-m-n theorem.
- Program optimisation is performed by specialisation.
- Evaluates computations on static data, producing a residual program.
- Running residual program on remaining dynamic data will achieve the same result, but hopefully more efficiently.
- Uses only constant propagation.
## Introduction

### Deforestation

- Algorithm for removing intermediate data structures (‘trees’) from functional programs by **composing** functions.
- Can also perform program **specialisation**.
- Function definitions restricted to **treeless form** to ensure termination.
- Also uses only **constant propagation**.
- First defined for a first-order language by Wadler\(^a\).
- First extended to a higher-order language by Hamilton\(^b\).

---

\(^a\)Wadler, P.: *Deforestation: Transforming Programs to Eliminate Trees*. Lecture Notes in Computer Science 300, 344-358 (1988)

Supercompilation

First developed by Turchin\textsuperscript{a} in Russia in 1970s, but did not become more widely known until much later.

- Published only in less accessible journals.
- Defined on the unconventional language \textit{Refal}.

Became more widely known through \textit{positive supercompilation} by Sørensen, Glück and Jones\textsuperscript{b}.

- Simplified algorithm.
- Defined on a more familiar functional language.

\textsuperscript{a}Turchin, V.: \textit{Program Transformation by Supercompilation}. Lecture Notes in Computer Science 217, 257-281 (1985)

Strictly more powerful than deforestation and partial evaluation.

- Extra power is due to information propagation which maintains known information about variables.
- Turchin’s supercompiler maintains positive and negative information about variables based on the outcome of tests.
- The positive supercompiler maintains only positive information.

Can also perform program specialisation.

Superlinear speedups are obtained for very few interesting programs.
Distillation

First defined by Hamilton\(^a\).

Builds on top of positive supercompilation.

Strictly more powerful than positive supercompilation, and hence also deforestation and partial evaluation.

Extra power is due to generalisation and folding being performed with respect to recursive terms.

Can produce many more superlinear speedups in programs that are not produced by positive supercompilation.

Language

Syntax

\[
\begin{align*}
\text{prog} &::= e_0 \textbf{ where } f_1 = e_1 \ldots f_k = e_k & \text{Program} \\
\text{e} &::= x \\
&\quad | c \; e_1 \ldots e_k \\
&\quad | \lambda x . e \\
&\quad | f \\
&\quad | e_0 \; e_1 \\
&\quad | \textbf{case} \; e_0 \; \textbf{of} \; p_1 \Rightarrow e_1 | \cdots | p_k \Rightarrow e_k & \text{Case Expression} \\
&\quad | \textbf{let} \; x = e_0 \; \textbf{in} \; e_1 & \text{Let Expression} \\
\text{p} &::= c \; x_1 \ldots x_k & \text{Pattern}
\end{align*}
\]
Language

Semantics

\[
\begin{align*}
((\lambda x. e_0) \ e_1) & \xrightarrow{\beta} (e_0 \{x \mapsto e_1\}) & \text{(let } x = e_0 \text{ in } e_1) & \xrightarrow{\beta} (e_1 \{x \mapsto e_0\}) \\
\frac{f = e}{f \xrightarrow{f} e} & & \frac{e_0 \xrightarrow{r} e'_0}{(e_0 \ e_1) \xrightarrow{r} (e'_0 \ e_1)} \\
\frac{e_0 \xrightarrow{r} e'_0}{(\text{case } e_0 \text{ of } p_1 : e_1 | \ldots | p_k : e_k) \xrightarrow{r} (\text{case } e'_0 \text{ of } p_1 : e_1 | \ldots | p_k : e_k)} \\
\frac{p_i = c \ x_1 \ldots x_n}{(\text{case } (c \ e_1 \ldots e_n) \text{ of } p_1 : e'_1 | \ldots | p_k : e'_k) \xrightarrow{c} (e_i \{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\})}
\end{align*}
\]
We give here a presentation of positive supercompilation making use of labelled transition systems in place of process trees.

The LTS associated with program $e_0$ is given by $(\mathcal{E}, e_0, \rightarrow, \text{Act})$:

- $\mathcal{E}$ is the set of states of the LTS.
  - Each is an expression, or the end-of-action state $0$.
- $e_0$ is the start state
- Act is a set of actions $\alpha$.
- $\rightarrow \subseteq \mathcal{E} \times \text{Act} \times \mathcal{E}$ is a transition relation that relates pairs of states by actions.
  - If $e \in \mathcal{E}$ and $e \xrightarrow{\alpha} e'$ then $e' \in \mathcal{E}$. 
Labelled Transition Systems

Actions:
- $x$, a free variable;
- $i$, a bound variable with De Bruijn index $i$;
- $c$, a constructor in an application or case pattern;
- $\lambda$, a $\lambda$-abstraction;
- $\circ$, the function in an application;
- $\#i$, the $i^{th}$ argument in an application;
- `case`, a case selector;
- `let`, a let variable;
- `in`, a let body.
Labelled Transition Systems
Labelled Transition Systems

\[ x \]
\[ x \]
\[ 0 \]

\[ c \quad e_1 \ldots e_k \]
\[ c \quad \#1 \quad \#k \]
\[ 0 \quad e_1 \quad e_k \]
Labelled Transition Systems

\[
x \quad 0
\]
\[
c \ e_1 \ldots e_k
\]
\[
\lambda x. e
\]

G.W. Hamilton  Program Transformation
Labelled Transition Systems

G. W. Hamilton  Program Transformation
Labelled Transition Systems

\[
\begin{align*}
\text{case } e_0 & \text{ of } p_1 \Rightarrow e_1 | \ldots | p_k \Rightarrow e_k \\
\text{case } c_1 & \\
\text{case } c_k & \\
\end{align*}
\]
Labelled Transition Systems

\[ \lambda x . e \]

\[ \text{case } e_0 \text{ of } p_1 \Rightarrow e_1 | \ldots | p_k \Rightarrow e_k \]

\[ \text{let } x = e_0 \text{ in } e_1 \]

\[ \text{let } \text{in} \]

G.W. Hamilton Program Transformation
Central concept is that of **driving**, which constructs a potentially infinite **process tree** (similar to LTS), representing all possible computations of the program by normal order reduction.

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

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

A (hopefully) more efficient program can be **residualised** from the resulting folded process tree.
In positive supercompilation, folding is performed when a renaming of a previous term is encountered. This is represented as follows in our LTS framework:
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 **generalisation**.

To represent the result of generalisation, we introduce **generalised states** into our labelled transition systems which have the following form:

\[
\text{let } x = e_0 \text{ in } e_1
\]
Generalisation

An expression $e$ is embedded in expression $e'$ if $e \subseteq e'$:

**Diving**

$$\exists i \in \{1 \ldots n\}. e \subseteq e_i$$

$$e \subseteq \phi(e_1, \ldots, e_n)$$

**Coupling**

$$\forall i \in \{1 \ldots n\}. e_i \subseteq e'_i$$

$$\phi(e_1, \ldots, e_n) \subseteq \phi(e'_1, \ldots, e'_n)$$

We write $e \leq_e e'$ if expression $e$ is coupled with expression $e'$.

$$f_1 x \leq_e f_2 (f_1 y)$$

$$f_1 x \leq_e f_1 (f_2 y)$$

$$f_1 (f_3 x) \leq_e f_1 (f_2 (f_3 y))$$

$$f_1(x, x) \leq_e f_1 (f_2 y, f_2 y)$$

$$f (g x) \not\leq_e f y$$

$$f (g x) \not\leq_e g y$$

$$f (g x) \not\leq_e g (f y)$$

$$f (g x) \not\leq_e f (h y)$$

G.W. Hamilton  
Program Transformation
Generalisation

Definition (Generalisation of Expressions)

The generalisation of expression $e$ with respect to expression $e'$ (denoted by $e \sqcap_e e'$) is defined as shown below.

$$e \sqcap_e e' = \begin{cases} (\phi(e''_1, \ldots, e''_n), \bigcup_{i=1}^n \theta_i, \bigcup_{i=1}^n \theta'_i), & \text{if } \phi = \phi' \\ (\phi(e_1, \ldots, e_n), \bigcup_{i=1}^n \theta_i, \bigcup_{i=1}^n \theta'_i), & \text{where} \\ e = \phi(e_1, \ldots, e_n) \\ e' = \phi'(e'_1, \ldots, e'_n) \\ \forall i \in \{1 \ldots n\}.e_i \sqcap_e e'_i = (e''_i, \theta_i, \theta'_i) \\ (x, \{x \mapsto e\}, \{x \mapsto e'\}), & \text{otherwise} \end{cases}$$

The result of this generalization is a triple $(e''', \theta, \theta')$ where $e'''$ is the generalized expression and $\theta$ and $\theta'$ are substitutions s.t. $e''\theta \equiv e$ and $e'''\theta' \equiv e'$. 
Generalisation

**Definition (Most Specific Generalisation)**

The *most specific generalisation*, denoted by $e \triangle e'$, of expressions $e$ and $e'$ is computed by exhaustively applying the following rewrite rule to the triple obtained from the generalisation $e \sqcap e'$:

$$
\begin{align*}
&\left( e, \\
&\quad \{x_1 \mapsto e', x_2 \mapsto e'\} \cup \theta, \\
&\quad \{x_1 \mapsto e'', x_2 \mapsto e''\} \cup \theta' \right) \\
\Rightarrow &\left( e\{x_1 \mapsto x_2\}, \\
&\quad \{x_2 \mapsto e'\} \cup \theta, \\
&\quad \{x_2 \mapsto e''\} \cup \theta' \right)
\end{align*}
$$

This minimises the substitutions by identifying **common substitutions** which were previously given different names.
### Generalisation

Some examples of **most specific generalisation**:

<table>
<thead>
<tr>
<th>$e$</th>
<th>$e'$</th>
<th>$(e_g, \theta, \theta')$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1 \ x$</td>
<td>$f_2 (f_1 \ y)$</td>
<td>$(v, {v \mapsto f_1 \ x}, {v \mapsto f_2 (f_1 \ y)})$</td>
</tr>
<tr>
<td>$f_1 \ x$</td>
<td>$f_1 (f_2 \ y)$</td>
<td>$(f_1 \ v, {v \mapsto x}, {v \mapsto f_2 \ y})$</td>
</tr>
<tr>
<td>$f_1 (f_3 \ x)$</td>
<td>$f_1 (f_2 (f_3 \ y))$</td>
<td>$(f_1 \ v, {v \mapsto f_3 \ x}, {v \mapsto f_2 (f_3 \ y)})$</td>
</tr>
<tr>
<td>$f_1 (x, x)$</td>
<td>$f_1 (f_2 \ y, f_2 \ y)$</td>
<td>$(f_1 (v, v), {v \mapsto x}, {v \mapsto f_2 \ y})$</td>
</tr>
</tbody>
</table>
Generalisation

To represent the result of generalisation, we use a let construct of the form \texttt{let } x = e_0 \texttt{ in } e_1. This allows the expressions $e_0$ and $e_1$ to be transformed separately.

**Definition (Extraction Operator)**

We define an extraction operator $\triangleleft_e$ on expressions which extracts the sub-terms resulting from generalisation using let expressions as follows:

\[
\emptyset \triangleleft_e e = e \\
(\{x \mapsto e'\} \cup \theta) \triangleleft_e e = \theta \triangleleft_e (\texttt{let } x = e' \texttt{ in } e)
\]
If the current term $e$ has a function as redex, it is processed as follows:

**Algorithm**

- **if** there is a memoised term $e'$ and renaming $\sigma$ s.t. $e'\sigma \equiv e$
  - **then** fold with respect to $e'$
- **else if** there is a memoised term $e'$ and renaming $\sigma$ s.t. $e'\sigma \preceq_e e$
  - **then** generalise with respect to $e'$ and further transform
- **else** memoise $e$, unfold and further transform
Example Program

\[
\text{sum} \ (\text{squares} \ (\text{upto} \ 1 \ n)) \ 0 \\
\text{where} \\
\text{sum} \quad = \lambda x. \lambda a. \ \text{case} \ x \ of \\
\quad \quad | \ [] \quad \Rightarrow \ a \\
\quad \quad | \ x' : x' \Rightarrow \text{sum} \ x' \ (a + x') \\
\text{squares} \quad = \lambda x. \ \text{case} \ x \ of \\
\quad \quad | \ [] \quad \Rightarrow \ [] \\
\quad \quad | \ x' : x' \Rightarrow (\text{square} \ x') : (\text{squares} \ x') \\
\text{upto} \quad = \lambda m. \lambda n. \ \text{case} \ (m > n) \ of \\
\quad \quad | \ True \quad \Rightarrow \ [] \\
\quad \quad | \ False \Rightarrow m : (\text{upto} \ (m + 1) \ n)
\]
Driving

\[ \text{sum} \left( \text{squares} \left( \text{upto} \ 1 \ n \right) \right) \ 0 \]
Driving

\[ \text{sum (squares (upto 1 n)) 0} \]

\[ \text{case} \]

\[ 1 > n \]
Driving

\[ \text{sum}(\text{squares}(\text{upto } 1 \ n)) \ 0 \]

\[ \text{case} \]

\[ 1 > n \]

\[ > \]
Driving

\[
\text{case}
\]

\[
\text{sum}\left(\text{squares}\left(\text{upto} \ 1 \ n\right)\right) \ 0
\]

\[
1 > n
\]

\[
0
\]
Driving

\[\text{sum (squares (upto 1 \ n)) 0}\]

\[
\text{case} \ \\
1 > n \\
@ \ \\
> \ \\
0
\]

G.W. Hamilton  
Program Transformation
Driving

```
sum (squares (upto 1 n)) 0
```

```
case

1 > n

@ #1

> 1

> 1

0 0
```
Driving

```
sum (squares (upto 1 n)) 0
```

```
case

1 > n

@  #1 #2

>  1  n

>  1

0  0
```
Driving

\[ \text{sum (squares (upto 1 n)) 0} \]

\[
\text{case } 1 > n \\
\text{@ } 1 \text{ } 2 \\
> \text{ } 1 \text{ } n \\
0 \text{ } 0 \text{ } 0
\]
Driving

\[
\text{sum} \ (\text{squares} \ (\text{upto} \ 1 \ n)) \ 0
\]

\[
\text{case} \ True
\]

1 > n

0

> 1 n

@ #1 #2

> 1 n

0 0 0

G.W. Hamilton  Program Transformation
Driving

\[
\text{case} \quad \text{True} \quad \text{of} \quad 1 > n \quad \text{sum (squares (upto 1 n)) 0}
\]

Diagram:
- Case 1 > n
  - 0
  - 1
  - n
- Case True
  - 0

G.W. Hamilton
Program Transformation
Driving

\[
\text{sum (squares (upto 1 n)) 0}
\]

\[
\begin{align*}
\text{case} & \quad \text{True} & \quad \text{False} \\
1 > n & \quad 0 & \quad \text{sum (squares (upto (1 + 1) n)) (0 + (square 1))}
\end{align*}
\]

G.W. Hamilton  Program Transformation
Driving

\[ \text{sum (squares (upto 1 n)) 0} \]

\[ \text{case} \quad \begin{cases} \text{True} & \quad 1 > n \\ \text{False} & \quad 0 \end{cases} \]

\[ \begin{array}{c}
\text{>} \\
\text{>} \\
\text{>}
\end{array} \quad \begin{array}{c}
1 \\
1 \\
n
\end{array} \quad \begin{array}{c}
n \\
\text{0}
\end{array} \]

\[ \begin{array}{c}
\text{0} \\
\text{0} \\
\text{0}
\end{array} \]

\[ \text{sum (squares (upto (1 + 1) n)) (0 + (square 1))} \]
Generalisation and Further Driving

\[
\text{let } x_1 = 1 \text{ in let } x_2 = 0 \text{ in sum (squares (upto } x_1 n \text{)) } x_2
\]
Generalisation and Further Driving

\[
\text{let } x_1 = 1 \text{ in let } x_2 = 0 \text{ in sum (squares (upto } x_1 \ n) \ x_2
\]

let

1
Generalisation and Further Driving

\[
\text{let } x_1 = 1 \text{ in let } x_2 = 0 \text{ in sum (squares (upto } x_1 \ n)) \ x_2
\]
Generalisation and Further Driving

```plaintext
let \( x_1 = 1 \) in let \( x_2 = 0 \) in sum (squares (upto \( x_1 \) \( n \))) \( x_2 \)
```
Generalisation and Further Driving

\[
\text{let } x_1 = 1 \text{ in let } x_2 = 0 \text{ in sum (squares (upto } x_1 \ n)) \ x_2
\]
Generalisation and Further Driving

\[
\text{let } x_1 = 1 \text{ in let } x_2 = 0 \text{ in sum (squares (upto } x_1 n)) x_2
\]
Generalisation and Further Driving

```plaintext
let x_1 = 1 in let x_2 = 0 in sum (squares (upto x_1 n)) x_2
```

1

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
</tr>
</thead>
</table>

\[ x_1 > n \]
Generalisation and Further Driving

\[
\text{let } x_1 = 1 \text{ in let } x_2 = 0 \text{ in sum \( (\text{squares \( \text{upto } x_1 \) \( n \)}) \times x_2 \)}
\]

Case: \( x_1 > n \)
Generalisation and Further Driving

let \( x_1 = 1 \) in let \( x_2 = 0 \) in \( \text{sum} (\text{squares} (\text{upto} \ x_1 \ n)) \ x_2 \)

```
let x_1 = 1 in let x_2 = 0 in sum (squares (upto x_1 n)) x_2
```

1

<table>
<thead>
<tr>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

```
x_1 > n
```

> 0
Generalisation and Further Driving

```
let x_1 = 1 in let x_2 = 0 in sum (squares (upto x_1 n)) x_2
```

\[
\text{let } x_1 = 1 \text{ in let } x_2 = 0 \text{ in sum (squares (upto } x_1 \text{ n)) } x_2
\]
Generalisation and Further Driving

\[
\text{let } x_1 = 1 \text { in let } x_2 = 0 \text { in sum (squares (upto } x_1 n) \times x_2)
\]
Generalisation and Further Driving


def f(x_1, x_2):
    if x_1 > n:
        case:
            > 1
            ≠ 1
            ≠ 2
            >
            x_1
            n
            0
            0
            0
            0
            sum (squares (upto x_1 n)) x_2

    let x_1 = 1 in let x_2 = 0 in sum (squares (upto x_1 n)) x_2
Generalisation and Further Driving

\[
\text{let } x_1 = 1 \text{ in let } x_2 = 0 \text{ in } \text{sum } \left( \text{squares } \left( \text{upto } x_1 n \right) \right) x_2
\]
Generalisation and Further Driving

let $x_1 = 1$ in let $x_2 = 0$ in sum (squares (upto $x_1$ $n$)) $x_2$

G.W. Hamilton
Program Transformation
Generalisation and Further Driving

```plaintext
let x_1 = 1 in let x_2 = 0 in sum (squares (upto x_1 n)) x_2
```

```plaintext
let x_1 = 1 in let x_2 = 0 in sum (squares (upto x_1 n)) x_2
```

1

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0
Generalisation and Further Driving

\[
\text{let } x_1 = 1 \text{ in let } x_2 = 0 \text{ in } \text{sum} \left( \text{squares} \left( \text{upto} \ x_1 \ n \right) \right) \times x_2
\]

\[
\text{let } x_1 = 1 \text{ in let } x_2 = 0 \text{ in } \text{sum} \left( \text{squares} \left( \text{upto} \ x_1 \ n \right) \right) \times x_2
\]

\[
\text{let } x_1 = 1 \text{ in let } x_2 = 0 \text{ in } \text{sum} \left( \text{squares} \left( \text{upto} \ x_1 \ n \right) \right) \times x_2
\]

\[
\text{let } x_1 = 1 \text{ in let } x_2 = 0 \text{ in } \text{sum} \left( \text{squares} \left( \text{upto} \ x_1 \ n \right) \right) \times x_2
\]

\[
\text{let } x_1 = 1 \text{ in let } x_2 = 0 \text{ in } \text{sum} \left( \text{squares} \left( \text{upto} \ x_1 \ n \right) \right) \times x_2
\]

\[
\text{let } x_1 = 1 \text{ in let } x_2 = 0 \text{ in } \text{sum} \left( \text{squares} \left( \text{upto} \ x_1 \ n \right) \right) \times x_2
\]

\[
\text{let } x_1 = 1 \text{ in let } x_2 = 0 \text{ in } \text{sum} \left( \text{squares} \left( \text{upto} \ x_1 \ n \right) \right) \times x_2
\]

\[
\text{let } x_1 = 1 \text{ in let } x_2 = 0 \text{ in } \text{sum} \left( \text{squares} \left( \text{upto} \ x_1 \ n \right) \right) \times x_2
\]

\[
\text{let } x_1 = 1 \text{ in let } x_2 = 0 \text{ in } \text{sum} \left( \text{squares} \left( \text{upto} \ x_1 \ n \right) \right) \times x_2
\]
Generalisation and Further Driving

\[
\text{let } x_1 = 1 \text{ in let } x_2 = 0 \text{ in } \text{sum } (\text{squares } (\text{upto } x_1 n)) \times x_2
\]

\[
\begin{align*}
\text{let} & \quad \text{let} \\
1 & \quad 0 \\
0 & \quad 0 \\
\text{case} & \quad \text{True} \quad \text{False} \\
x_1 > n & \quad x_2 \\
> & \quad x_1 \\
& \quad n \\
0 & \quad 0 \\
\end{align*}
\]

\[
\text{sum } (\text{squares } (\text{upto } x_1 n)) \times x_2 \\
\text{sum } (\text{squares } (\text{upto } (x_1 + 1) n)) \times (x_2 + (x_1^2))
\]
Generalisation and Further Driving

```
let x_1 = 1 in let x_2 = 0 in sum (squares (upto x_1 n)) x_2
```

```
let x_3 = x_1 + 1 in let x_4 = x_2 + (x_1^2) in (sum (squares (upto x_3 n)) x_4)
```
Generalisation and Further Driving

```latex
\begin{align*}
\text{let } x_1 &= 1 \text{ in let } x_2 &= 0 \text{ in sum \{squares \{upto x_1 \ n\}\} \ x_2} \\
\text{let } x_1 > n \text{ in case } \begin{cases} \\
\text{True} & \text{if } x_1 > n \Rightarrow 0 \\
\text{False} & \text{if } x_1 \leq n \Rightarrow 0 \\
\end{cases} \\
\text{let } x_3 &= x_1 + 1 \text{ in let } x_4 &= x_2 + (x_1^2) \text{ in (sum \{squares \{upto x_3 \ n\}\} \ x_4} \\
\end{align*}
```
Generalisation and Further Driving

let \( x_1 = 1 \) in let \( x_2 = 0 \) in \( \text{sum (squares (upto } x_1 \text{ n))} \times x_2 \)

let \( x_3 = x_1 + 1 \) in let \( x_4 = x_2 + (x_1^2) \) in \( \text{sum (squares (upto } x_3 \text{ n))} \times x_4 \)
Generalisation and Further Driving

\[
\begin{align*}
\text{let } x_1 &= 1 \text{ in let } x_2 = 0 \text{ in sum (squares (upto } x_1 \text{ )} ) x_2 \\
\text{let } x_1 > n \text{ in } & \quad \begin{cases} \\
\text{True: } & \text{let } x_3 = x_1 + 1 \text{ in let } x_4 = x_2 + (x_1^2) \text{ in (sum (squares (upto } x_3 \text{ )} ) x_4} \\
\text{False: } & \text{let } x_3 = x_1 + 1 \end{cases} \\
\text{let } x_1 + 1 \text{ in } & \\
\end{align*}
\]
Generalisation and Further Driving

\[
\text{let } x_1 = 1 \text{ in let } x_2 = 0 \text{ in } \text{sum (squares (upto } x_1 \text{ n)) } x_2
\]

\[
\text{let } x_1 > n \text{ in case } \begin{cases} \text{True} & \text{let } x_3 = x_1 + 1 \text{ in let } x_4 = x_2 + (x_1^2) \text{ in (sum (squares (upto } x_3 \text{ n)) } x_4 \\ \text{False} & \end{cases}
\]

\[
\text{let } x_1 + 1 \text{ in ...
\]
Generalisation and Further Driving

\[
\text{let } x_1 = 1 \text{ in let } x_2 = 0 \text{ in sum (squares (upto } x_1 n) \times x_2)
\]

\[
\text{let } x_3 = x_1 + 1 \text{ in let } x_4 = x_2 + (x_1^2) \text{ in sum (squares (upto } x_3 n) \times x_4)
\]
Generalisation and Further Driving

let \( x_1 = 1 \) in let \( x_2 = 0 \) in sum (squares (upto \( x_1 \) \( n \))) \( x_2 \)

let \( x_3 = x_1 + 1 \) in let \( x_4 = x_2 + (x_1^2) \) in (sum (squares (upto \( x_3 \) \( n \))) \( x_4 \))

G.W. Hamilton  Program Transformation
Generalisation and Further Driving

\[
\text{let } x_1 = 1 \text{ in let } x_2 = 0 \text{ in } \text{sum } (\text{squares } (\text{upto } x_1 n)) \times x_2
\]

\[
\text{let } x_3 = x_1 + 1 \text{ in let } x_4 = x_2 + (x_1^2) \text{ in } \text{sum } (\text{squares } (\text{upto } x_3 n)) \times x_4
\]
Generalisation and Further Driving

let \( x_1 = 1 \) in let \( x_2 = 0 \) in \( \text{sum (squares (upto } x_1 n) \times x_2} \)

let \( x_3 = x_1 + 1 \) in let \( x_4 = x_2 + (x_1^2) \) in \( \text{sum (squares (upto } x_3 n) \times x_4} \)
Generalisation and Further Driving

let $x_1 = 1$ in let $x_2 = 0$ in sum (squares ( upto $x_1$ $n$ )) $x_2$

let $x_3 = x_1 + 1$ in let $x_4 = x_2 + (x_1^2)$ in (sum (squares ( upto $x_3$ $n$ ))) $x_4$
Generalisation and Further Driving

\[
\text{let } x_1 = 1 \text{ in let } x_2 = 0 \text{ in sum (squares (upto } x_1 n) x_2
\]

\[
\text{let } x_3 = x_1 + 1 \text{ in let } x_4 = x_2 + (x_1^2) \text{ in (sum (squares (upto } x_3 n) x_4
\]

G.W. Hamilton  Program Transformation
Generalisation and Further Driving

\[
\text{let } x_1 = 1 \text{ in let } x_2 = 0 \text{ in } \text{sum} \ (\text{squares (upto } x_1 \ n)) \ x_2
\]

\[
\text{let } x_1 = 1 \text{ in let } x_2 = 0 \text{ in } \text{sum} \ (\text{squares (upto } x_1 \ n)) \ x_2
\]

\[
\text{let } x_3 = x_1 + 1 \text{ in let } x_4 = x_2 + (x_1^2) \text{ in } (\text{sum} \ (\text{squares (upto } x_3 \ n)) \ x_4)
\]

\[
\text{let } x_1 = 1 \text{ in let } x_2 = 0 \text{ in } \text{sum} \ (\text{squares (upto } x_1 \ n)) \ x_2
\]

\[
\text{let } x_3 = x_1 + 1 \text{ in let } x_4 = x_2 + (x_1^2) \text{ in } (\text{sum} \ (\text{squares (upto } x_3 \ n)) \ x_4)
\]
Generalisation and Further Driving

let \( x_1 = 1 \) in let \( x_2 = 0 \) in sum (squares (upto \( x_1 \) \( n \))) \( x_2 \)

\[
\begin{align*}
\text{let } x_1 = 1 \text{ in let } x_2 = 0 \text{ in } & \text{sum (squares (upto } x_1 \text{ } n)) \text{ } x_2 \\
& \text{case } x_1 > n \\
& \text{let } x_3 = x_1 + 1 \text{ in let } x_4 = x_2 + (x_1^{\wedge} 2) \text{ in } \text{sum (squares (upto } x_3 \text{ } n)) \text{ } x_4
\end{align*}
\]
Generalisation and Further Driving

\[
\text{let } x_1 = 1 \text{ in let } x_2 = 0 \text{ in sum (squares (upto } x_1 \text{ } n) \times x_2)
\]

Case True:
\[
\text{let } x_3 = x_1 + 1 \text{ in let } x_4 = x_2 + (x_1^2) \text{ in (sum (squares (upto } x_3 \text{ } n)) \times x_4)
\]

Case False:
\[
\text{let } x_3 = x_1 + 1 \text{ in let } x_4 = x_2 + (x_1^2) \text{ in (sum (squares (upto } x_3 \text{ } n)) \times x_4)
\]
Generalisation and Further Driving

```
let x_1 = 1 in let x_2 = 0 in sum (squares (upto x_1 n)) x_2
```

```
let x_3 = x_1 + 1 in let x_4 = x_2 + (x_1^2) in (sum (squares (upto x_3 n)) x_4)
```

G.W. Hamilton
Program Transformation
Generalisation and Further Driving

\[
\text{let } x_1 = 1 \text{ in let } x_2 = 0 \text{ in } \text{sum } (\text{squares } (\text{upto } x_1 \ n)) \times x_2
\]

\[
\text{let } x_3 = x_1 + 1 \text{ in let } x_4 = x_2 + (x_1^2) \text{ in } \text{sum } (\text{squares } (\text{upto } x_3 \ n)) \times x_4
\]
let $x_1 = 1$ in let $x_2 = 0$ in \( \text{sum (squares (upto $x_1$ $n$))} \times x_2 \)
Generalisation and Further Driving

```
let x1 = 1 in let x2 = 0 in sum (squares (upto x1 n)) x2
```

```
let x3 = x1 + 1 in let x4 = x2 + (x1^2) in (sum (squares (upto x3 n)) x4)
```

```
sum (squares (upto x3 n)) x4
```
Residualization

Resulting Program

```ml
let x₁ = 1
in let x₂ = 0
  in f x₁ x₂ n
  where
  f = λx₁.λx₂.λn.case (x₁ > n) of
    True ⇒ x₂
    | False ⇒ let x₃ = x₁ + 1
      in let x₄ = x₂ + x₁^2
          in f x₃ x₄ n
```

G.W. Hamilton  Program Transformation
In order to prove that positive supercompilation terminates, we need to prove the following:

**Lemma (Transformation Steps of Positive Supercompilation)**

Every infinite sequence of transformation steps must include *function unfolding*.

This would not normally be the case, for example:

\[(\lambda x \to x \ x) \ (\lambda x \to x \ x)\]

We therefore require that all \(\lambda\)-abstractions are replaced by a freshly named function prior to their substitution.
We then need to show that the embedding relation $\preceq_e$ is a well-quasi order.

**Definition (Well-Quasi Order)**

A well-quasi order on a set $S$ is a reflexive, transitive relation $\leq$ such that for any infinite sequence $s_1, s_2, \ldots$ of elements from $S$ there are numbers $i, j$ with $i < j$ and $s_i \leq s_j$.

**Lemma ($\preceq_e$ is a Well-Quasi Order)**

The embedding relation $\preceq_e$ is a well-quasi order on any sequence of expressions that are encountered during transformation.
Proof

The proof is by **contradiction**:

- If the positive supercompilation algorithm did not terminate then the set of memoised expressions must be infinite, since every infinite sequence of transformation steps must include function unfolding.

- Every new expression which is memoised cannot have any of the previously memoised expressions embedded within it by the homeomorphic embedding relation $\preceq_e$, since folding or generalisation would have been performed instead.

- This contradicts the fact that $\preceq_e$ is a well-quasi-order.
Correctness

In order to prove the correctness of positive supercompilation, we need to show that the result of transformation is equivalent to the original program.

Definition (Observational Equivalence)

**Observational equivalence**, denoted by $\simeq$, equates two expressions if and only if they exhibit the same termination behaviour in all closing contexts i.e. $e_1 \simeq e_2$ iff $\forall C. (C[e_1] \downarrow$ iff $C[e_2] \downarrow)$.

Theorem (Correctness of Positive Supercompilation)

$$S[e] = t \Rightarrow e \simeq R[t]$$

The proof of this is by co-induction on the structure of the LTS $t$. 
Efficiency

- Without the additional rewrite rule to identify common substitutions, only linear improvements are possible.
- Even using this rewrite rule, superlinear improvements are obtained for very few useful programs.
- Consider the following example program:

\[
f \equiv \lambda x.\lambda y.\text{case } x \text{ of } \begin{align*}
\text{Zero} & \Rightarrow y \\
\text{Succ}(x) & \Rightarrow f(f x x)(f x x)
\end{align*}
\]

This program takes exponential time \(O(2^n)\), where \(n\) is the size of the variable \(x\).
Efficiency

During transformation of the previous program by positive supercompilation, the term corresponding to \((f \times x)\) is extracted twice, but then identified by rewriting to obtain the most specific generalization.

The program is transformed to the following:

\[
f' \ x \\
\text{where} \\
f' = \lambda x.\text{case } x \text{ of} \\
\quad Zero \Rightarrow Zero \\
\mid \text{Succ}(x) \Rightarrow f' (f' x)
\]

This program takes linear time \(O(n)\) on the same input, so a superlinear speedup has been achieved.
We would expect to be able to transform the following:

**Naive Reverse**

\[ nrev \ vs \]

where

\[ nrev = \lambda x.\text{case } x \text{ of} \]

\[ [] \Rightarrow [] \]

\[ | x' : x' \Rightarrow \text{app } (nrev x') [x'] \]

\[ \text{app } = \lambda x.\lambda y.\text{case } x \text{ of} \]

\[ [] \Rightarrow y \]

\[ | x' : x' \Rightarrow x' : (\text{app } x' \ y) \]
Distillation

Into the following:

Accumulating Reverse

\[ \text{arev vs} \]

\textbf{where}

\[ \text{arev} = \lambda xs. \text{arev}' \, xs \, [] \]

\[ \text{arev}' = \lambda xs. \lambda ys. \text{case} \, xs \, \text{of} \]

\[ [] \Rightarrow ys \]

\[ | x' : xs' \Rightarrow \text{arev}' \, xs' \, (x' : ys) \]

But we cannot do this using positive supercompilation.
Distillation

- Many sub-terms extracted by generalisation during positive supercompilation may actually be intermediate structures e.g., the recursive call to the nrev function.
- In distillation, such sub-terms are substituted back and further transformed; only sub-terms which are generalised by distillation are permanently extracted.
- We add blazing to let terms (of the form let\(\ominus\)) that are the result of generalisation by distillation.
- let terms that are not blazed are the result of generalisation by positive supercompilation.
Central concept is also driving to construct a LTS representing all possible computations of the program by normal order reduction.

The terms in the nodes of this LTS which have a function as redex are transformed by positive supercompilation to obtain their corresponding LTS representation.

Generalisation and folding are performed with respect to these LTSs.

Can produce many more superlinear speedups in programs that are not produced by positive supercompilation.

For example, the naive reverse example from earlier can be transformed into accumulating reverse.
Bisimulation

In distillation, folding is performed when the LTS produced by supercompiling a term is a bisimulation of a previous one.

**Definition (Simulation)**

A binary relation $R \subseteq \mathcal{E} \times \mathcal{E}'$ is a simulation of labelled transition systems $t = (\mathcal{E}, e_0, \text{Act}, \rho)$ by $t' = (\mathcal{E}', e'_0, \text{Act}', \rho')$ if $(e_0, e'_0) \in R$, and for every pair $(e_i, e'_i) \in R$ the following holds:

$$\forall e_j \in \mathcal{E} \text{ s.t. } (e_i \xrightarrow{\alpha} e_j) \in \rho. (\exists e'_j \in \mathcal{E}' \text{ s.t. } (e'_i \xrightarrow{\alpha} e'_j) \in \rho'. (e_j, e'_j) \in R)$$

**Definition (Bisimulation)**

A bisimulation $\sim$ is a binary relation $R$ such that both $R$ and its inverse $R^{-1}$ are simulations.
LTS Embedding

In distillation, **generalisation** is performed when the LTS produced by supercompiling a term is an **embedding** of a previous one.

**Definition (LTS Embedding)**

A binary relation $\mathcal{R} \subseteq \mathcal{E} \times \mathcal{E}'$ is an **embedding** of labelled transition system $t = (\mathcal{E}, e_0, \text{Act}, \rho)$ by $t' = (\mathcal{E}', e'_0, \text{Act}', \rho')$ if $(e_0, e'_0) \in \mathcal{R}$, and for every pair $(e_i, e'_i) \in \mathcal{R}$ one of the following holds:

1. $\forall e_j \in \mathcal{E} \text{ s.t. } (e_i \xrightarrow{\alpha} e_j) \in \rho. (\exists e'_j \in \mathcal{E}' \text{ s.t. } (e'_i \xrightarrow{\alpha} e'_j) \in \rho'. (e_j, e'_j) \in \mathcal{R})$

2. $\exists e'_j \in \mathcal{E}' \text{ s.t. } (e'_i \xrightarrow{\alpha} e'_j) \in \rho'. (e_i, e'_j) \in \mathcal{R}$

We write $t \preceq_t t'$ if LTS $t$ is coupled with LTS $t'$. 
Defintion (Generalisation of LTSs)

The generalisation of LTS \( t \) with respect to LTS \( t' \) (denoted by \( t \sqcap_t t' \)) is defined as shown below.

\[
\begin{align*}
(e \to (\text{let}, t_0), (\text{in}, t_1)) \sqcap_t (e' \to (\text{let}, t'_0), (\text{in}, t'_1)) &= e \to (\text{let}, t_0), (\text{in}, \theta \triangleleft_t t'_1) \\
\text{where } t_1 \sqcap_t t'_1 &= (t''_1, \theta)
\end{align*}
\]

\[
\begin{align*}
t \sqcap_t t' &= \begin{cases}
(e \to (\alpha_1, t''_1), \ldots, (\alpha_n, t''_n), \bigcup_{i=1}^n \theta_i), & \text{if } \forall i. \alpha_i = \alpha'_i \\
(t = (e \to (\alpha_1, t_1), \ldots, (\alpha_n, t_n)) \\
(t' = (e' \to (\alpha'_1, t'_1), \ldots, (\alpha'_n, t'_n)) \\
\forall i \in \{1 \ldots n\}. t_i \sqcap_t t'_i = (t''_i, \theta_i) \\
(x \to (x, 0), \{x \mapsto t\}), & \text{otherwise}
\end{cases}
\end{align*}
\]

Common substitutions are also identified.
LTS Generalisation

To represent the result of generalisation in distillation, we use the let⁻ construct of the form let⁻ \( x = e_0 \) in \( e_1 \). This represents the permanent extraction of the expression \( e_0 \).

**Definition (LTS Extraction Operator)**

The result of generalisation is made into a nested let⁻ using \( \theta \triangleleft_t t \) for substitution \( \theta \) and LTS \( t \), which is defined as follows:

\[
\emptyset \triangleleft_t t = t \\
\left( \{ x \mapsto t' \} \cup \theta \right) \triangleleft_t t = \\
\theta \triangleleft_t ((\text{let}^- x = \text{root}(t') \text{ in root}(t)) \rightarrow (\text{let}^-, t'), (\text{in}, t))
\]
If the current term $e$ has a function as redex, and the result of applying positive supercompilation to this term is the LTS $t$, then this is processed as follows:

**Algorithm**

```
if there is a memoised LTS $t'$ and renaming $\sigma$ s.t. $t'\sigma \sim t$
then fold with respect to $t'$
else if there is a memoised LTS $t'$ and renaming $\sigma$ s.t. $t'\sigma \preceq_t t$
then generalise with respect to $t'$, residualise and transform
else memoise LTS $t$, unfold $e$ and further transform
```
Distillation Example

Consider the naive reverse example from earlier:

### Naive Reverse

\[ nrev \ vs \]

**where**

\[ nrev = \lambda xs. \text{case} \; xs \; \text{of} \]

\[
\begin{align*}
& \; [] \quad \Rightarrow \; [] \\
& \; | \; x' : xs' \quad \Rightarrow \; \text{app} \; (nrev \; xs') \; [x']
\end{align*}
\]

\[ \text{app} = \lambda xs. \lambda ys. \text{case} \; xs \; \text{of} \]

\[
\begin{align*}
& \; [] \quad \Rightarrow \; ys \\
& \; | \; x' : xs' \quad \Rightarrow \; x' : (\text{app} \; xs' \; ys)
\end{align*}
\]
Distillation Example

During transformation, the following LTS is encountered:

\[
\text{let } \text{in} \\
\text{nrev } xs' \\
\text{case} \\
\text{vs } \\
\text{#1 } \\
\text{#2 } \\
\equiv \\
\text{x' } \\
\text{v' } \\
\text{case} \\
\text{vs' } \\
\text{#1#2 } \\
\text{x' } \\
\text{v' }
\]
Later, the following LTS is encountered:

```
let
in
case vs []:
  x''
  #1 #2
  x'
  #1 #2
  v'
  #1
  #2
  case vs' []:
    x''
    #1 #2
    x'
    #1 #2
    vs'
    #1 #2
    x':[]
    #1
    #2
    v'
```

G.W. Hamilton  Program Transformation
This is an embedding:

\[
\text{let } \text{in}
\]

\[
\text{case vs }\[
\]
\]

\[
\text{case vs }\[
\]
\]

\[
\text{case vs }\[
\]
\]

\[
\equiv
\]

G.W. Hamilton  Program Transformation
Generalisation is performed to replace this term with $w$:
Distillation Example

Later, the following LTS is encountered:

\[
\text{let } \ nrev \ xs'' \ \text{in}
\]

\[
\begin{cases}
  \text{vs} :& \#1 \ x'' \ #2 \\
  \text{vs}' :& \#1 \ x'' \ #2 \\
  \text{v} :& \#1 \ v'' \ #2 \\
  \text{v}' :& \#1 \ v'' \ #2
\end{cases}
\]
Distillation Example

This is also an embedding:

```
let in
  case
  vs [] #1
  x'' #2
  w : #1 #2
  case vs' [] #1
  x' w
  x'' #1 #2
  v'
  vs' #1 #2
  x':w
  v''
```

G.W. Hamilton  Program Transformation
Distillation Example

Generalisation is again performed to replace this term with $w'$:
This is a renaming, so folding is performed:
This is residualised to the following program:

```
case xs of
  [] ⇒ []
  | x' : xs' ⇒ let w = []
      in f x' xs' w

where
f = λx'.λxs'.λw. case xs' of
  [] ⇒ x' : w
  | x'' : xs'' ⇒ let w' = x' : w
      in f x'' xs'' w'
```

This program has a run-time which is linear with respect to the length of the input list, while the original program is quadratic.
Further Distillation Example

The \textit{fib} function:

\begin{verbatim}
\begin{verbatim}
\textbf{fib} \textit{n}
\textbf{where}
\textbf{fib} = \lambda n.\textbf{case} \textit{n} \textbf{of}
\hspace{1cm} \textbf{Zero} \Rightarrow \textbf{Succ} \textbf{Zero}
\hspace{1cm} | \hspace{1cm} \textbf{Succ} \textit{n'} \Rightarrow \textbf{case} \textit{n'} \textbf{of}
\hspace{2cm} \textbf{Zero} \Rightarrow \textbf{Succ} \textbf{Zero}
\hspace{3cm} | \hspace{3cm} \textbf{Succ} \textit{n''} \Rightarrow (\textbf{fib} \textit{n'}) + (\textbf{fib} \textit{n''})
\end{verbatim}
\end{verbatim}

is transformed to something similar to the following by distillation:

\begin{verbatim}
\begin{verbatim}
\textbf{f} \textit{n} (\textbf{Succ} \textbf{Zero}) (\textbf{Succ} \textbf{Zero})
\textbf{where}
\textbf{f} = \lambda n.\lambda x.\lambda y.\textbf{case} \textit{n} \textbf{of}
\hspace{1cm} \textbf{Zero} \Rightarrow x
\hspace{1cm} | \hspace{1cm} \textbf{Succ} \textit{n'} \Rightarrow f \textit{n'} y (x + y)
\end{verbatim}
\end{verbatim}

G.W. Hamilton  Program Transformation
The output of distillation will be in the following form:

**Distilled Form**

| prog | ::= | \(e_0^\emptyset\) where \(f_1 = e_1^\emptyset\ldots f_k = e_k^\emptyset\) |
| \(e^\rho\) | ::= | x \(e_1^\rho\ldots e_n^\rho\) |
|      |      | c \(e_1^\rho\ldots e_n^\rho\) |
|      |      | \(\lambda x . e^\rho\) |
|      |      | \(f \; x_1 \ldots x_n\) |
|      |      | case \((x \; e_1^\rho \ldots e_n^\rho)\) of \(p_1 \Rightarrow e_{n+1}^\rho\) \cdots | \(p_k \Rightarrow e_{n+k}^\rho,\) |
|      |      | x \(\notin \rho\) |
|      |      | let \(x = e_0^\rho\) in \(e_1^{\rho \cup \{x\}}\) |
Distillation

Distilled Form

- Distilled form is a simplified form that makes it easier to analyse and reason about.
- Many possible applications:
  - **Program Construction**: G.W. Hamilton and M.H. Kabir, “Constructing Programs From Metasystem Transition Proofs”, META 2008
  - **Program Parallelisation**: V. Kannan and G.W. Hamilton, “Program Transformation to Identify List-Based Parallel Skeletons”, *EPTCS* 216: 118-136, 2016
The proof of termination for distillation is similar to that for positive supercompilation; we need to show that the embedding relation $\preceq_t$ is a well-quasi order.

**Lemma ($\preceq_t$ is a Well-Quasi Order)**

The embedding relation $\preceq_t$ is a well-quasi order on any sequence of LTSs that are encountered during transformation.
The proof of correctness for distillation is again similar to that for positive supercompilation; we need to show that the result of transformation is equivalent to the original program.

**Theorem (Correctness of Distillation)**

\[ D[e] = t \implies e \simeq R[t] \]

The proof of this is again by co-induction on the structure of the LTS \( t \).
Efficiency

- Speedups are obtained in positive supercompilation and distillation through two mechanisms:
  1. performing reductions
  2. identification of extracted components resulting from generalization
- Identification of extracted components can always give a superlinear speedup.
- In positive supercompilation, there can only be a constant number of reductions between each recursive call of a function.
  - Removing these will therefore only give a linear speedup.
- In distillation, the number of reductions between each recursive call of a function can be increasing.
  - Removing these can therefore give a superlinear speedup.
Conclusions

- We have presented both positive supercompilation and distillation in our LTS framework.
- The algorithms are very similar; distillation just works at the level above positive supercompilation.
- Distillation extends the range of transformations which can be performed beyond those which can be performed by positive supercompilation.
- The extra power of distillation is obtained by removing computations from within recursive terms.
- The extra power of distillation comes at a price; there can be an exponential increase in the number of steps required in the worst case.
Some semi-automatic techniques which also work at the level above supercompilation and are capable of obtaining the same superlinear speedups as distillation are as follows:

- Walk grammars (Turchin)
- Second-order replacement (Kott)
- Higher-level supercompilation (Klyuchnikov)

However:
- usually require eureka steps
- often need to make use of specific laws

Distillation is a fully automatic technique capable of obtaining these superlinear speedups.