A Hierarchy of Program Transformers

G.W. Hamilton

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

META 2012

Outline

1. Background
2. Language
3. Labelled Transition Systems
4. Hierarchy of Program Transformers
5. Distillation
6. Conclusions

Background

- This work is inspired by the supercompilation transformation algorithm developed by Turchin.
- Supercompilation became more widely known through the positive supercompilation algorithm (Sørensen, Glück and Jones).
- Positive supercompilation can only produce a linear speedup in programs (Sørensen).
- More recently, the distillation algorithm was propsed, which can produce a superlinear improvement in programs.

Background

- Positive supercompilation uses the expressions encountered during transformation to determine when to perform generalization and folding.
- Distillation as originally devised (PEPM 2007) used the results obtained by positive supercompilation to determine when to perform generalization and folding.
- We could envisage another level on top of this which uses the results of this transformation to determine when to perform generalization and folding.
- This suggests the existence of a hierarchy of levels of program transformation.

Language Syntax

We use the following higher-order functional language:

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

Example Program

$$
\begin{array}{l}
\mathit{fib}\ n \\
\textbf{where} \\
\mathit{fib} \quad = \quad \lambda n.\textbf{case}\ n\ \textbf{of} \\
\qquad\qquad\qquad\qquad Z \quad \Rightarrow S\ Z \\
\qquad\qquad\qquad\quad\ \mid S\ n' \Rightarrow \textbf{case}\ n'\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad Z \quad \Rightarrow S\ Z \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid S\ n'' \Rightarrow +\ (\mathit{fib}\ n'')\ (\mathit{fib}\ n')
\end{array}
$$

Reduction Rules

$$((\lambda x.e_0)\ e_1) \overset{\beta}{\leadsto} (e_0\{x \mapsto e_1\}) \qquad (\textbf{let}\ x = e_0\ \textbf{in}\ e_1) \overset{\beta}{\leadsto} (e_1\{x \mapsto e_0\})$$

$$\frac{f = e}{f \overset{f}{\leadsto} e} \qquad \frac{e_0 \overset{r}{\leadsto} e_0'}{(e_0\ e_1) \overset{r}{\leadsto} (e_0'\ e_1)}$$

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

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

Labelled Transition Systems for Language

Similarly to Gordon, we use labelled transition systems to characterise the run-time behaviour of a program.

- We extend the work of Gordon by the inclusion of free variables.

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

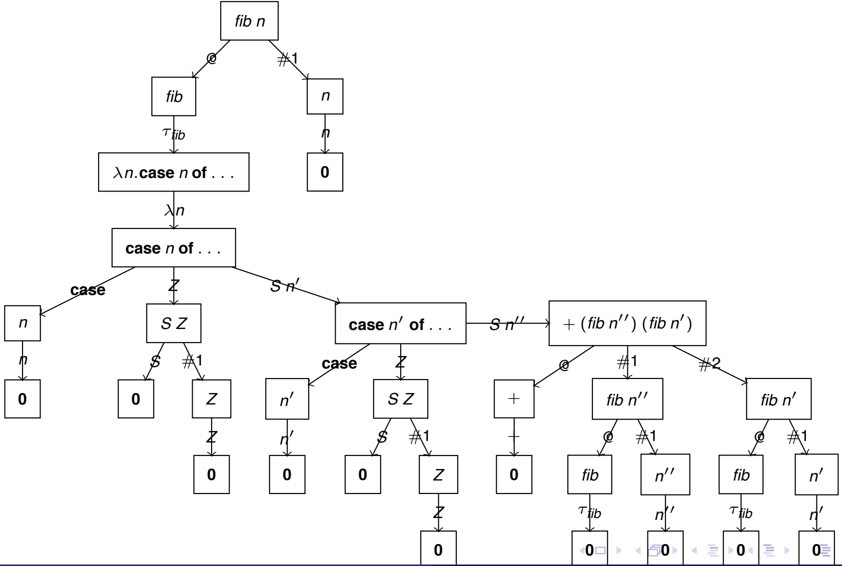
- $\mathcal{E}$ is the set of states of the LTS.
  - Each is an expression, or the end-of-action state **0**.
- e_0 is the start state
- Act is a set of actions α that can be silent (τ) or non-silent.
- $\rightarrow \subseteq \mathcal{E} \times Act \times \mathcal{E}$ is a transition relation that relates pairs of states by actions.
  - If $e \in \mathcal{E}$ and $(e, \alpha, e') \in \rightarrow$ then $e' \in \mathcal{E}$.

Labelled Transition Systems for Language

Actions:

x	variable
c	constructor
@	function in an application
$\#i$	i^{th} argument in an application
λx	abstraction over variable x
case	case selector
p	case branch pattern
let	an abstraction
τ_f	unfolding of the function f
τ_c	elimination of the constructor c
τ_β	β-substitution

LTS Representation of Example Program



Hierarchy of Program Transformers

- At each level, the transformer takes a program as input and produces a LTS as output.
- LTSs corresponding to previously encountered terms are compared to the LTS for the current term.
- If a renaming of a previously encountered LTS is detected, then folding is performed.
- If an embedding of a previously encountered LTS is detected, then generalization is performed.

Hierarchy of Program Transformers

- An LTS is a renaming of another if the same transitions are possible from each corresponding state (modulo variable renaming)[a].
- An LTS is an embedding of another if the same transitions are eventually possible from each corresponding state (modulo variable renaming).
- When generalizing, corresponding states with different transitions are extracted using **let**s (extractions which are weakly bisimilar are identified).

---

[a] Skipping over silent transitions.

Level 0 Transformer

- Corresponds closely to the positive supercompilation algorithm.
- Performs a normal-order reduction of the input program.
- The (LTS representation of) previously encountered terms with a function redex are memoized.
- If the (LTS representation of the) current term is a renaming of a memoized one, then folding is performed.
- If the (LTS representation of the) current term is an embedding of a memoized one, then generalization is performed.

Level 0 Transformer: Example

Result of level 0 transformation of the *fib* function (after residualization):

$$\begin{array}{lll}
f\ n & & \\
\textbf{where} & & \\
f & = & \lambda n.\textbf{case}\ n\ \textbf{of} \\
 & & \qquad\qquad Z \quad \Rightarrow 1 \\
 & & \qquad\quad\ \mid S\ n' \Rightarrow \textbf{case}\ n'\ \textbf{of} \\
 & & \qquad\qquad\qquad\qquad\quad Z \quad \Rightarrow 1 \\
 & & \qquad\qquad\qquad\qquad\ \mid S\ n'' \Rightarrow +\ (f\ n'')\ (f\ (S\ n''))
\end{array} \tag{1}$$

Level $n+1$ Transformer

- Rules are very similar to those for the level 0 transformer.
- Also performs a normal-order reduction of the input program.
- Differs from the level 0 transformer in that the LTSs which are memoized for the purposes of comparison when determining whether to fold or generalize are those resulting from the level n transformation of previously encountered terms.

Level 1 Transformer: Example

If the result of the level 0 transformation of the *fib* program is unfolded and further transformed within a level 1 transformer, then the following level 0 result is subsequently encountered:

$$\begin{array}{l} f\ n \\ \textbf{where} \\ f \;=\; \lambda n.\textbf{case}\ n\ \textbf{of} \\ \qquad\qquad\quad Z \;\;\Rightarrow + \ 1\ 1 \\ \qquad\qquad\; \mid S\ n' \Rightarrow \textbf{case}\ n'\ \textbf{of} \\ \qquad\qquad\qquad\qquad\quad Z \;\;\;\Rightarrow + \ 1\ (+\ 1\ 1) \\ \qquad\qquad\qquad\qquad\; \mid S\ n'' \Rightarrow + \ (f\ n'')\ (f\ (S\ n'')) \end{array} \tag{2}$$

Level 1 Transformer: Example

(2) is an embedding of (1):

$$
\begin{array}{l}
f\ n\\
\textbf{where}\\
f \;=\; \lambda n.\textbf{case}\ n\ \textbf{of}\\
\qquad\qquad Z \;\;\Rightarrow +\ 1\ 1\\
\qquad\qquad |\ S\ n' \Rightarrow \textbf{case}\ n'\ \textbf{of}\\
\qquad\qquad\qquad\qquad Z \;\;\Rightarrow +\ 1\ (+\ 1\ 1)\\
\qquad\qquad\qquad\qquad |\ S\ n'' \Rightarrow +\ (f\ n'')\ (f\ (S\ n''))
\end{array}
$$

Level 1 Transformer: Example

Generalization is therefore performed with respect to the previous level 0 result to obtain the following level 1 result:

$$
\begin{array}{l}
\mathbf{let}\ x = +\ 1\ 1 \\
\mathbf{in}\ \ \mathbf{let}\ x' = +\ 1\ (+\ 1\ 1) \\
\qquad \mathbf{in}\ \ f\ n \\
\qquad\qquad \mathbf{where} \\
\qquad\qquad f \ = \ \lambda n.\mathbf{case}\ n\ \mathbf{of} \\
\qquad\qquad\qquad\qquad\quad Z \quad \Rightarrow x \\
\qquad\qquad\qquad\qquad\ \mid S\ n' \Rightarrow \mathbf{case}\ n'\ \mathbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad Z \quad \Rightarrow x' \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \mid S\ n'' \Rightarrow +\ (f\ n'')\ (f\ (S\ n''))
\end{array}
\tag{3}
$$

Level 2 Transformer: Example

If this level 1 program is unfolded and further transformed within a level 2 transformer, then the following level 1 result is subsequently encountered.

$$
\begin{array}{l}
\textbf{let } x'' = +\ (+\ 1\ 1)\ (+\ 1\ (+\ 1\ 1)) \\
\textbf{in}\ \ \textbf{let } x''' = +\ (+\ 1\ (+\ 1\ 1))\ (+\ (+\ 1\ 1)\ (+\ 1\ (+\ 1\ 1))) \\
\qquad \textbf{in}\ \ f\ n \\
\qquad\qquad \textbf{where} \\
\qquad\qquad f = \lambda n.\textbf{case}\ n\ \textbf{of} \\
\qquad\qquad\qquad\qquad\quad \begin{array}{ll}
Z & \Rightarrow x'' \\
\mid S\ n' & \Rightarrow \textbf{case}\ n'\ \textbf{of} \\
 & \qquad \begin{array}{ll}
Z & \Rightarrow x''' \\
\mid S\ n'' & \Rightarrow +\ (f\ n'')\ (f\ (S\ n''))
\end{array}
\end{array}
\end{array} \tag{4}
$$

Level 2 Transformer: Example

(4) is an embedding of (3):

$$
\begin{array}{l}
\mathbf{let}\ x'' = +\ (+\ 1\ 1)\ \ (+\ 1\ (+\ 1\ 1))\\
\mathbf{in}\ \ \mathbf{let}\ x''' = +\ (+\ 1\ (+\ 1\ 1))\ \ (+\ (+\ 1\ 1)\ \ (+\ 1\ (+\ 1\ 1))\ )\\
\qquad \mathbf{in}\ \ f\ n\\
\qquad\qquad \mathbf{where}\\
\qquad\qquad f\ =\ \lambda n.\mathbf{case}\ n\ \mathbf{of}\\
\qquad\qquad\qquad\qquad\quad Z \ \ \ \Rightarrow x''\\
\qquad\qquad\qquad\qquad \mid S\ n' \Rightarrow \mathbf{case}\ n'\ \mathbf{of}\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad Z \ \ \ \ \Rightarrow x'''\\
\qquad\qquad\qquad\qquad\qquad\qquad \mid S\ n'' \Rightarrow +\ (f\ n'')\ (f\ (S\ n''))
\end{array}
$$

Level 1 Transformer: Example

Generalization is therefore performed with respect to the previous level 1 result to obtain the following level 2 result (identifying equivalent extractions):

$$
\begin{array}{l}
\textbf{let } x'' = +\ x\ x' \\
\textbf{in } \ \textbf{let } x''' = +\ x'\ (+\ x\ x') \\
\qquad \textbf{in } \ f\ n \\
\qquad\qquad \textbf{where} \\
\qquad\qquad f \ = \ \lambda n.\textbf{case } n \textbf{ of} \\
\qquad\qquad\qquad\qquad Z \quad \Rightarrow x'' \\
\qquad\qquad\qquad\qquad \mid S\ n' \Rightarrow \textbf{case } n' \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad Z \quad \Rightarrow x''' \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid S\ n'' \Rightarrow +\ (f\ n'')\ (f\ (S\ n''))
\end{array}
$$

Result of Transformation

$$
\begin{array}{l}
\textbf{case } n \textbf{ of} \\
\quad \begin{array}{ll}
Z & \Rightarrow 1 \\
\mid S\ n' & \Rightarrow \textbf{case } n' \textbf{ of} \\
& \quad \begin{array}{ll}
Z & \Rightarrow 1 \\
\mid S\ n'' & \Rightarrow f\ n''\ (+\ 1\ 1)\ (+\ 1\ (+\ 1\ 1))
\end{array}
\end{array} \\
\textbf{where} \\
f \;=\; \lambda n.\lambda x.\lambda x'.\textbf{case } n \textbf{ of} \\
\qquad\qquad\qquad \begin{array}{ll}
Z & \Rightarrow x \\
\mid S\ n' & \Rightarrow \textbf{case } n' \textbf{ of} \\
& \quad \begin{array}{ll}
Z & \Rightarrow x' \\
\mid S\ n'' & \Rightarrow f\ n''\ (+\ x\ x')\ (+\ x'\ (+\ x\ x'))
\end{array}
\end{array}
\end{array}
$$

This program has a run-time which is linear with respect to the size of the input number.

Program Transformation Hierarchy

- Each transformer in the hierarchy performs more specific generalization.
- Over-generalization is therefore less likely to occur when moving up through the levels.
- The problem is knowing which level is sufficient for the given program.
  - This should be the level beyond which no further improvements are obtained.
  - If an arbitrary level is chosen, this may be overkill in many cases.

Distillation

- Distillation in its most recent formulation (PEPM 2012) uses the results obtained from evaluating a program to determine when to perform generalization and folding.
- This formulation of distillation can also be described within our program transformation hierarchy by starting at level 0 and moving up to the next level when necessary,
- The need for generalization at one level in the hierarchy indicates the need to move up to the next level.
- The transformation of the *fib* function described earlier using a level 2 transformer can also be obtained using distillation.
    - Transformation starts at level 0.
    - The first generalization causes a move up to level 1.
    - The second generalization causes a move up to level 2.
    - The desired result is obtained at level 2.

Conclusions

- We have defined a hierarchy of transformers in which the transformer at each level of the hierarchy makes use of the transformers at lower levels.
- At the bottom of the hierarchy is the level 0 transformer, which corresponds to positive supercompilation, and is capable of achieving only linear improvements in efficiency.
- Higher levels in the hierarchy are capable of achieving superlinear improvements in efficiency; the first published definition of distillation (PEPM 2007) is at level 1 in this hierarchy.
- We have shown how the more recently published definition of distillation (PEPM 2012) can be simulated by moving up through the levels of the transformation hierarchy until no further improvements can be made.

Related Work

- Some semi-automatic techniques which also work on a number of levels and are capable of obtaining superlinear speedups 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.