A Graph-Based Definition of Distillation

Geoff Hamilton Gavin Mendel-Gleason
{hamilton,ggleason}@computing.dcu.ie

Lero@DCU
School of Computing
Dublin City University
Dublin, Ireland

META 2010

Background

- This work is inspired by the supercompilation transformation algorithm developed by Turchin.
- Although originally developed in the 1960s/70s, supercompilation did not become more widely known outside Russia until much later:
  - Published only in less accessible journals.
  - Defined on the unconventional language Refal.
- Supercompilation became more widely known through the positive supercompilation algorithm (Sørensen, Glück and Jones):
  - Simplified algorithm.
  - Defined on a more familiar functional language.

Motivation

Only linear improvements in performance are possible using the positive supercompilation algorithm. Transformations such as the following are not therefore possible:

$$
\begin{array}{l}
nrev\ xs \\
\textbf{where} \\
\begin{array}{lcl}
nrev & = & \lambda xs.\textbf{case}\ xs\ \textbf{of} \\
 & & \qquad\qquad \begin{array}{rcl} [] & \Rightarrow & [] \\ \mid x' : xs' & \Rightarrow & app\ (nrev\ xs')\ (x' : []) \end{array} \\
app & = & \lambda xs.\lambda ys.\textbf{case}\ xs\ \textbf{of} \\
 & & \qquad\qquad\qquad \begin{array}{rcl} [] & \Rightarrow & ys \\ \mid x' : xs' & \Rightarrow & x' : (app\ xs'\ ys) \end{array}
\end{array}
\end{array}
$$

$$\Downarrow$$

$$
\begin{array}{l}
arev\ xs \\
\textbf{where} \\
\begin{array}{lcl}
arev & = & \lambda xs.arev'\ xs\ [] \\
arev' & = & \lambda xs.\lambda ys.\textbf{case}\ xs\ \textbf{of} \\
 & & \qquad\qquad\qquad \begin{array}{rcl} [] & \Rightarrow & ys \\ \mid x' : xs' & \Rightarrow & arev'\ xs'\ (x' : ys) \end{array}
\end{array}
\end{array}
$$

Language Syntax

We use the following higher-order functional language:

$prog$	$::= e_0$ **where** $f_1 \ = \ e_1 \ldots \ f_k \ = \ e_k$	Program
e	$::= v$	Variable
	$\mid c \ e_1 \ldots e_k$	Constructor
	$\mid f$	Function Call
	$\mid \lambda v.e$	λ-Abstraction
	$\mid e_0 \ e_1$	Application
	$\mid$ **case** e_0 **of** $p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k$	Case Expression
p	$::= c \ v_1 \ldots v_k$	Pattern

The operational semantics of this language is normal-order reduction.

Positive Supercompilation

- Central concept is that of driving, which constructs a potentially infinite process tree, representing all possible computations of the program by normal order reduction.
- Generalization is required to ensure the termination of driving.
- Folding can be performed on encountering an instance of a previously encountered term, thus producing a finite partial process tree.
- A (hopefully) more efficient recursive program can be extracted from the resulting partial process tree.

Process Trees

- $e \rightarrow t_1, \ldots, t_n$ is the process tree with root labelled e and n children which are the subtrees $t_1, \ldots, t_n$ respectively.
- $root(t)$ denotes the root node of process tree t.
- $t(\alpha)$ denotes the label of node α within process tree t.
- $anc(t, \alpha)$ denotes the set of ancestors of node α within t.
- $t\{\alpha := t'\}$ denotes the tree obtained by replacing the subtree with root α in t by the tree t'.

Expression Instance

- An expression e' is an instance of expression e, denoted by $e \lessdot_e e'$, if there is a substitution θ such that $e\,\theta \equiv e'$.
- When an instance is encountered, a repeat node is constructed.
- $e \dashrightarrow \alpha$ denotes a repeat node where the expression e is an instance of the label of node α.

Expression Embedding

The homeomorphic embedding relation $\lesssim_e$ is a well-quasi order which provides a so-called whistle to stop driving due to potential divergence, and to indicate that generalization should be performed.

$$\frac{e_1 \; \lhd_e \; e_2}{e_1 \; \unlhd_e \; e_2} \qquad \frac{e_1 \; \bowtie_e \; e_2}{e_1 \; \unlhd_e \; e_2} \qquad \frac{e \unlhd_e (e'[v/v'])}{\lambda v.e \bowtie_e \lambda v'.e'}$$

$$\frac{\exists i \in \{1 \ldots n\}.e \unlhd_e e_i}{e \lhd_e \phi(e_1, \ldots, e_n)} \qquad \frac{\forall i \in \{1 \ldots n\}.e_i \unlhd_e e_i'}{\phi(e_1, \ldots, e_n) \bowtie_e \phi(e_1', \ldots, e_n')}$$

$$\frac{e_0 \unlhd_e e_0' \quad \forall i \in \{1 \ldots n\}.\exists \theta_i.p_i \equiv (p_i' \; \theta_i) \wedge e_i \unlhd_e (e_i' \; \theta_i)}{(\textbf{case } e_0 \textbf{ of } p_1 : e_1 | \ldots | p_n : e_n) \bowtie_e (\textbf{case } e_0' \textbf{ of } p_1' : e_1' | \ldots | p_n' : e_n')}$$

$$e_1 \lesssim_e e_2 \text{ iff } \exists \theta.e_1 \; \theta \bowtie_e e_2$$

A generalization of expressions e and e' is a triple (e_g, θ, θ') where $e_g\ \theta \equiv e$ and $e_g\ \theta' \equiv e'$.

$$e \sqcap_e e' = \begin{cases} (\phi(e_1^g, \ldots, e_n^g), \bigcup_{i=1}^n \theta_i, \bigcup_{i=1}^n \theta_i'), & \text{if } e \lesssim_e e' \\ \quad \text{where} \quad e = \phi(e_1, \ldots, e_n) & \\ \qquad\qquad\quad e' = \phi(e_1', \ldots, e_n') & \\ \qquad\qquad\quad (e_i^g, \theta_i, \theta_i') = e_i \sqcap_e e_i' & \\ (v, [e/v], [e'/v]), & \text{otherwise} \end{cases}$$

Expression Generalization

- When we encounter an expression e' which is a coupling of a previously encountered expression e, we perform generalization.
- To represent the result of generalization, we introduce a **let** construct of the form **let** $v_1 = e_1, \ldots, v_n = e_n$ **in** e_0 into our language.
- The expression e is replaced by its generalized form, which is constructed using the *abstract*$_e$ operation:
  $abstract_e(e, e')$ = **let** $v_1 = e_1, \ldots, v_n = e_n$ **in** e_g
  where $e \sqcap_e e' = (e_g, \{v_1 := e_1, \ldots, v_n := e_n\}, \theta)$

Positive Supercompilation: Algorithm

If the partially constructed process tree t contains a node β in which the redex is a function, the node is processed as follows:

if $\exists \alpha \in \mathit{anc}(t,\beta).t(\alpha) \lessdot_e t(\beta)$
then $t\{\beta := t(\beta) \dashrightarrow \alpha\}$
else if $\exists \alpha \in \mathit{anc}(t,\beta).t(\alpha) \lesssim_e t(\beta)$
then $t\{\alpha := \mathcal{S}[\![\mathit{abstract}_e(t(\alpha), t(\beta))]\!]\}$
else $t\{\beta := t(\beta) \rightarrow \mathcal{S}[\![\mathit{unfold}(t(\beta))]\!]\}$

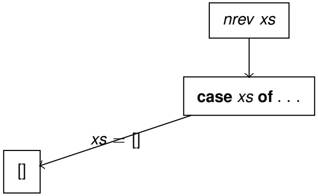
Positive Supercompilation: Example

$\boxed{nrev\ xs}$

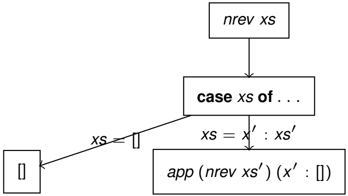
Positive Supercompilation: Example

$$\begin{tikzcd} \boxed{\mathit{nrev}\ \mathit{xs}} \arrow[d] \\ \boxed{\textbf{case}\ \mathit{xs}\ \textbf{of}\ \ldots} \end{tikzcd}$$

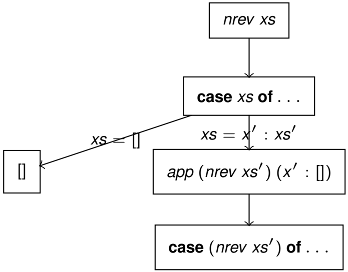
Positive Supercompilation: Example



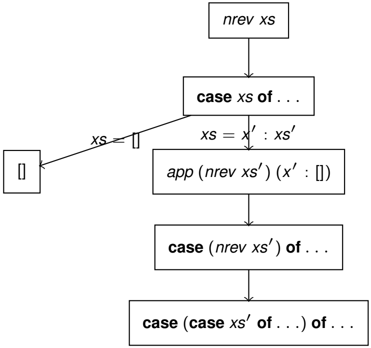
Positive Supercompilation: Example



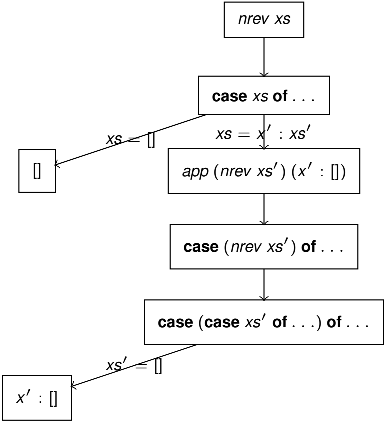
Positive Supercompilation: Example



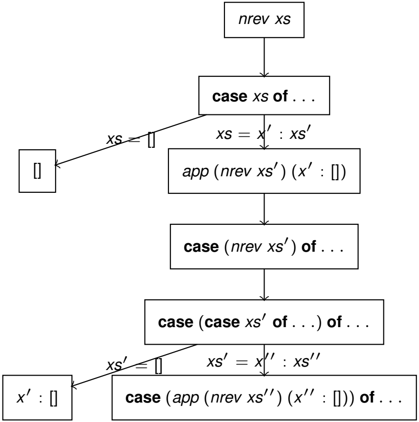
Positive Supercompilation: Example



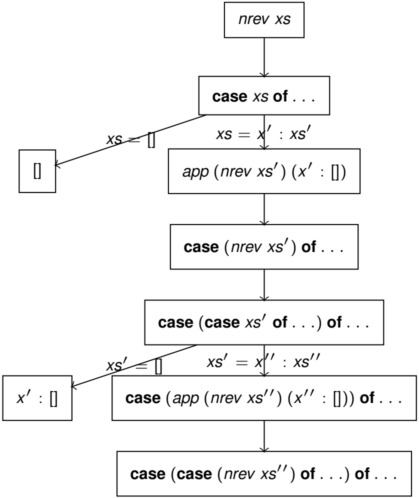
Positive Supercompilation: Example



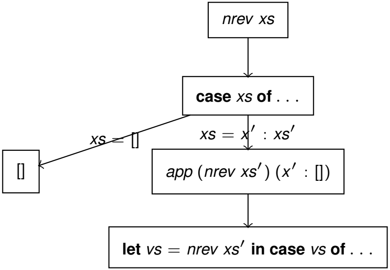
Positive Supercompilation: Example



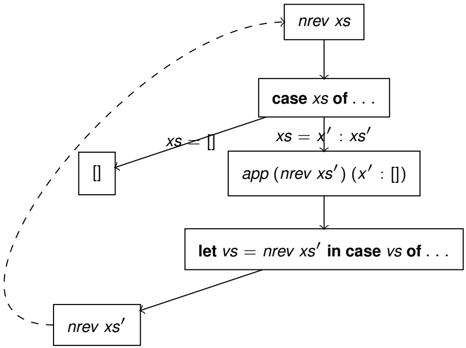
Positive Supercompilation: Example



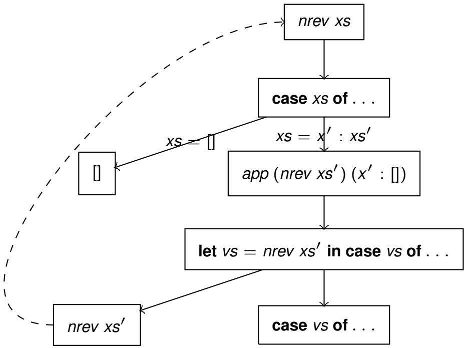
Positive Supercompilation: Example



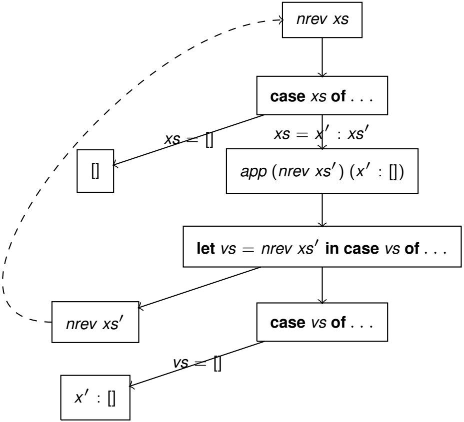
Positive Supercompilation: Example



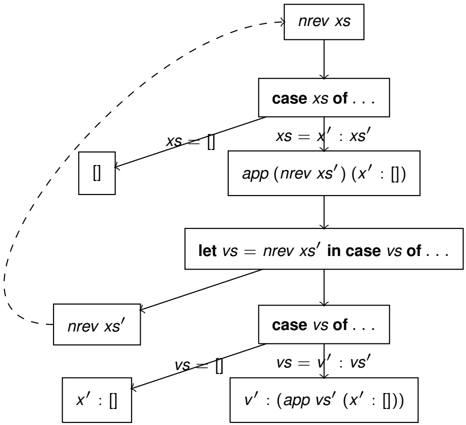
Positive Supercompilation: Example



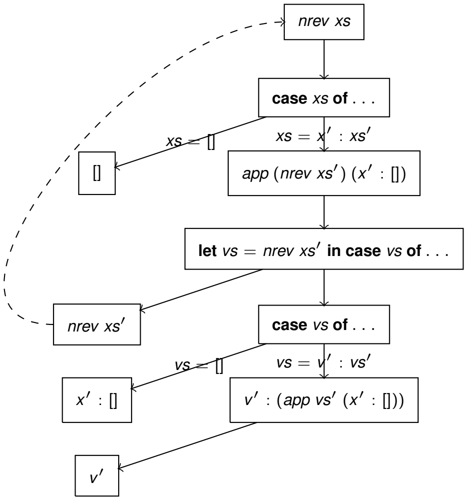
Positive Supercompilation: Example



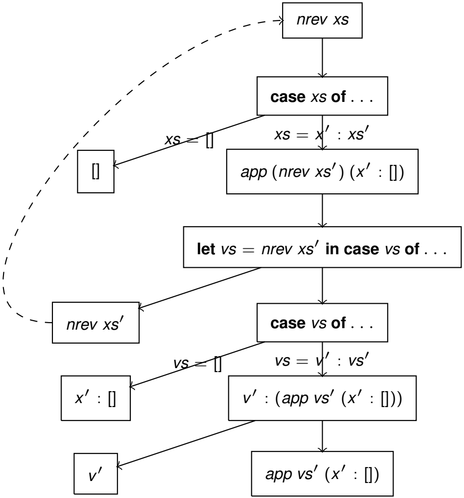
Positive Supercompilation: Example



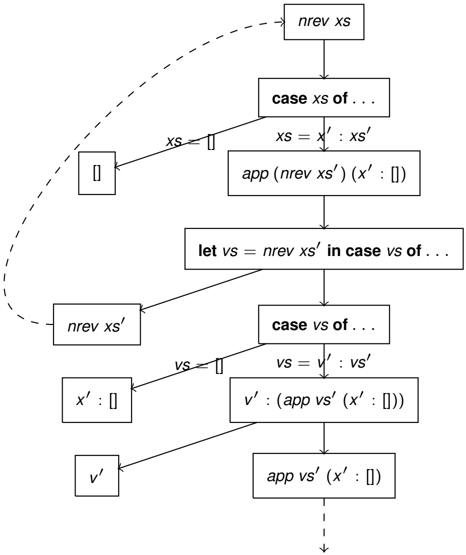
Positive Supercompilation: Example



Positive Supercompilation: Example



Positive Supercompilation: Example



The following program is constructed from this process graph:

$$
\begin{array}{l}
f\ xs \\
\textbf{where} \\
\begin{array}{lcl}
f & = & \lambda xs.\textbf{case}\ xs\ \textbf{of} \\
 & & \qquad\qquad \begin{array}{lll} & [] & \Rightarrow [] \\ \mid & x' : xs' & \Rightarrow \textbf{case}\ (f\ xs')\ \textbf{of} \\ & & \qquad \begin{array}{lll} & [] & \Rightarrow [x'] \\ \mid & x'' : xs'' & \Rightarrow x'' : (f'\ xs''\ x') \end{array} \end{array} \\
f' & = & \lambda xs.\lambda y.\textbf{case}\ xs\ \textbf{of} \\
 & & \qquad\qquad \begin{array}{lll} & [] & \Rightarrow [y] \\ \mid & x' : xs' & \Rightarrow x' : (f'\ xs'\ y) \end{array}
\end{array}
\end{array}
$$

Distillation

- Central concept is also driving to construct a potentially infinite process tree representing all possible computations of the program by normal order reduction.
- The terms in the nodes of this process tree are transformed by positive supercompilation to obtain their corresponding transition system representation.
- Generalization and folding are performed with respect to these transition systems.
- As a result, it is possible to obtain a super-linear improvement in performance as computationally expensive terms can be extracted from within loops in these transition systems.

Transition System Instance

Two transition systems are equivalent if the following relation is satisfied:

$$con\langle e \rangle \ \rightarrow \ t_1, \ldots, t_n \equiv con'\langle e' \rangle \ \rightarrow \ t'_1, \ldots, t'_n, \quad \text{iff } e \lesssim_e e' \wedge \forall i \in \{1 \ldots n\}.t_i \equiv t'_i$$

$$e \dashrightarrow t \equiv e' \dashrightarrow t', \quad \text{iff } t \equiv t'$$

A transition system t' is an instance of another transition system t (denoted by $t \lessdot_t t'$) iff there is a substitution θ s.t. $t \equiv t' \ \theta$.

Transition System Embedding

The embedding relation $\lesssim_t$ on transition systems is defined as follows:

$$\frac{t_1 \lhd_t t_2}{t_1 \unlhd_t t_2} \qquad \frac{t_1 \bowtie_t t_2}{t_1 \unlhd_t t_2} \qquad \frac{t \unlhd_t (t'[v/v'])}{\lambda v.e \rightarrow t \bowtie_t \lambda v'.e' \rightarrow t'}$$

$$\frac{e \bowtie_e e' \quad \forall i \in \{1 \ldots n\}.t_i \unlhd_t t'_i}{con\langle e\rangle \rightarrow t_1, \ldots, t_n \bowtie_t con'\langle e'\rangle \rightarrow t'_1, \ldots, t'_n} \qquad \frac{\exists i \in \{1 \ldots n\}.t \unlhd_t t_i}{t \lhd_t e \rightarrow t_1, \ldots, t_n}$$

$$\frac{t \bowtie_t t'}{e \dashrightarrow t \bowtie_t e' \dashrightarrow t'}$$

$$\frac{t_0 \unlhd_t t'_0 \quad \forall i \in \{1 \ldots n\}.\exists \theta_i.p_i \equiv (p'_i\ \theta_i) \wedge t_i \unlhd_t (t'_i\ \theta_i)}{(\mathbf{case}\ e_0\ \mathbf{of}\ p_1 : e_1 | \ldots | p_n : e_n) \rightarrow t_0, \ldots, t_n \bowtie_t (\mathbf{case}\ e'_0\ \mathbf{of}\ p'_1 : e'_1 | \ldots | p'_n : e'_n) \rightarrow t'_0, \ldots, t'_n}$$

$$t_1 \lesssim_t t_2 \text{ iff } \exists \theta.t_1\ \theta \bowtie_t t_2$$

Transition System Generalization

A generalization of transition systems t and t' is a triple (t_g, θ, θ') where $t_g\ \theta \equiv t$ and $t_g\ \theta' \equiv t'$.

$$t \sqcap_t t' = \begin{cases} (e \rightarrow t_1^g, \ldots, t_n^g, \bigcup_{i=1}^n \theta_i, \bigcup_{i=1}^n \theta_i'), & \text{if } t \lesssim_t t' \\ \quad \text{where} \quad t = e \rightarrow t_1, \ldots, t_n & \\ \quad\quad\quad\quad\quad t' = e' \rightarrow t_1', \ldots, t_n' & \\ \quad\quad\quad\quad\quad (t_i^g, \theta_i, \theta_i') = t_i \sqcap_t t_i' & \\ (v\ v_1 \ldots v_k, \{v := \lambda v_1 \ldots v_k.t\}, \{v := \lambda v_1 \ldots v_k.t'\}), & \\ & \text{otherwise} \\ \text{where} \quad \{v_1 \ldots v_k\} = bv(t) \cup bv(t') & \end{cases}$$

Transition System Generalization

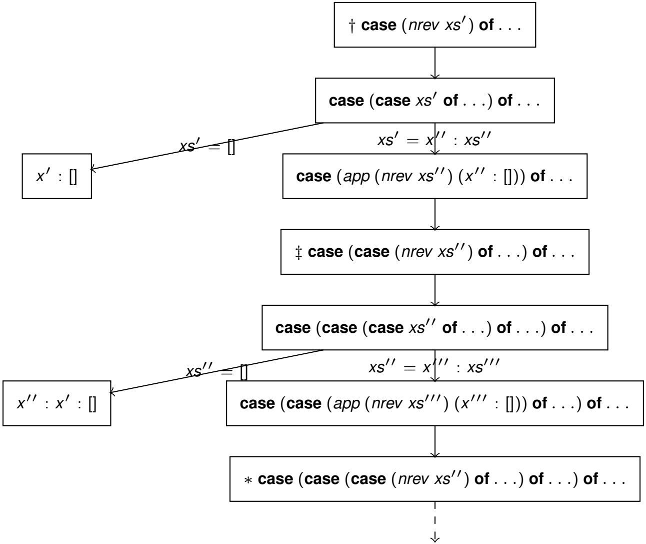
- When we encounter a transition system t' which is a coupling of a previously encountered transition system t, we perform generalization.
- To represent the result of generalization, we introduce a generalized node of the form **let** $v_1 = t_1, \ldots, v_n = t_n$ **in** t_0 into our transition systems.
- The transition system t is replaced by its generalized form, which is constructed using the *abstract$_t$* operation: *abstract$_t$*(t, t') = **let** $v_1 = t_1, \ldots, v_n = t_n$ **in** t_g where $t \sqcap_t t' = (t_g, \{v_1 := t_1, \ldots, v_n := t_n\}, \theta)$

Distillation: Algorithm

If the partially constructed process tree t contains a node β in which the redex is a function, the node is processed as follows:

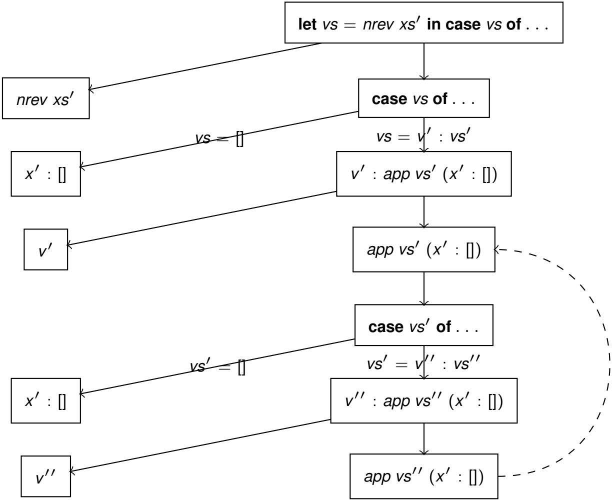
if $\exists \alpha \in \mathit{anc}(t, \beta).\mathcal{S}[\![t(\alpha)]\!] \lessdot_t \mathcal{S}[\![t(\beta)]\!]$
then $t\{\beta := t(\beta) \dashrightarrow \alpha\}$
else if $\exists \alpha \in \mathit{anc}(t, \beta).\mathcal{S}[\![t(\alpha)]\!] \lesssim_t \mathcal{S}[\![t(\beta)]\!]$
  then $t\{\alpha := \mathcal{D}[\![\mathit{abstract}_t(\mathcal{S}[\![t(\alpha)]\!], \mathcal{S}[\![t(\beta)]\!])]\!]\}$
  else $t\{\beta := t(\beta) \rightarrow \mathcal{D}[\![\mathit{unfold}(t(\beta))]\!]\}$

Distillation Example: *nrev xs*



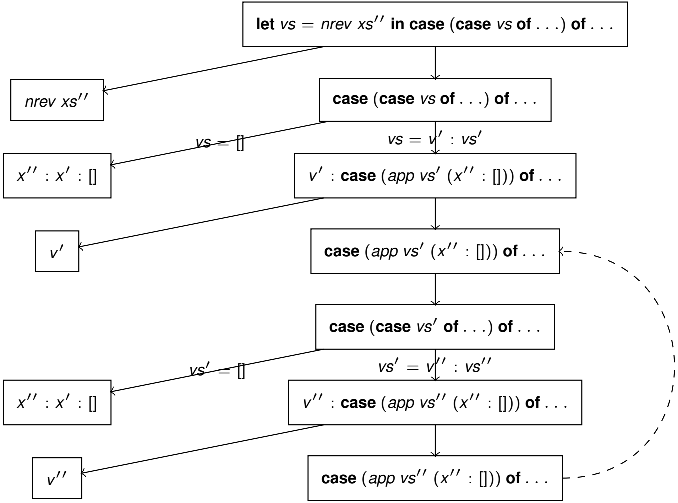
Distillation Example: *nrev xs*

The following transition system is constructed from the term †:



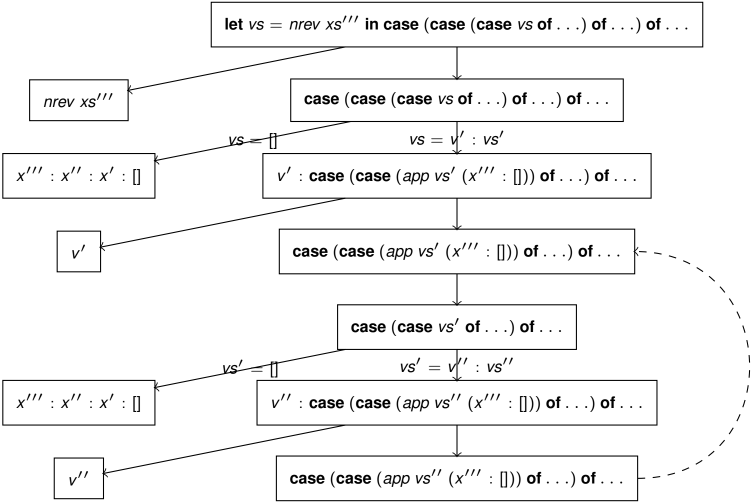
Distillation Example: *nrev xs*

The following transition system is constructed from the term ‡:

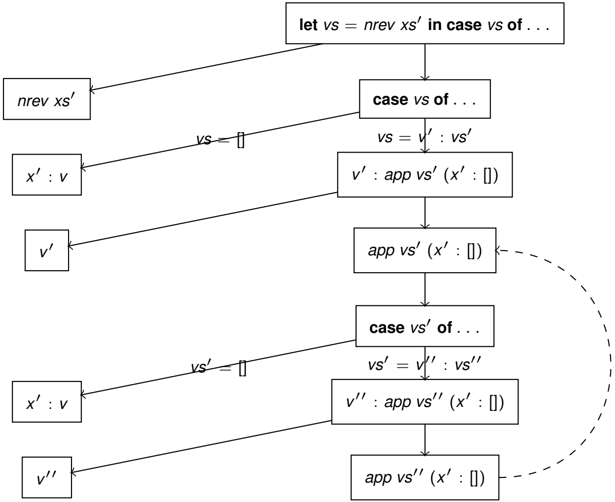


Distillation Example: *nrev xs*

The following transition system is constructed from the term $*$:

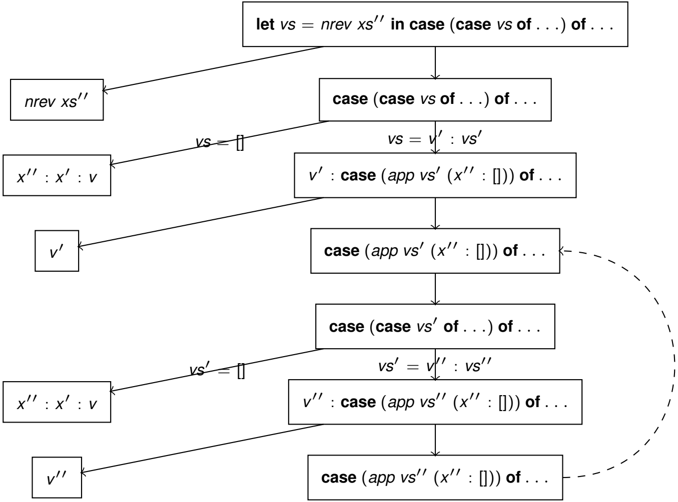


Transition system † is generalized with respect to transition system ‡ to give:



Distillation Example: *nrev xs*

Transition system $\ddagger$ is generalized with respect to transition system $*$ to give:



Distillation Example: *nrev xs*

The second of these transition systems is an instance of the first, so folding is performed to obtain the following program:

$$
\begin{array}{l}
\textbf{case } xs \textbf{ of} \\
\quad \begin{array}{ll} [\,] & \Rightarrow [\,] \\ \mid x' : xs' & \Rightarrow f\ x'\ xs'\ [\,] \end{array} \\
\textbf{where} \\
f \;=\; \lambda x'.\lambda xs'.\lambda v.\textbf{case } xs' \textbf{ of} \\
\qquad\qquad\qquad\qquad \begin{array}{ll} [\,] & \Rightarrow x' : v \\ \mid x'' : xs'' & \Rightarrow f\ x''\ xs''\ (x' : v) \end{array}
\end{array}
$$

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

Distillation Example: $app\ (arev'\ xs\ ys)\ zs$

The following program is obtained after applying distillation:

$$
\begin{array}{l}
f\ xs\ (g\ ys\ zs) \\
\textbf{where} \\
\begin{array}{lcl}
f & = & \lambda xs.\lambda v.\textbf{case}\ xs\ \textbf{of} \\
 & & \qquad\qquad\quad \begin{array}{ll} [] & \Rightarrow v \\ \mid x' : xs' & \Rightarrow f\ xs'\ (x' : v) \end{array} \\
g & = & \lambda ys.\lambda zs.\textbf{case}\ ys\ \textbf{of} \\
 & & \qquad\qquad\quad \begin{array}{ll} [] & \Rightarrow zs \\ \mid y' : ys & \Rightarrow y' : (g\ ys'\ zs) \end{array}
\end{array}
\end{array}
$$

The intermediate list $(arev'\ xs\ ys)$ within the initial program has therefore been eliminated. This intermediate list is not removed using positive supercompilation.

Distillation Example: *fib n*

The *fib* function:

$$
\begin{array}{l}
fib\ n \\
\textbf{where} \\
\begin{array}{lcl}
fib & = & \lambda n.\textbf{case}\ n\ \textbf{of} \\
& & \quad\quad \begin{array}{ll} Zero & \Rightarrow Succ\ Zero \\ \mid Succ\ n' & \Rightarrow \textbf{case}\ n'\ \textbf{of} \\ & \quad\quad \begin{array}{ll} Zero & \Rightarrow Succ\ Zero \\ \mid Succ\ n'' & \Rightarrow add\ (fib\ n')\ (fib\ n'') \end{array} \end{array} \\
add & = & \lambda x.\lambda y.\textbf{case}\ x\ \textbf{of} \\
& & \quad\quad \begin{array}{ll} Zero & \Rightarrow y \\ \mid Succ\ x' & \Rightarrow Succ\ (add\ x'\ y) \end{array}
\end{array}
\end{array}
$$

is transformed to something similar to the following by distillation:

$$
\begin{array}{l}
f\ n\ (\lambda n.Succ\ n)\ (\lambda n.Succ\ n) \\
\textbf{where} \\
\begin{array}{lcl}
f & = & \lambda n.\lambda g.\lambda h.\textbf{case}\ n\ \textbf{of} \\
& & \quad\quad \begin{array}{ll} Zero & \Rightarrow g\ Zero \\ \mid Succ\ n' & \Rightarrow f\ n'\ h\ (\lambda n.h\ (g\ n)) \end{array}
\end{array}
\end{array}
$$

Conclusions

- Distillation and positive supercompilation are very similar algorithms.
- Distillation extends the range of transformations which can be performed beyond those which can be performed by positive supercompilation.
- Distillation can produce a superlinear speedup in programs, which is not possible using positive supercompilation.
- The extra power of distillation is obtained by extracting computationally expensive terms from within loops in transition systems.
- The extra power of distillation comes at a price; there is an exponential increase in the number of steps required in the worst case.

Further Work

- Use of the distillation algorithm to facilitate the proof of temporal formulae in the μ-calculus.
- Proofs of termination and correctness of the distillation algorithm.
- More detailed comparison between distillation and higher-level supercompilation.
- Incorporation of the distillation algorithm into the Haskell programming language.
- Use of the distillation algorithm to facilitate the parallelization of code.