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
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 therefore not possible:

\[
\begin{align*}
nrev \; xs \\
\text{where} \\
nrev & = \lambda xs. \text{case} \; xs \; \text{of} \\
\emptyset & \Rightarrow \emptyset \\
| \; x' : xs' & \Rightarrow \text{app} \; (nrev \; xs') \; (x' : \emptyset) \\
\text{app} & = \lambda xs. \lambda ys. \text{case} \; xs \; \text{of} \\
\emptyset & \Rightarrow ys \\
| \; x' : xs' & \Rightarrow x' : (\text{app} \; xs' \; ys) \\
\downarrow \\
\text{arev} \; xs \\
\text{where} \\
\text{arev} & = \lambda xs. \text{arev'} \; xs \; \emptyset \\
\text{arev'} & = \lambda xs. \lambda ys. \text{case} \; xs \; \text{of} \\
\emptyset & \Rightarrow ys \\
| \; x' : xs' & \Rightarrow \text{arev'} \; xs' \; (x' : ys)
\end{align*}
\]
We use the following higher-order functional language:

\[
prog ::= e_0 \textbf{where } f_1 = e_1 \ldots f_k = e_k \\

\]

\[
e ::= v \\
| c \ e_1 \ldots e_k \\
| f \\
| \lambda v. e \\
| e_0 \ e_1 \\
| \textbf{case } e_0 \textbf{ of } p_1 \Rightarrow e_1 | \cdots | p_k \Rightarrow e_k
\]

Program

Variable

Constructor

Function Call

\(\lambda\)-Abstraction

Application

Case Expression

\[
p ::= c \ v_1 \ldots v_k
\]

Pattern

The operational semantics of this language is normal-order reduction.
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.
- \( \text{root}(t) \) denotes the root node of process tree \( t \).
- \( t(\alpha) \) denotes the label of node \( \alpha \) within process tree \( t \).
- \( \text{anc}(t, \alpha) \) denotes the set of ancestors of node \( \alpha \) within \( t \).
- \( t\{\alpha := t'\} \) denotes the tree obtained by replacing the subtree with root \( \alpha \) in \( t \) by the tree \( t' \).
An expression $e'$ is an instance of expression $e$, denoted by $e \prec_e e'$, if there is a substitution $\theta$ such that $e \theta \equiv e'$.

When an instance is encountered, a repeat node is constructed.

e $\rightarrow \alpha$ denotes a repeat node where the expression $e$ is an instance of the label of node $\alpha$. 
The homeomorphic embedding relation $\preceq_e$ 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.

\[
\begin{align*}
\frac{e_1 \triangleleft_e e_2}{e_1 \preceq_e e_2} & \quad \frac{e_1 \triangleright_e e_2}{e_1 \preceq_e e_2} \quad \frac{e \preceq_e (e'[v/v'])}{\lambda v.e \triangleleft_e \lambda v'.e'}
\end{align*}
\]

\[
\begin{align*}
\exists i \in \{1 \ldots n\}. e \preceq_e e_i & \quad \forall i \in \{1 \ldots n\}. e_i \preceq_e e_i' \\
 e \preceq_e \phi(e_1, \ldots, e_n) & \quad \phi(e_1, \ldots, e_n) \triangleright_e \phi(e'_1, \ldots, e'_n)
\end{align*}
\]

\[
\begin{align*}
\exists i \in \{1 \ldots n\}. e_0 \preceq_e e_i' & \quad \forall i \in \{1 \ldots n\}. \exists \theta_i. p_i \equiv (p_i' \theta_i) \land e_i \preceq_e (e_i' \theta_i) \\
(case ~ e_0 ~ of ~ p_1 : e_1 | \ldots | p_n : e_n) \triangleright_e (case ~ e_0' ~ of ~ p_1' : e_1' | \ldots | p_n' : e_n') & \quad e_1 \preceq_e e_2 \text{ iff } \exists \theta. e_1 \theta \triangleright_e e_2
\end{align*}
\]
A generalization of expressions $e$ and $e'$ is a triple $(e_g, \theta, \theta')$ where $e_g \theta \equiv e$ and $e_g \theta' \equiv e'$.

$$e \sqcap_e e' = \begin{cases} 
(\phi(e^g_1, \ldots, e^g_n), \bigcup_{i=1}^n \theta_i, \bigcup_{i=1}^n \theta'_i), & \text{if } e \preceq_e e' \\
\phi(e_1, \ldots, e_n) & \\
\phi(e'_1, \ldots, e'_n) & \\
(e^g_i, \theta_i, \theta'_i) = e_i \sqcap_e e'_i & \\
(v, [e/v], [e'/v]), & \text{otherwise}
\end{cases}$$
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') = \text{let } v_1 = e_1, \ldots, v_n = e_n \text{ in } e_g \\
\text{where } e \sqcap_\( e e' = (e_g, \{ v_1 := e_1, \ldots, v_n := e_n \}, \theta)\)
\]
If the partially constructed process tree $t$ contains a node $\beta$ in which the redex is a function, the node is processed as follows:

if $\exists \alpha \in \text{anc}(t, \beta). t(\alpha) \preceq_e t(\beta)$
then $t\{\beta := t(\beta) \rightarrow \alpha\}$
else if $\exists \alpha \in \text{anc}(t, \beta). t(\alpha) \preceq_e t(\beta)$
then $t\{\alpha := S[\text{abstract}_e(t(\alpha), t(\beta))]\}$
else $t\{\beta := t(\beta) \rightarrow S[\text{unfold}(t(\beta))]\}$
Positive Supercompilation: Example

```haskell
nrev xs
```

Geoff Hamilton, Gavin Mendel-Gleason
A Graph-Based Definition of Distillation
Positive Supercompilation: Example

\[ nrev\;xs \]

\[ \text{case}\;xs\;\text{of}\;\ldots \]
Positive Supercompilation: Example

```
let vs = nrev xs in case vs of . . .
```

```
nrev xs
```

```
xs -> case xs of . . .
```

```
[] -> nrev xs
```

```
app (nrev xs' (x':[]))
```

```
k :: xs' -> xs' = x':xs'
```

```
case (nrev xs' of . . .)
```

```
case (case xs' of . . .)
```

```
x':[] -> vs' = []
```

```
app vs' (x':[])
```

```
v' :: vs' = app vs (x':[])
```

Geoff Hamilton, Gavin Mendel-Gleason

A Graph-Based Definition of Distillation
Positive Supercompilation: Example

```
nrev xs

case xs of . . .

[]

xs = x': xs'

app (nrev xs') (x' : [])
```
Positive Supercompilation: Example

Let \( \text{vs} = \text{nrev} \, \text{xs} \) and consider the following case:

\[
\text{case} \, \text{xs} \, \text{of} \, \ldots
\]

The case expands as follow:

\[
\text{app} \, \left( \text{nrev} \, \text{xs}' \right) \left( x' : [] \right)
\]

Finally, the case simplifies to:

\[
\text{case} \, \left( \text{nrev} \, \text{xs}' \right) \, \text{of} \, \ldots
\]
Positive Supercompilation: Example

```
let vs = nrev xs' in case vs of . . .
```

```
nrev xs
```

```
xs = x' : xs'
```

```
app (nrev xs') (x' : [])
```

```
case (nrev xs') of . . .
```

```
case (case xs' of . . .) of . . .
```

Geoff Hamilton, Gavin Mendel-Gleason
A Graph-Based Definition of Distillation
Positive Supercompilation: Example

\[ nrev \, xs \]

\[ \text{case} \, xs \, \text{of} \, . . . \]

\[ \text{case} \, (nrev \, xs') \, \text{of} \, . . . \]

\[ \text{case} \, (\text{case} \, xs' \, \text{of} \, . . .) \, \text{of} \, . . . \]

Geoff Hamilton, Gavin Mendel-Gleason

A Graph-Based Definition of Distillation
Positive Supercompilation: Example

Geoff Hamilton, Gavin Mendel-Gleason  A Graph-Based Definition of Distillation
Positive Supercompilation: Example

\[ \text{let } v = \text{nrev } xs' \text{ in case } v \text{ of . . .} \]

\[ \text{case } v \text{ of . . .} \]

\[ \text{case } (\text{case } xs' \text{ of . . .}) \text{ of . . .} \]

\[ \text{case } (\text{case } xs' \text{ of . . .}) \text{ of . . .} \]

\[ \text{case } (\text{case } xs' \text{ of . . .}) \text{ of . . .} \]

\[ \text{case } (\text{case } xs' \text{ of . . .}) \text{ of . . .} \]

\[ \text{case } (\text{case } xs' \text{ of . . .}) \text{ of . . .} \]

\[ \text{case } (\text{case } xs' \text{ of . . .}) \text{ of . . .} \]

\[ \text{case } (\text{case } xs' \text{ of . . .}) \text{ of . . .} \]

\[ \text{case } (\text{case } xs' \text{ of . . .}) \text{ of . . .} \]

\[ \text{case } (\text{case } xs' \text{ of . . .}) \text{ of . . .} \]

\[ \text{case } (\text{case } xs' \text{ of . . .}) \text{ of . . .} \]
Positive Supercompilation: Example

```
let vs = nrev xs' in case vs of ... 
```

```
app (nrev xs') (x' : []) 
```

```
case xs of ... 
```

```
x = x' : xs' 
```

```
[] 
```

```
nrev xs 
```

```
``
Positive Supercompilation: Example

\[
\text{let } \texttt{vs} = \texttt{nrev xs}' \text{ in case } \texttt{vs} \text{ of } \ldots
\]

\[
\text{app } (\texttt{nrev xs}') (x': [])
\]

\[
\text{case } \texttt{xs} \text{ of } \ldots
\]

\[
\text{xs} = x': xs'
\]

\[
\text{let } \texttt{vs} = \texttt{nrev xs}' \text{ in case } \texttt{vs} \text{ of } \ldots
\]

\[
\text{app } (\texttt{nrev xs}') (x': [])
\]

\[
\text{case } \texttt{xs} \text{ of } \ldots
\]

\[
\text{xs} = x': xs'
\]
Positive Supercompilation: Example

\[
\text{let } vs = \text{nrev } xs' \text{ in case } vs \text{ of . . .}
\]

\[
\text{case } xs \text{ of . . .}
\]

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

\[
\text{case } xs' \text{ of . . .}
\]

\[
\text{nrev } xs'
\]

\[
\text{xs = x' : xs'}
\]
Positive Supercompilation: Example

\[\text{let } vs = \text{nrev } xs' \text{ in case } vs \text{ of . . .}\]

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

\[\text{case x of . . .}\]

\[\text{xs = x' : xs'}\]

\[\text{nrev } xs\]

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

\[\text{xs} = []\]

\[\text{x' : []}\]
Positive Supercompilation: Example

Geoff Hamilton, Gavin Mendel-Gleason
A Graph-Based Definition of Distillation
Positive Supercompilation: Example

```
nrev xs

\[
\text{case } xs \text{ of } \ldots
\]

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

\[
\text{let } vs = nrev xs' \text{ in case } vs \text{ of } \ldots
\]

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

\[
x' : []
\]

\[
v' : (\text{app } vs' (x' : []))
\]
Positive Supercompilation: Example

```
nrev xs

case xs of . . .

xs = x' : xs'

app (nrev xs') (x' : [])

let vs = nrev xs' in case vs of . . .

nrev xs'

case vs of . . .

vs = v' : vs'

x' : []

v' : (app vs' (x' : []))

v'

app vs' (x' : [])
```
Positive Supercompilation: Example

Let $vs = \text{nrev } xs'$ in \text{case } vs of . . .

\begin{align*}
\text{case } vs' \text{ of . . .} \\
v' : (\text{app } vs' (x' : []))
\end{align*}
The following program is constructed from this process graph:

\[
\begin{align*}
f \; xs \\
\text{where} \\
f & = \lambda xs. \text{case } xs \text{ of} \\
& \quad \emptyset \Rightarrow \emptyset \\
& \quad | x' : xs' \Rightarrow \text{case } (f \; xs') \text{ of} \\
& \quad \quad \emptyset \Rightarrow [x'] \\
& \quad \quad | x'' : xs'' \Rightarrow x'' : (f' \; xs'' \; x') \\
\end{align*}
\]

\[
\begin{align*}
f' & = \lambda xs. \lambda y. \text{case } xs \text{ of} \\
& \quad \emptyset \Rightarrow [y] \\
& \quad | x' : xs' \Rightarrow x' : (f' \; xs' \; y)
\end{align*}
\]
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.
Two transition systems are equivalent if the following relation is satisfied:

\[\text{con}\langle e\rangle \to t_1, \ldots, t_n \equiv \text{con}'\langle e'\rangle \to t'_1, \ldots, t'_n,\]

iff \(e \lesssim e' \land \forall i \in \{1 \ldots n\}. t_i \equiv t'_i\)

\(e \longrightarrow t \equiv e' \longrightarrow t',\) iff \(t \equiv t'\)

A transition system \(t'\) is an instance of another transition system \(t\) (denoted by \(t \lhd_t t'\)) iff there is a substitution \(\theta\) s.t. \(t \equiv t' \theta\).
The embedding relation \( \triangleleft_t \) on transition systems is defined as follows:

\[
\begin{align*}
& t_1 \triangleleft_t t_2 \\
\implies & t_1 \sqsubseteq_t t_2 \\
& t_1 \triangleright_t t_2 \\
\implies & t_1 \sqsubseteq_t t_2 \\
& t \sqsubseteq_t (t'[v/v']) \\
\implies & \lambda v.e \rightarrow t \triangleright_t \lambda v'.e' \rightarrow t' \\
& e \bowtie_t e' \quad \forall i \in \{1 \ldots n\}. t_i \sqsubseteq_t t_i' \\
\implies & \text{con}\langle e \rangle \rightarrow t_1, \ldots, t_n \bowtie_t \text{con}\langle e' \rangle \rightarrow t_1', \ldots, t_n' \\
& \exists i \in \{1 \ldots n\}. t \sqsubseteq_t t_i \\
\implies & t \triangleright_t e \rightarrow t_1, \ldots, t_n \\
& t \triangleright_t t' \\
\implies & e \rightarrow t \triangleright_t e' \rightarrow t' \\
& t_0 \sqsubseteq_t t_0' \quad \forall i \in \{1 \ldots n\}. \exists \theta_i. p_i \equiv (p_i' \theta_i) \land t_i \sqsubseteq_t (t_i' \theta_i) \\
& (\text{case } e_0 \text{ of } p_1 : e_1 | \ldots | p_n : e_n) \rightarrow t_0, \ldots, t_n \triangleright_t (\text{case } e_0' \text{ of } p_1' : e_1' | \ldots | p_n' : e_n') \rightarrow t_0', \ldots, t_n' \\
& t_1 \triangleleft_t t_2 \text{ iff } \exists \theta. t_1 \theta \triangleright_t t_2
\end{align*}
\]
A generalization of transition systems $t$ and $t'$ is a triple $(t_g, \theta, \theta')$ where $t_g \theta \equiv t$ and $t_g \theta' \equiv t'$.

$$t \cap_t t' = \begin{cases} (e \rightarrow t^g_1, \ldots, t^g_n, \bigcup_{i=1}^n \theta_i, \bigcup_{i=1}^n \theta'_i), & \text{if } t \lesssim_t t' \\ \text{where } t = e \rightarrow t_1, \ldots, t_n \\
\quad t' = e' \rightarrow t'_1, \ldots, t'_n \\
\quad (t^g_i, \theta_i, \theta'_i) = t_i \cap_t t'_i \\
(\nu \nu_1 \ldots \nu_k, \{\nu := \lambda \nu_1 \ldots \nu_k \cdot t\}, \{\nu := \lambda \nu_1 \ldots \nu_k \cdot t'\}), & \text{otherwise} \\
\text{where } \{\nu_1 \ldots \nu_k\} = \text{bv}(t) \cup \text{bv}(t') \end{cases}$$

Geoff Hamilton, Gavin Mendel-Gleason
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 \texttt{let } v_1 = t_1, \ldots, v_n = t_n \texttt{ in } t_0 \texttt{ into our transition systems.}
- The transition system $t$ is replaced by its generalized form, which is constructed using the $\textit{abstract}_t$ operation:

$$\textit{abstract}_t(t, t') = \texttt{let } v_1 = t_1, \ldots, v_n = t_n \texttt{ in } t_g$$

where $t \sqcap_t t' = (t_g, \{ v_1 := t_1, \ldots, v_n := t_n \}, \theta)$
If the partially constructed process tree $t$ contains a node $\beta$ in which the redex is a function, the node is processed as follows:

\[
\begin{align*}
\text{if } & \exists \alpha \in \text{anc}(t, \beta). S[t(\alpha)] \preceq_t S[t(\beta)] \\
\text{then } & t\{\beta := t(\beta) \xrightarrow{\alpha}\} \\
\text{else if } & \exists \alpha \in \text{anc}(t, \beta). S[t(\alpha)] \preceq_t S[t(\beta)] \\
& \quad \text{then } t\{\alpha := D[t(\text{abstract}_t(S[t(\alpha)], S[t(\beta)]))]\} \\
\text{else } & t\{\beta := t(\beta) \rightarrow D[t(\text{unfold}(t(\beta)))]\}
\end{align*}
\]
Distillation Example: $nrev\;xs$

\[
\begin{align*}
\uparrow & \text{case (nrev \;xs')} \text{ of \ldots} \\
\text{case (case \;xs' \text{ of \ldots}) \text{ of \ldots}} \\
\downarrow & \text{case (app (nrev \;xs'') (x'' : [])) \text{ of \ldots}} \\
\downarrow & \text{case (case (nrev \;xs'') \text{ of \ldots}) \text{ of \ldots}} \\
\downarrow & \text{case (case (case \;xs'' \text{ of \ldots}) \text{ of \ldots}) \text{ of \ldots}} \\
\downarrow & \text{case (case (app (nrev \;xs'''') (x''' : [])) \text{ of \ldots}) \text{ of \ldots}} \\
\downarrow & \text{case (case (case (nrev \;xs'')) \text{ of \ldots}) \text{ of \ldots}) \text{ of \ldots}} \\
\downarrow & \text{case (case (case (case \;xs''') \text{ of \ldots}) \text{ of \ldots}) \text{ of \ldots}}
\end{align*}
\]
Distillation Example: $nrev\ xs$

The following transition system is constructed from the term $\vdash$:

\[
\text{let } vs = nrev\ xs' \text{ in case } vs \text{ of . . .}
\]

\[
\begin{align*}
\text{case } vs \text{ of . . .} & \\
vs = & \, \emptyset \\
v' & : \text{app vs' (} x' \, : \emptyset) \\
\text{app vs' (} x' \, : \emptyset) & \\
\text{case vs' of . . .} & \\
vs' = & \, v'' : vs'' \\
v'' & : \text{app vs'' (} x' \, : \emptyset) \\
\text{app vs'' (} x' \, : \emptyset) & \\
\end{align*}
\]
Distillation Example: \( nrev \, xs \)

The following transition system is constructed from the term \( \oplus \):

\[
\text{let } vs = \text{nrev } xs'' \text{ in case (case } vs \text{ of . . .) of . . .}
\]

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

\[
vs = [] \quad \text{vs} = v' : vs'
\]

\[
x'' : x' : []
\]

\[
v'
\]

\[
\text{case (app } vs' \text{ (x'' : [])) of . . .}
\]

\[
\text{case (app } vs' \text{ (x'' : [])) of . . .}
\]

\[
s = v'' : vs'' \quad vs'' = v' : vs''
\]

\[
x'' : x' : []
\]

\[
v''
\]

\[
v'': \text{ case (app } vs'' \text{ (x'' : [])) of . . .}
\]

\[
\text{case (app } vs'' \text{ (x'' : [])) of . . .}
\]
The following transition system is constructed from the term \( nrev \, xs \):

\[
\text{let } vs = nrev \, xs''' \text{ in case (case (case vs of . . .) of . . .) of . . .}
\]

\[
\begin{align*}
\text{nrev } xs''' & \quad \text{case (case (case vs of . . .) of . . .) of . . .} \\
\text{vs} & \quad \text{vs = } v': vs' \\
\text{x'''} : x'' : x' : [] & \quad \text{v' : case (case (app vs' (x''' : [])) of . . .) of . . .} \\
\text{v'} & \quad \text{case (case vs' of . . .) of . . .} \\
\text{vs'} & \quad \text{vs' = } v'': vs'' \\
\text{x'''} : x'' : x' : [] & \quad \text{v'' : case (case (app vs'' (x''' : [])) of . . .) of . . .} \\
\text{v''} & \quad \text{case (case (app vs'' (x''' : [])) of . . .) of . . .}
\end{align*}
\]
Distillation Example: \( nrev \) \( xs \)

Transition system \( \dagger \) is generalized with respect to transition system \( \ddagger \) to give:

\[
\text{let } vs = nrev \; xs' \text{ in case } vs \text{ of } . . .
\]

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

\[
v' : app \; vs' \; (x' : [])
\]

\[
\text{app } vs' \; (x' : [])
\]

\[
\text{case } vs' \; \text{ of } . . .
\]

\[
v'' : app \; vs'' \; (x' : [])
\]

\[
\text{app } vs'' \; (x' : [])
\]
Transition system $\vdash$ is generalized with respect to transition system $\ast$ to give:

\[
\text{let } vs = \text{nrev } xs'' \text{ in case (case } vs \text{ of . . .) of . . .}
\]

\[
\begin{align*}
&\text{nrev } xs'' \\
&x'' : x' : v \\
&v' \\
&v'' \\
&vs' = [] \\
&vs'' = []
\end{align*}
\]

\[
\begin{align*}
&\text{case (case } vs \text{ of . . .) of . . .} \\
&v' : \text{case (app } vs' (x'' : []) \text{) of . . .} \\
&\text{case (app } vs' (x'' : []) \text{) of . . .} \\
&v'' : \text{case (app } vs'' (x'' : []) \text{) of . . .} \\
&\text{case (app } vs'' (x'' : []) \text{) of . . .}
\end{align*}
\]
Distillation Example: \textit{nrev xs}

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

\begin{verbatim}
\textbf{case} \textbf{xs of}
\hspace{1em} [] \Rightarrow [] \\
\hspace{1em} \mid x' : xs' \Rightarrow f x' xs' []
\textbf{where}
\hspace{1em} f = \lambda x'. \lambda xs'. \lambda v. \textbf{case} \textbf{xs'} of
\hspace{2em} [] \Rightarrow x' : v \\
\hspace{2em} \mid x'' : xs'' \Rightarrow f x'' xs'' (x' : v)
\end{verbatim}

This program has a run-time which is linear with respect to the length of the input list, while the original program is quadratic.
The following program is obtained after applying distillation:

\[ f \; xs \; (g \; ys \; zs) \]

where

\[
\begin{align*}
f & = \lambda xs. \lambda v. \text{case } xs \text{ of} \\
& \quad \; [] \Rightarrow v \\
& \quad \; | x' : xs' \Rightarrow f \; xs' \; (x' : v)
\end{align*}
\]

\[
\begin{align*}
g & = \lambda ys. \lambda zs. \text{case } ys \text{ of} \\
& \quad \; [] \Rightarrow zs \\
& \quad \; | y' : ys \Rightarrow y' : (g \; ys' \; zs)
\end{align*}
\]

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: \( \textit{fib} \ n \)

The \( \textit{fib} \) function:

\[
\text{fib} \ n \quad \text{where} \quad \text{fib} = \lambda n. \begin{cases} 
\text{Zero} & \Rightarrow \text{Succ Zero} \\
\text{Succ } n' & \Rightarrow \text{case } n' \text{ of} \\
& \begin{cases} 
\text{Zero} & \Rightarrow \text{Succ Zero} \\
\text{Succ } n'' & \Rightarrow \text{add} (\text{fib } n') (\text{fib } n'')
\end{cases}
\end{cases}
\]

\[\text{add} = \lambda x. \lambda y. \begin{cases} 
\text{Zero} & \Rightarrow y \\
\text{Succ } x' & \Rightarrow \text{Succ} (\text{add } x' \ y)
\end{cases}\]

is transformed to something similar to the following by distillation:

\[
f \ n \ (\lambda n. \text{Succ } n) (\lambda n. \text{Succ } n) \quad \text{where} \quad f = \lambda n. \lambda g. \lambda h. \begin{cases} 
\text{Zero} & \Rightarrow g \ \text{Zero} \\
\text{Succ } n' & \Rightarrow f \ n' \ h (\lambda n. h (g \ n))
\end{cases}
\]

Geoff Hamilton, Gavin Mendel-Gleason

A Graph-Based Definition of Distillation
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 $\mu$-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.