Distillation with Labelled Transition Systems

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

Neil D. Jones
Computer Science Department,
University of Copenhagen,
2100 Copenhagen, Denmark
neil@diku.dk

Abstract

In this paper, we provide an improved basis for the "distillation" program transformation. It is known that superlinear speedups can be obtained using distillation, but cannot be obtained by other earlier automatic program transformation techniques such as deforestation, positive supercompilation and partial evaluation. We give distillation an improved semantic basis, and explain how superlinear speedups can occur.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—optimization; I.2.2 [*Artificial Intelligence*]: Automatic Programming—program transformation; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Prog. Languages

General Terms Languages, performance, theory.

Keywords Program transformation, superlinear improvement, labelled transitions systems, bisimulation, unfold/fold.

1. Introduction

It is well known that programs written using functional programming languages often make use of intermediate data structures and this can be inefficient. Several program transformation techniques have been proposed to eliminate some of these intermediate data structures; for example *partial evaluation* [11], *deforestation* [25] and *supercompilation* [22]. *Positive supercompilation* [21] is a variant of Turchin's supercompilation that was introduced in an attempt to study and explain the essentials of Turchin's supercompiler. Although strictly more powerful than both partial evaluation and deforestation, Sørensen has shown that positive supercompilation (and hence also partial evaluation and deforestation) can only produce a linear speedup in programs [20]. For superlinear speedup, a more powerful transformation algorithm is needed.

EXAMPLE 1.1. Consider the function call $nrev\ vs$ shown in Figure 1. This reverses the list vs, but the recursive function call $(nrev\ xs')$ is an intermediate data structure, so in terms of time and space usage, it is quadratic with respect to the length of the list vs. A more efficient function that is linear with respect to the length of the list vs is the function $arev$ shown in Figure 1. A number of algebraic transformations have been proposed that can perform this transformation (e.g. [3, 24]) making essential use of "eureka" steps,

```
nrev vs
where
nrev  =  λxs.case xs of
                 []        ⇒  []
               | x' : xs'  ⇒  app (nrev xs') [x']
app   =  λxs.λys.case xs of
                    []        ⇒  ys
                  | x' : xs'  ⇒  x' : (app xs' ys)
arev vs
where
arev  =  λxs.arev' xs []
arev' =  λxs.λys.case xs of
                    []        ⇒  ys
                  | x' : xs'  ⇒  arev' xs' (x' : ys)
```

Figure 1. Alternative Definitions of List Reversal

requiring human insight and not easy to automate (for the given example above, this is done by making use of a specific law stating the associativity of the *app* function). However, none of the automatic program transformation techniques mentioned above (deforestation, positive supercompilation and partial evaluation) are capable of performing this transformation.

The first author defined a transformation algorithm called *distillation* [6–8] to allow transformations such as the above to be performed. However, the correctness of this algorithm was not perfectly clear; it needed a more solid theoretical foundation. In more recent work by both authors [9], a theoretical framework using *labelled transition systems* was defined that can be used to prove the correctness of unfold/fold program transformations using *bisimilarity*, and the correctness of positive supercompilation was proved within this framework. In this paper, we define the distillation algorithm in the same framework and prove its correctness. We also investigate the efficiency improvements that can be obtained by distillation and explain how superlinear speedups can occur.

The paper is structured as follows. In Section 2 we define the higher-order functional language on which the described transformations are performed, and give an operational semantics for this language. In Section 3 we give an alternate characterisation of Turchin's "driving" algorithm. This is done by translating, via symbolic execution, a functional program into an equivalent labelled transition system. The equivalence between source and target semantics uses weak bisimulation, an approach related to work by Howe and Gordon. In Section 4 we define the distillation algorithm and give an example for which a superlinear speedup is obtained. In Section 5 we consider the termination and correctness of the distillation algorithm and show how superlinear speedups are sometimes achieved. Section 6 concludes and discusses related work.

2. Language

In this section, we describe the higher-order functional language that will be used throughout this paper. It uses call-by-name evaluation.

DEFINITION 2.1 (Language Syntax). The syntax of this language is as shown in Figure 2.

$prog$	$::= e_0$ **where** $f_1 = e_1 \ldots f_k = e_k$	Program
e	$::= x$	Variable
	$\mid c\ e_1 \ldots e_k$	Constructor
	$\mid f$	Function Call
	$\mid \lambda x.e$	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	$::= c\ x_1 \ldots x_k$	Pattern

Figure 2. Language Grammar

A program in the language consists of an expression to evaluate and a set of function definitions. For notational convenience we sometimes assume the set of function definitions have been collected into a function environment denoted by Δ. An expression can be a variable, constructor application, function call, λ-abstraction, application or **case**. Variables introduced by λ-abstraction or **case** patterns are *bound*; all other variables are *free*. We write $e_1 \equiv e_2$ if e_1 and e_2 differ only in the names of bound variables.

Each constructor has a fixed arity; for example Nil has arity 0 and $Cons$ has arity 2. In an expression $c\ e_1 \ldots e_n$, n must equal the arity of c. We allow the usual notation [] for Nil, $x : xs$ for $Cons\ x\ xs$ and $[e_1, \ldots, e_k]$ for $Cons\ e_1 \ldots (Cons\ e_k\ Nil)$. The patterns in **case** expressions may not be nested. No variable may appear more than once within a pattern. We assume that the patterns in a **case** expression are non-overlapping and exhaustive. It is also assumed that erroneous terms such as $(c\ e_1 \ldots e_n)\ e$ where c is of arity n and **case** $(\lambda x.e)$ **of** $p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k$ cannot occur.

DEFINITION 2.2 (Substitution). $\theta = \{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$ denotes a *substitution*. If e is an expression, then $e\theta = e\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$ is the result of simultaneously substituting the expressions $e_1, \ldots, e_n$ for the corresponding variables $x_1, \ldots, x_n$, respectively, in the expression e while ensuring that bound variables are renamed appropriately to avoid name capture.

DEFINITION 2.3 (Renaming). $\sigma = \{x_1 \mapsto x'_1, \ldots, x_n \mapsto x'_n\}$, where σ is a bijective mapping, denotes a *renaming*. If e is an expression, then $e\{x_1 \mapsto x'_1, \ldots, x_n \mapsto x'_n\}$ is the result of simultaneously replacing the variables $x_1, \ldots, x_n$ with the corresponding variables $x'_1, \ldots, x'_n$, respectively, in the expression e.

DEFINITION 2.4 (Context). A context C is an expression with a "hole" [] in the place of one sub-expression. $C[e]$ is the expression obtained by replacing the hole in context C with the expression e. Free variables within e may become bound within $C[e]$; if $C[e]$ is closed then we call it a *closing context* for e.

DEFINITION 2.5 (Evaluation Contexts, Redexes and Observables). Evaluation contexts, redexes and observables are defined as shown in Figure 3, where E ranges over evaluation contexts, R ranges over redexes and O ranges over observables.

LEMMA 2.6 (Unique Decomposition Property). For every expression e, either e is an observable or there is a unique evaluation context E and redex e' such that $e = E[e']$.

$$\begin{array}{ll}
E ::= & [] \\
 & \mid E\ e \\
 & \mid \textbf{case}\ E\ \textbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k \\
 & \\
R ::= & f \\
 & \mid (\lambda x.e_0)\ e_1 \\
 & \mid \textbf{case}\ (x\ e_1 \ldots e_n)\ \textbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k \\
 & \mid \textbf{case}\ (c\ e_1 \ldots e_n)\ \textbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k \\
 & \\
O ::= & x\ e_1 \ldots e_n \\
 & \mid c\ e_1 \ldots e_n \\
 & \mid \lambda x.e
\end{array}$$

Figure 3. Syntax of Evaluation Contexts, Redexes and Observables

The call-by-name operational semantics of our language is standard: define an evaluation relation $\Downarrow$ between closed expressions and *values*, where values are expressions in *weak head normal form* (i.e. constructor applications or λ-abstractions). We define a one-step reduction relation $\overset{r}{\rightsquigarrow}$ inductively as shown in Figure 4, where the reduction r can be f (unfolding of function f), c (elimination of constructor c) or $\downarrow e$ (substitution of the expression e).

We use the notation $e \overset{r}{\rightsquigarrow}$ if the expression e reduces, $e \Uparrow$ if e diverges, $e \Downarrow$ if e converges and $e \Downarrow v$ if e evaluates to the value v. These are defined as follows, where $\overset{r}{\rightsquigarrow}^*$ denotes the reflexive transitive closure of $\overset{r}{\rightsquigarrow}$:

$$\begin{array}{ll}
e \overset{r}{\rightsquigarrow}, \text{ iff } \exists e'.e \overset{r}{\rightsquigarrow} e' & e\Downarrow, \text{ iff } \exists v.e \Downarrow v \\
e \Downarrow v, \text{ iff } e \overset{r}{\rightsquigarrow}^* v \wedge \neg(v \overset{r}{\rightsquigarrow}) & e\Uparrow, \text{ iff } \forall e'.e \overset{r}{\rightsquigarrow}^* e' \Rightarrow e' \overset{r}{\rightsquigarrow}
\end{array}$$

DEFINITION 2.7 (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 $C[e_2]\Downarrow$.

3. Labelled Transition Systems

In this section, we use a labelled transition system to characterise the run-time behaviour of a functional program. In the spirit of Gordon [5], we define a particular labelled transition system, in general infinite, that characterises the immediate observations that can be made on expressions to determine their observational equivalence. We extend [5] by allowing free variables in expressions and thus also in actions. Observational equivalence will therefore require that the free variables in actions match up in addition to the bound variables. A similar extension has also been done for *environmental bisimulations* by Sangiorgi et al. [19].

DEFINITION 3.1 (Driven LTS). The *driven LTS* associated with program e_0 is given by $t = (\mathcal{E}, e_0, \rightarrow, Act)$ where:

- $\mathcal{E}$ is the set of *states* of the LTS. Each is an expression, or the end-of-action state **0**.
- t always contains as root the expression e_0. Notation: $root(t)$.
- $\rightarrow \subseteq \mathcal{E} \times Act \times \mathcal{E}$ is a *transition relation* that relates pairs of states by actions according to Figure 5.
- If $e \in \mathcal{E}$ and $e \xrightarrow{\alpha} e'$ then $e' \in \mathcal{E}$.
- Act is a set of actions α that can be *silent* or *non-silent*. A non-silent action may be: x, a variable; c, a constructor; $\#i$, the i^{th} argument in an application; λx, an abstraction over variable x; **case**, a case selector; p, a case branch pattern; or **let**, an abstraction. A silent action may be: τ_f, unfolding of the function f; τ_c, elimination of the constructor c; or $\tau_{\downarrow e}$, substitution of the expression e for the variable at the current redex position.

$$\frac{(f = e) \in \Delta}{f \overset{f}{\rightsquigarrow} e} \qquad ((\lambda x.e_0)\ e_1) \overset{\downarrow e_1}{\rightsquigarrow} (e_0\{x \mapsto e_1\}) \qquad \frac{e_0 \overset{r}{\rightsquigarrow} e_0'}{(e_0\ e_1) \overset{r}{\rightsquigarrow} (e_0'\ e_1)}$$

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

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

Figure 4. One-Step Reduction Relation

$$\mathcal{D}[\![e]\!] = \mathcal{D}'[\![e]\!]\ \emptyset$$

$$\begin{array}{ll}
\mathcal{D}'[\![e = x\ e_1 \ldots e_n]\!]\ \theta & = \begin{cases} e \rightarrow (\tau_{\downarrow\theta(x)}, \mathcal{D}'[\![\theta(x)\ e_1 \ldots e_n]\!]\ \theta), & \text{if } x \in dom(\theta) \\ e \rightarrow (x, \mathbf{0}), (\#1, \mathcal{D}'[\![e_1]\!]\ \theta), \ldots, (\#n, \mathcal{D}'[\![e_n]\!]\ \theta), & \text{otherwise} \end{cases} \\
\mathcal{D}'[\![e = c\ e_1 \ldots e_n]\!]\ \theta & = e \rightarrow (c, \mathbf{0}), (\#1, \mathcal{D}'[\![e_1]\!]\ \theta), \ldots, (\#n, \mathcal{D}'[\![e_n]\!]\ \theta) \\
\mathcal{D}'[\![e = \lambda x.e]\!]\ \theta & = e \rightarrow (\lambda x, \mathcal{D}'[\![e]\!]\ \theta) \\
\mathcal{D}'[\![e = E[f]]\!]\ \theta & = e \rightarrow (\tau_f, \mathcal{D}'[\![E[e]]\!]\ \theta) \\
& \quad \text{where } (f = e) \in \Delta \\
\mathcal{D}'[\![e = E[(\lambda x.e_0)\ e_1]]\!]\ \theta & = \mathcal{D}'[\![E[e_0]]\!]\ (\theta \cup \{x \mapsto e_1\}) \\
\mathcal{D}'[\![e = E[\mathbf{case}\ x\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k]]\!]\ \theta & = \\
\multicolumn{2}{r}{\begin{cases} e \rightarrow (\tau_{\downarrow\theta(x)}, \mathcal{D}'[\![E[\mathbf{case}\ \theta(x)\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k]]\!]\ \theta), & \text{if } x \in dom(\theta) \\ e \rightarrow (\mathbf{case}, \mathcal{D}'[\![x]\!]\ \theta), (p_1, \mathcal{D}'[\![E[e_1]]\!]\ (\theta \cup \{x \mapsto p_1\})), \ldots, (p_k, \mathcal{D}'[\![E[e_k]]\!]\ (\theta \cup \{x \mapsto p_k\})), & \text{otherwise} \end{cases}} \\
\mathcal{D}'[\![e = E[\mathbf{case}\ (x\ e_1 \ldots e_n)\ \mathbf{of}\ p_1 \Rightarrow e_1' \mid \cdots \mid p_k \Rightarrow e_k']]\!]\ \theta & = e \rightarrow (\mathbf{case}, \mathcal{D}'[\![x\ e_1 \ldots e_n]\!]\ \theta), (p_1, \mathcal{D}'[\![E[e_1']]\!]\ \theta), \ldots, (p_k, \mathcal{D}'[\![E[e_k']]\!]\ \theta) \\
\mathcal{D}'[\![e = E[\mathbf{case}\ (c\ e_1 \ldots e_n)\ \mathbf{of}\ p_1 \Rightarrow e_1' \mid \cdots \mid p_k \Rightarrow e_k']]\!]\ \theta & = e \rightarrow (\tau_c, \mathcal{D}'[\![E[e_i']]\!]\ (\theta \cup \{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\})) \\
& \quad \text{where } p_i = c\ x_1 \ldots x_n
\end{array}$$

Figure 5. Driving Rules

Within the actions of a LTS, λ-abstracted variables, **case** pattern variables and **let** variables are *bound*; all other variables are *free*. We use $fv(t)$ and $bv(t)$ to denote the free and bound variables respectively of LTS t. No **let** transitions will appear in the driven LTS; these are only introduced later as a result of generalization.

The LTS notation allows us to determine program behaviour solely from the labels of LTS transitions, without looking at the state contents, i.e., it is not necessary to know the expressions in the node labels. To this end, transitions from a variable or a constructor will lead to **0**, a state from which there are no transitions.

A concise notation: We write $e \xrightarrow{\alpha} e'$ in place of $(e, \alpha, e') \in \rightarrow$. We also write $e \rightarrow (\alpha_1, t_1), \ldots, (\alpha_n, t_n)$ for a LTS with root state e where $t_1 \ldots t_n$ are the LTSs obtained by following the transitions labelled $\alpha_1 \ldots \alpha_n$ respectively from e. We write $e \Rightarrow e'$ iff there is a (possibly empty) sequence of silent transitions leading from e to e'. For each non-silent action α, we write $e_1 \overset{\alpha}{\Rightarrow} e_2$ iff there are e_1' and e_2' such that $e_1 \Rightarrow e_1' \xrightarrow{\alpha} e_2' \Rightarrow e_2$.

Intuitive explanation of the driving rules in Figure 5: These perform normal order reduction on input program e. The result $\mathcal{D}[\![e]\!]$ is an LTS representation of e after evaluation. Computationally, reductions will be performed wherever possible producing silent LTS transitions; non-silent LTS transitions will be generated where no reductions are possible. The generated LTS $\mathcal{D}[\![e]\!]$ will usually be infinite, creating a need for further work to ensure that program transformation terminates.

If a **case** cannot be evaluated, then LTS transitions are generated, with information propagated according to the patterns for each possible **case** branch. θ gives the values of variables within the current term; these values are only substituted for the corresponding variables when they are in the redex position.

Function unfolding, constructor elimination and substitution are irrelevant to the observational equivalence of original and transformed programs. Thus they are represented by silent transitions, and weak bisimulation is used to compare program behaviour.

EXAMPLE 3.2. A portion of the LTS generated by driving the naive list reversal program in Figure 1 is shown in Figure 6.

Note: in some examples in this paper, some silent transitions have been omitted from LTSs. This is only to simplify the presentation and diagrams, and has no significant impact on the results obtained.

DEFINITION 3.3 (LTS Substitution). $\theta = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ denotes a *LTS substitution*. If t is a LTS, then $t\theta = t\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ is the result of simultaneously replacing the transitions $(\rightarrow (x_1, \mathbf{0})), \ldots, (\rightarrow (x_n, \mathbf{0}))$ with the corresponding transitions $(\rightarrow (\tau_{\downarrow t_1}, t_1)), \ldots, (\rightarrow (\tau_{\downarrow t_n}, t_n))$, respectively, in the LTS t while ensuring that bound variables are renamed appropriately to avoid name capture.

DEFINITION 3.4 (Pure LTS). We define a *pure* LTS to be one that contains no **let** transitions. An impure LTS can be converted into a behaviourally equivalent pure LTS using the purify operation $\mathcal{P}$ which is defined as follows:

$$\begin{array}{ll}
\mathcal{P}[\![e \rightarrow (\mathbf{let}, t_0), (x, t_1)]\!] & = \mathcal{P}[\![t_0\{x \mapsto t_1\}]\!] \\
\mathcal{P}[\![e \rightarrow (\alpha_1, t_1), \ldots, (\alpha_n, t_n)]\!] & = e \rightarrow (\alpha_1, \mathcal{P}[\![t_1]\!]), \ldots, \alpha_n, \mathcal{P}[\![t_n]\!])
\end{array}$$

DEFINITION 3.5 (Weak Simulation). Binary relation $\mathcal{R} \subseteq \mathcal{E}_1 \times \mathcal{E}_2$ is a *weak simulation* of pure LTS $(\mathcal{E}_1, e_0^1, \rightarrow_1, Act_1)$ by pure LTS $(\mathcal{E}_2, e_0^2, \rightarrow_2, Act_2)$ if $(e_0^1, e_0^2) \in \mathcal{R}$, and for every pair $(e_1, e_2) \in \mathcal{R}, \alpha \in Act_1, e_1' \in \mathcal{E}_1$:

$$\text{if } e_1 \overset{\alpha}{\Rightarrow} e_1' \text{ then } \exists e_2' \in \mathcal{E}_2 . e_2 \overset{\alpha}{\Rightarrow} e_2' \wedge (e_1', e_2') \in \mathcal{R}$$

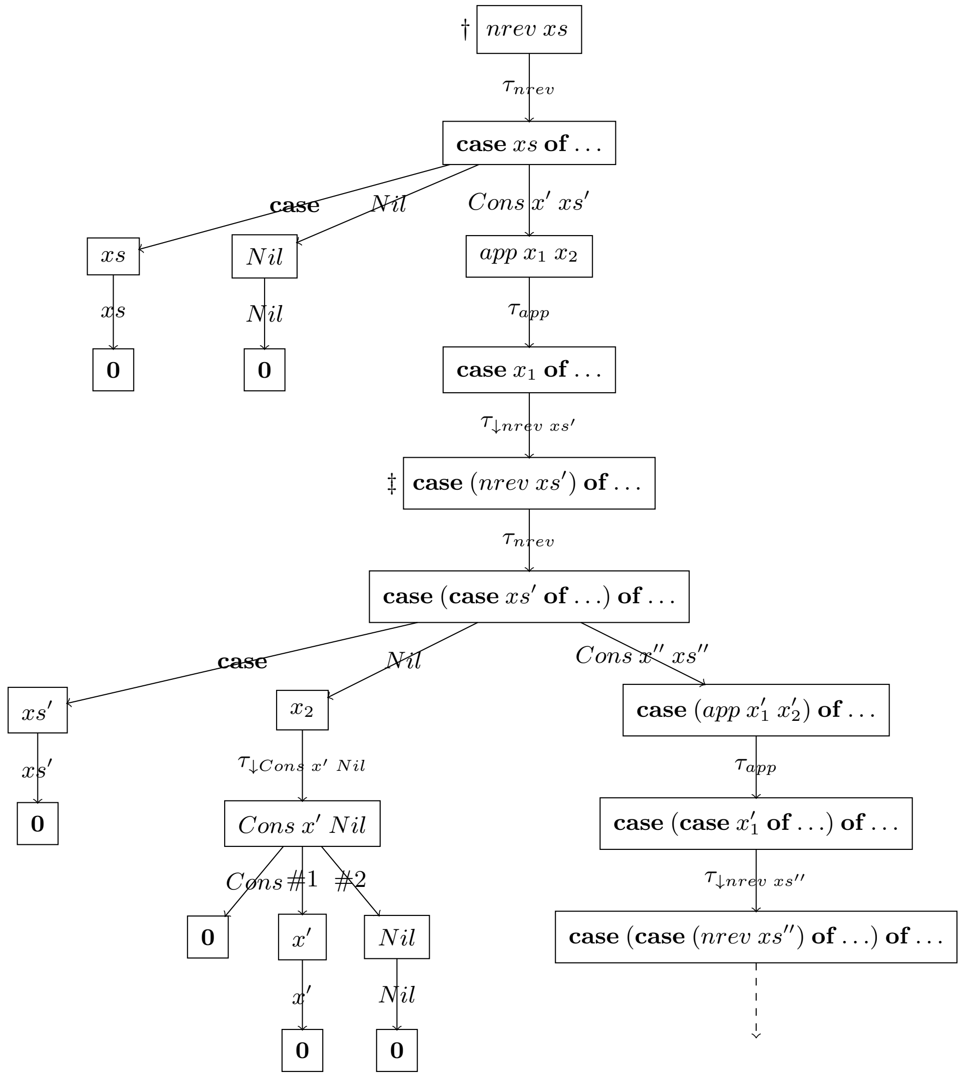

† $nrev\ xs$
τ_{nrev}
case xs **of** ...
case
Nil
$Cons\ x'\ xs'$
xs
Nil
$app\ x_1\ x_2$
xs
Nil
τ_{app}
0
0
case x_1 **of** ...
$\tau_{\downarrow nrev\ xs'}$
‡ **case** $(nrev\ xs')$ **of** ...
τ_{nrev}
case (**case** xs' **of** ...) **of** ...
case
Nil
$Cons\ x''\ xs''$
xs'
x_2
case $(app\ x_1'\ x_2')$ **of** ...
xs'
$\tau_{\downarrow Cons\ x'\ Nil}$
τ_{app}
0
$Cons\ x'\ Nil$
case (**case** x_1' **of** ...) **of** ...
$Cons$ #1 #2
$\tau_{\downarrow nrev\ xs''}$
0
x'
Nil
case (**case** $(nrev\ xs'')$ **of** ...) **of** ...
x'
Nil
0
0

Figure 6. Portion of Driven LTS for $nrev\ xs$

DEFINITION 3.6 (Weak Bisimulation). A *weak bisimulation* is a binary relation $\mathcal{R}$ such that both $\mathcal{R}$ and its inverse $\mathcal{R}^{-1}$ are weak simulations.

DEFINITION 3.7 (Weak Bisimilarity). If there exists a weak bisimulation $\mathcal{R}$ between pure LTSs $(\mathcal{E}_1, e_0^1, \rightarrow_1, Act_1)$ and $(\mathcal{E}_2, e_0^2, \rightarrow_2, Act_2)$, then there exists a unique maximal one, henceforth denoted by $\sim$. A notation: we also write $e_0^1 \sim e_0^2$ in place of $(e_0^1, e_0^2) \in \sim$.

A central property: the weak bisimilarity relation $\sim$ is a congruence, and coincides with observational equivalence.

THEOREM 3.8 (Congruence). $\forall C \,.\, e \sim e' \Rightarrow C[e] \sim C[e']$

PROOF 3.9. Similar to that of Howe [10].

THEOREM 3.10 (Operational Extensionality). $\simeq = \sim$

PROOF 3.11. The proof that $\sim \subseteq \simeq$ follows from the congruence of $\sim$. The reverse inclusion follows by co-induction after showing that $\simeq$ is a bisimulation on $\mathcal{D}[\![e]\!]$.

4. Distillation

In this section, we define the distillation program transformation algorithm within our LTS framework. The algorithm is actually very similar to the positive supercompilation algorithm [21]. The distillation algorithm computes $\mathcal{R}[\![\mathcal{T}[\![\mathcal{D}[\![e]\!]]\!]]\!]$. This can be viewed graphically as follows:

$$\begin{tikzcd} \text{PGM} \arrow[r, "\mathcal{D}"] & \text{LTS} \arrow[loop above, "\mathcal{T}"] \arrow[r, "\mathcal{R}"] & \text{PGM} \end{tikzcd}$$

Given a program $e \in PGM$, the driving rules $\mathcal{D}$ in Figure 5 are applied to construct the LTS $\mathcal{D}[\![e]\!]$ with root e. Transformation rules $\mathcal{T}$ are then applied to the LTS $\mathcal{D}[\![e]\!]$ to transform it into a LTS that contains a finite number of states.

Although the LTS $\mathcal{D}[\![e]\!]$ can be infinite in general, the transformation rules $\mathcal{T}$ will traverse only finite portions of $\mathcal{D}[\![e]\!]$ by working lazily from the root. If a danger of non-termination of the transformation is detected ("the whistle is blown"), then *generalization*

$$t \lesssim^{\rho}_{\sigma} t', \text{ if } (t \lhd^{\rho}_{\sigma} t') \vee (t \bowtie^{\rho}_{\sigma} t')$$

$$\begin{array}{ll}
t \lhd^{\rho}_{\sigma} (e \rightarrow (\tau_f, t')), & \text{if } (f \notin \rho) \wedge (t \lesssim^{\rho \cup \{f\}}_{\sigma} t') \\
t \lhd^{\rho}_{\sigma} (e \rightarrow (\tau_{\downarrow e'}, t')), & \text{if } ((\downarrow e') \notin \rho) \wedge (t \lesssim^{\rho \cup \{\downarrow e'\}}_{\sigma} t') \\
t \lhd^{\rho}_{\sigma} (e \rightarrow (\alpha_1, t_1), \ldots, (\alpha_n, t_n)), & \text{if } \exists i \in \{1 \ldots n\}. t \lesssim^{\rho}_{\sigma} t_i
\end{array}$$

$$\begin{array}{lll}
(e \rightarrow (x, \mathbf{0}), (\#1, t_1), \ldots, (\#n, t_n)) & \bowtie^{\rho}_{\sigma} (e' \rightarrow (x', \mathbf{0}), (\#1, t'_1), \ldots, (\#n, t'_n)), & \text{if } (x \mapsto x') \in \sigma \wedge \forall i \in \{1 \ldots n\}. t_i \lesssim^{\rho}_{\sigma} t'_i \\
(e \rightarrow (\tau_f, t)) & \bowtie^{\rho}_{\sigma} (e' \rightarrow (\tau_f, t')), & \text{if } (f \in \rho) \vee (t \lesssim^{\rho \cup \{f\}}_{\sigma} t') \\
(e \rightarrow (\tau_{\downarrow e_1}, t)) & \bowtie^{\rho}_{\sigma} (e' \rightarrow (\tau_{\downarrow e_2}, t')), & \text{if } (e_1\ \sigma \equiv e_2) \wedge (((\downarrow e_2) \in \rho) \vee (t \lesssim^{\rho \cup \{\downarrow e_2\}}_{\sigma} t')) \\
(e \rightarrow (\alpha_1, t_1), \ldots, (\alpha_n, t_n)) & \bowtie^{\rho}_{\sigma} (e' \rightarrow (\alpha'_1, t'_1), \ldots, (\alpha'_n, t'_n)), & \text{if } \forall i \in \{1 \ldots n\}. \exists \sigma'. (\alpha_i\ \sigma' = \alpha'_i \wedge t_i \lesssim^{\rho}_{\sigma \cup \sigma'} t'_i)
\end{array}$$

Figure 7. Embedding Relation for LTSs

$$\begin{array}{ll}
\mathcal{G}\llbracket e \rightarrow (\tau_f, t) \rrbracket \llbracket e' \rightarrow (\tau_f, t') \rrbracket\ \theta\ \sigma & = \mathcal{G}'\llbracket e' \rightarrow (\tau_f, tg) \rrbracket\ \theta' \\
& \quad \text{where} \\
& \quad (tg, \theta') = \mathcal{G}\llbracket t \rrbracket \llbracket t' \rrbracket\ \theta\ \sigma \\
\mathcal{G}\llbracket e \rightarrow (\tau_{\downarrow e_1}, t) \rrbracket \llbracket e' \rightarrow (\tau_{\downarrow e_2}, t') \rrbracket\ \theta\ \sigma & = \mathcal{G}'\llbracket e' \rightarrow (\tau_{\downarrow e_2}, tg) \rrbracket\ \theta', \text{ if } e_1\ \sigma \equiv e_2 \\
& \quad \text{where} \\
& \quad (tg, \theta') = \mathcal{G}\llbracket t \rrbracket \llbracket t' \rrbracket\ \theta\ \sigma \\
\mathcal{G}\llbracket e \rightarrow (\alpha_1, t_1), \ldots, (\alpha_n, t_n) \rrbracket \llbracket e' \rightarrow (\alpha'_1, t'_1), \ldots, (\alpha'_n, t'_n)) \rrbracket\ \theta\ \sigma & = ((e' \rightarrow (\alpha'_1, tg_1), \ldots, (\alpha'_n, tg_n)), \bigcup_{i=1}^{n} \theta_i) \\
& \quad \text{where} \\
& \quad \forall i \in \{1 \ldots n\}. \exists \sigma'. (\alpha_i\ \sigma' = \alpha'_i \wedge (tg_i, \theta_i) = \mathcal{G}\llbracket t_i \rrbracket \llbracket t'_i \rrbracket\ \theta\ (\sigma \cup \sigma')) \\
\mathcal{G}\llbracket t \rrbracket \llbracket t' \rrbracket\ \theta\ \sigma & = \begin{cases} (\mathcal{D}\llbracket x\ x_1 \ldots x_n \rrbracket, \emptyset), & \text{if } \exists (x \mapsto t_1) \in \theta. t_1 \eqsim^{\emptyset}_{\emptyset} t_2 \\ (\mathcal{D}\llbracket x\ x_1 \ldots x_n \rrbracket, \{x \mapsto t_2\}), & \text{otherwise } (x \text{ is fresh}) \end{cases} \\
& \quad \text{where} \\
& \quad \{x_1 \ldots x_n\} = fv(t') \cap rng(\sigma) \\
& \quad t_2 = root(t') \xrightarrow{\lambda x_1} \ldots \xrightarrow{\lambda x_n} t'
\end{array}$$

$$\begin{array}{ll}
\mathcal{G}'\llbracket e \rightarrow (\alpha, t) \rrbracket\ \emptyset & = e \rightarrow (\alpha, t) \\
\mathcal{G}'\llbracket e \rightarrow (\alpha, t) \rrbracket\ (\{x \mapsto t'\} \cup \theta) & = e \rightarrow (\mathbf{let}, \mathcal{G}'\llbracket e \rightarrow (\alpha, t) \rrbracket\ \theta), (x, t')
\end{array}$$

Figure 8. Rules for Generalization

is performed to convert the current LTS into a new version without danger of non-termination. Overall, the effect of generalization is to ensure that a renaming of a previously encountered LTS will eventually be encountered, at which point *folding* can be applied to convert the current LTS into a new version with only a finite number of states. Rules $\mathcal{R}$ then produce the output program by "residualizing" the resulting finite folded LTS. The syntax and efficiency of this output program may be substantially different from those of the original program.

The main difference between distillation and positive supercompilation [9, 21] is that generalization and folding are performed with respect to LTSs in distillation, while they are performed with respect to expressions in positive supercompilation.

Distillation compares a current LTS with previously encountered ones to determine whether to generalize or to fold. This is particularly tricky as the objects being compared are potentially infinite. However, as any infinite sequence of transitions within either LTS must contain either a function unfolding or a substitution, a record is kept of previously encountered function unfolding and substitution transitions. If one of these recorded transitions is encountered again no further comparison is done.

4.1 Generalization

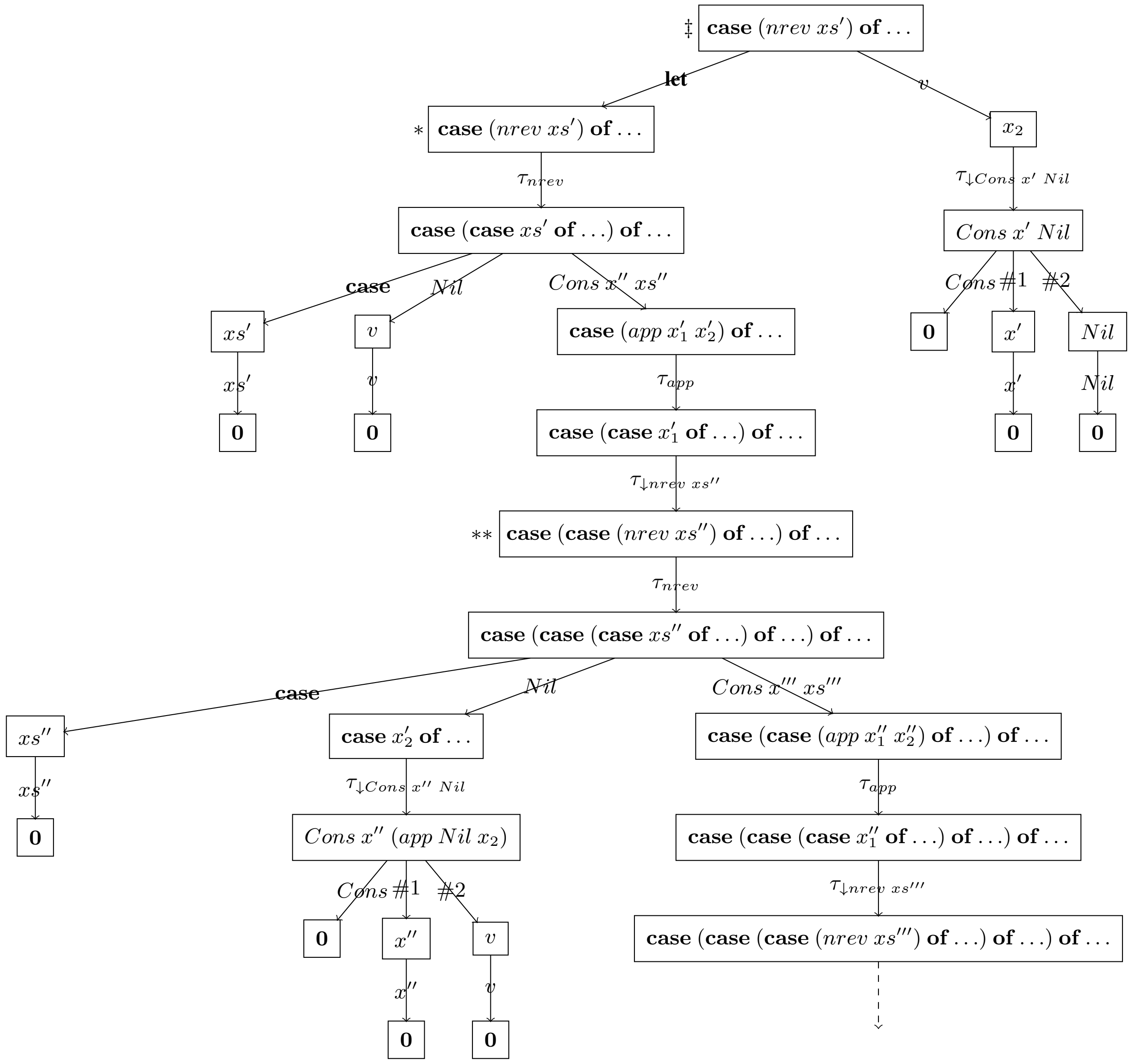
Generalization is performed when an LTS is encountered in which a previously encountered LTS is embedded. The embedding relation on LTSs is defined as follows. Its purpose is to guarantee termination of distillation; more on this in Section 5.

DEFINITION 4.1 (LTS Embedding). LTS t_1 is *embedded* within LTS t_2 iff there is a renaming σ such that $t_1 \lesssim^{\emptyset}_{\sigma} t_2$, where the reflexive, transitive and anti-symmetric relation $\lesssim^{\rho}_{\sigma}$ is defined as shown in Figure 7.

One LTS is embedded within another by this relation if either *diving* (denoted by $\lhd^{\rho}_{\sigma}$) or *coupling* (denoted by $\bowtie^{\rho}_{\sigma}$) can be performed. In the rules for diving, a transition can be followed from the current state in the embedding LTS that is not followed from the current state in the embedded one. In the rules for coupling, the same transitions are possible from each of the current states. Matching transitions may contain different free variables; in this case the transition labels should respect the renaming σ. Matching transitions may also introduce bound variables (λ, **case** and **let**); in these cases the corresponding bound variables are added to the renaming σ. The parameter ρ is used to ensure termination of the rules; a record is kept in ρ of previously encountered function unfolding and substitution transitions and if one of these transitions is re-encountered no further rules are applied. The rules for diving and coupling are applied in top-down order, where the final rule is a catch-all.

DEFINITION 4.2 (Generalization of LTSs). Figure 8 defines function $\mathcal{G}\llbracket t \rrbracket \llbracket t' \rrbracket\ \theta\ \sigma$ that generalizes LTS t' with respect to LTS t, where θ is the set of previous generalizations that can be reused and σ is a renaming s.t. $t \bowtie^{\emptyset}_{\sigma} t'$.

Figure 8 performs generalization incrementally from the roots of the two LTSs, the "increment" being the interval between function unfolding and substitution transitions. The generalizations that are



‡ case (nrev xs′) of ...
let
v
∗ case (nrev xs′) of ...
x₂
τnrev
τ↓Cons x′ Nil
case (case xs′ of ...) of ...
Cons x′ Nil
case
Nil
Cons x″ xs″
Cons #1 #2
xs′
v
case (app x′₁ x′₂) of ...
0
x′
Nil
τapp
0
0
case (case x′₁ of ...) of ...
0
0
τ↓nrev xs″
∗∗ case (case (nrev xs″) of ...) of ...
τnrev
case (case (case xs″ of ...) of ...) of ...
case
Nil
Cons x‴ xs‴
xs″
case x′₂ of ...
case (case (app x″₁ x″₂) of ...) of ...
τ↓Cons x″ Nil
τapp
0
Cons x″ (app Nil x₂)
case (case (case x″₁ of ...) of ...) of ...
Cons #1 #2
τ↓nrev xs‴
0
x″
v
case (case (case (nrev xs‴) of ...) of ...) of ...
0
0

Figure 9. Portion of Generalized LTS for $nrev\ xs$

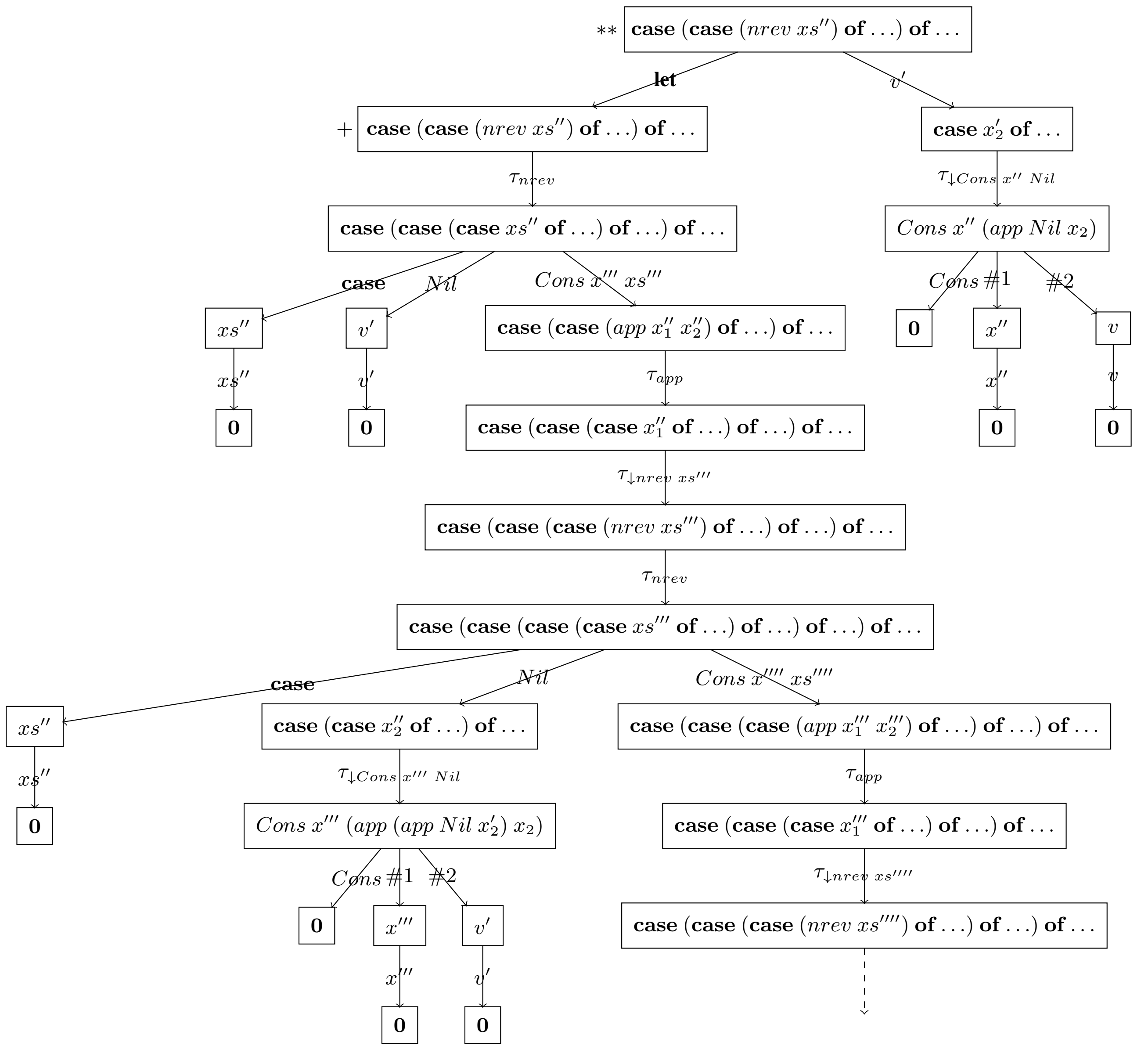
performed within each of these intervals are extracted using **let**s at the start of the interval. These generalizations are then passed down for use in further generalizations. If the same generalization is subsequently encountered again, the same generalization variable is reused. The **let**s that are introduced will therefore appear just before function unfolding and substitution transitions and will be distributed throughout the generalized LTS.

The generalization rules $\mathcal{G}$ are applied in top-down order. Generalization is performed as follows. If two corresponding states have the same transitions, these transitions remain in the resulting generalized LTS, and the corresponding targets of these transitions are then generalized. Matching transitions may contain different free variables; in this case the transition labels should respect the renaming σ. Matching transitions may also introduce bound variables (λ, **case** and **let**); in these cases the corresponding bound variables are added to the renaming σ. Unmatched states are replaced by variable applications. The arguments of these applications are the variables of the second LTS which differ from those of the first LTS according to the renaming σ; this also ensures that bound variables are not extracted outside their binders. It is assumed that the new variables introduced are all different and distinct from the original program variables.

EXAMPLE 4.3. In the driven LTS of Figure 6, it can be seen that the LTS † is embedded in the LTS ‡. Figure 9 shows a portion of the LTS resulting from generalizing LTS ‡ with respect to LTS †. Within the result, one can see a further embedding: LTS $*$ is embedded in the LTS $**$. Next step: Figure 10 shows a portion of the result of generalizing the LTS $**$ with respect to LTS $*$.

4.2 Folding

Folding is performed when a LTS is encountered that is a renaming of a previously encountered LTS. The renaming relation on LTSs is defined as follows.



** case (case (nrev xs'') of ...) of ...
let
v'
+ case (case (nrev xs'') of ...) of ...
case x'_2 of ...
τ_nrev
τ_↓Cons x'' Nil
case (case (case xs'' of ...) of ...) of ...
Cons x'' (app Nil x_2)
case
Nil
Cons x''' xs'''
Cons
#1
#2
xs''
v'
case (case (app x''_1 x''_2) of ...) of ...
0
x''
v
τ_app
case (case (case x''_1 of ...) of ...) of ...
τ_↓nrev xs'''
case (case (case (nrev xs''') of ...) of ...) of ...
τ_nrev
case (case (case (case xs''' of ...) of ...) of ...) of ...
case
Nil
Cons x'''' xs''''
case (case x''_2 of ...) of ...
case (case (case (app x'''_1 x'''_2) of ...) of ...) of ...
τ_↓Cons x''' Nil
τ_app
Cons x''' (app (app Nil x'_2) x_2)
case (case (case x'''_1 of ...) of ...) of ...
τ_↓nrev xs''''
x'''
case (case (case (nrev xs'''') of ...) of ...) of ...

Figure 10. Portion of Generalized LTS for $nrev\ xs$

$$
\begin{array}{lll}
(e \rightarrow (x, \mathbf{0}), (\#1, t_1), \ldots, (\#n, t_n)) \eqsim^{\rho}_{\sigma} (e' \rightarrow (x', \mathbf{0}), (\#1, t'_1), \ldots, (\#n, t'_n)), & \text{if } (x \mapsto x') \in \sigma \wedge \forall i \in \{1 \ldots n\}. t_i \eqsim^{\rho}_{\sigma} t'_i \\
(e \rightarrow (\tau_f, t)) \eqsim^{\rho}_{\sigma} (e' \rightarrow (\tau_f, t')), & \text{if } (f \in \rho) \vee (t \eqsim^{\rho \cup \{f\}}_{\sigma} t') \\
(e \rightarrow (\tau_{\downarrow e_1}, t)) \eqsim^{\rho}_{\sigma} (e' \rightarrow (\tau_{\downarrow e_2}, t')), & \text{if } (e_1\ \sigma \equiv e_2) \wedge (((\downarrow e_2) \in \rho) \vee (t \eqsim^{\rho \cup \{\downarrow e_2\}}_{\sigma} t')) \\
(e \rightarrow (\alpha_1, t_1), \ldots, (\alpha_n, t_n)) \eqsim^{\rho}_{\sigma} (e' \rightarrow (\alpha'_1, t'_1), \ldots, (\alpha'_n, t'_n)), & \text{if } \forall i \in \{1 \ldots n\}. \exists \sigma'. (\alpha_i\ \sigma' = \alpha'_i \wedge t_i \eqsim^{\rho}_{\sigma \cup \sigma'} t'_i)
\end{array}
$$

Figure 11. Renaming Relation for LTSs

DEFINITION 4.4 (LTS Renaming). LTS t_1 is a *renaming* of LTS t_2 iff there is a renaming σ such that $t_1 \eqsim^{\emptyset}_{\sigma} t_2$, where the reflexive, transitive and symmetric relation $\eqsim^{\rho}_{\sigma}$ is defined in Figure 11.

The rules for renaming are applied in top-down order, where the final rule is a catch-all. Two LTSs are renamings of each other if the same transitions are possible from each corresponding state (modulo variable renaming according to the renaming σ). The definition also handles transitions that introduce bound variables (λ, **case** and **let**); in these cases the corresponding bound variables are added to the renaming σ. The parameter ρ is used to ensure termination of the rules in the same manner as the rules for embedding.

DEFINITION 4.5 (Folding of LTSs). Folding with respect to the previously encountered LTS t with the renaming σ is performed by the function $\mathcal{F}[\![t]\!]\ \sigma$, which is defined as follows.

$$
\begin{array}{ll}
\mathcal{F}[\![t]\!]\ \emptyset & = t \\
\mathcal{F}[\![t]\!]\ (\{x \mapsto x'\} \cup \sigma) & = root(t) \rightarrow (\mathbf{let}, \mathcal{F}[\![t]\!]\ \theta), (x, x' \xrightarrow{x'} \mathbf{0})
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{T}\llbracket e\rrbracket = \mathcal{T}'\llbracket e\rrbracket\ \emptyset\ \emptyset & \\
\mathcal{T}'\llbracket t = e \rightarrow (\tau_f, t')\rrbracket\ \pi\ \theta & = \begin{cases} \mathcal{F}\llbracket t''\rrbracket\ \sigma, & \text{if } \exists t'' \in \pi, \sigma.t'' \eqsim^{\emptyset}_{\sigma} t \\ \mathcal{T}'\llbracket \mathcal{G}\llbracket t''\rrbracket\llbracket t\rrbracket\ \theta\ \sigma\rrbracket\ \pi\ \phi, & \text{if } \exists t'' \in \pi, \sigma.t'' \bowtie^{\emptyset}_{\sigma} t \\ e \rightarrow (\tau_f, \mathcal{T}'\llbracket t'\rrbracket\ (\pi \cup \{t\})\ \theta), & \text{otherwise} \end{cases} \\
\mathcal{T}'\llbracket t = e \rightarrow (\tau_{\downarrow e'}, t')\rrbracket\ \pi\ \theta & = \begin{cases} \mathcal{F}\llbracket t''\rrbracket\ \sigma, & \text{if } \exists t'' \in \pi, \sigma.t'' \eqsim^{\emptyset}_{\sigma} t \\ \mathcal{T}'\llbracket \mathcal{G}\llbracket t''\rrbracket\llbracket t\rrbracket\ \theta\ \sigma\rrbracket\ \pi\ \phi, & \text{if } \exists t'' \in \pi, \sigma.t'' \bowtie^{\emptyset}_{\sigma} t \\ e \rightarrow (\tau_{\downarrow e'}, \mathcal{T}'\llbracket t'\rrbracket\ (\pi \cup \{t\})\ \theta), & \text{otherwise} \end{cases} \\
\mathcal{T}'\llbracket t = e \rightarrow (\mathbf{let}, t_0), (x, t_1)\rrbracket\ \pi\ \theta & = \begin{cases} \mathcal{T}'\llbracket t_0\{x \mapsto x'\}\rrbracket\ \pi\ \theta, & \text{if } \exists (x' \mapsto t_2) \in \theta. t_1 \eqsim^{\emptyset}_{\emptyset} t_2 \\ e \rightarrow (\mathbf{let}, \mathcal{T}'\llbracket t_0\rrbracket\ \pi\ (\theta \cup \{x \mapsto t_1\})), (x, \mathcal{T}'\llbracket t_1\rrbracket\ \pi\ \theta), & \text{otherwise} \end{cases} \\
\mathcal{T}'\llbracket e \rightarrow (\alpha_1, t_1), \ldots, (\alpha_n, t_n)\rrbracket\ \pi\ \theta & = e \rightarrow (\alpha_1, \mathcal{T}'\llbracket t_1\rrbracket\ \pi\ \theta), \ldots, (\alpha_n, \mathcal{T}'\llbracket t_n\rrbracket\ \pi\ \theta)
\end{array}
$$

Figure 12. Transformation Rules

$$
\begin{array}{ll}
\mathcal{R}\llbracket e\rrbracket = \mathcal{R}'\llbracket e\rrbracket\ \emptyset & \\
\mathcal{R}'\llbracket e \rightarrow (x, \mathbf{0}), (\#1, t_1), \ldots, (\#n, t_n)\rrbracket\ \varepsilon & = x\ (\mathcal{R}'\llbracket t_1\rrbracket\ \varepsilon) \ldots (\mathcal{R}'\llbracket t_n\rrbracket\ \varepsilon) \\
\mathcal{R}'\llbracket e \rightarrow (c, \mathbf{0}), (\#1, t_1), \ldots, (\#n, t_n)\rrbracket\ \varepsilon & = c\ (\mathcal{R}'\llbracket t_1\rrbracket\ \varepsilon) \ldots (\mathcal{R}'\llbracket t_n\rrbracket\ \varepsilon) \\
\mathcal{R}'\llbracket e \rightarrow (\lambda x, t)\rrbracket\ \varepsilon & = \lambda x.(\mathcal{R}'\llbracket t\rrbracket\ \varepsilon) \\
\mathcal{R}'\llbracket e \rightarrow (\mathbf{case}, t_0)(p_1, t_1), \ldots, (p_n, t_k)\rrbracket\ \varepsilon & = \mathbf{case}\ (\mathcal{R}'\llbracket t_0\rrbracket\ \varepsilon)\ \mathbf{of}\ p_1 \Rightarrow (\mathcal{R}'\llbracket t_1\rrbracket\ \varepsilon) \mid \cdots \mid p_k \Rightarrow (\mathcal{R}'\llbracket t_k\rrbracket\ \varepsilon) \\
\mathcal{R}'\llbracket e \rightarrow (\mathbf{let}, t_0), (x, t_1)\rrbracket\ \varepsilon & = (\mathcal{R}'\llbracket t_0\rrbracket\ \varepsilon)\{x \mapsto (\mathcal{R}'\llbracket t_1\rrbracket\ \varepsilon)\} \\
\mathcal{R}'\llbracket e \rightarrow (\tau_f, t)\rrbracket\ \varepsilon & = \begin{cases} f'\ x_1 \ldots x_n, \text{ if } \exists (f'\ x_1 \ldots x_n = e) \in \varepsilon \\ f'\ x_1 \ldots x_n\ \mathbf{where}\ f' = \lambda x_1 \ldots x_n.(\mathcal{R}'\llbracket t\rrbracket\ (\varepsilon \cup \{f'\ x_1 \ldots x_n = e\})), \\ \qquad \text{otherwise } (f' \text{ is fresh, } \{x_1 \ldots x_n\} = fv(t)) \end{cases} \\
\mathcal{R}'\llbracket e \rightarrow (\tau_{\downarrow e'}, t)\rrbracket\ \varepsilon & = \begin{cases} f\ x_1 \ldots x_n, \text{ if } \exists (f\ x_1 \ldots x_n = e) \in \varepsilon \\ f\ x_1 \ldots x_n\ \mathbf{where}\ f = \lambda x_1 \ldots x_n.(\mathcal{R}'\llbracket t\rrbracket\ (\varepsilon \cup \{f\ x_1 \ldots x_n = e\})), \\ \qquad \text{otherwise } (f \text{ is fresh, } \{x_1 \ldots x_n\} = fv(t)) \end{cases} \\
\mathcal{R}'\llbracket e \rightarrow (\tau_c, t)\rrbracket\ \varepsilon & = \mathcal{R}'\llbracket t\rrbracket\ \varepsilon
\end{array}
$$

Figure 13. Rules For Residualization

EXAMPLE 4.6. The LTS $+$ in Figure 10 is a renaming of the LTS $*$ in Figure 9, so folding will be applied at this point.

4.3 Transformation Rules

The rules to transform a driven LTS are shown in Figure 12. It is assumed that rules $\mathcal{T}'$ are applied in top-down order, where the final rule is a catch-all. The driven LTS is traversed lazily from the root. As any infinite sequence of steps must include function unfolding or substitution, all LTSs in which there is such a transition from the root are recorded in the environment π. If a previously recorded LTS is a renaming of the current one then folding is done, creating a transition back to a previously encountered state. If a previously recorded LTS is coupled with the current one, the current LTS is generalized, and we continue on the generalized LTS. In these rules, the parameter θ is used to record all components previously extracted by generalization. If the same component is subsequently extracted, then it is replaced by the same variable.

4.4 Residualization

DEFINITION 4.7 (Residual Program Construction). A residual program can be constructed from a folded LTS using rules $\mathcal{R}$ in Figure 13. In these rules, the parameter ε contains the set of new function calls that have been created, and associates them with the expressions they replace. On re-encountering an associated expression, it is also replaced by the corresponding function call.

EXAMPLE 4.8. The program constructed from the LTS resulting from the distillation of $nrev\ xs$, via Figure 10 and Example 4.6, is shown in Figure 14. We can see that the distilled program is a super-linear improvement over the original.

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

Figure 14. Result of Applying Distillation to $nrev\ xs$

5. Termination, Correctness and Efficiency

In this section, we consider the termination and correctness of the distillation algorithm and the efficiency gains that can be obtained.

5.1 Termination

THEOREM 5.1 (Termination of Distillation). The distillation algorithm always terminates.

PROOF 5.2. We firstly need to show that in any infinite sequence of LTSs encountered during transformation $t_0, t_1, \ldots$ there definitely exists some $i < j, \sigma$ where $t_i \bowtie^{\emptyset}_{\sigma} t_j$, so an embedding must eventually be encountered and transformation will not continue indefinitely without the need for generalization. This amounts to proving that the embedding relation $t_i \bowtie^{\emptyset}_{\sigma} t_j$ is a *well-quasi order*.

DEFINITION 5.3 (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 5.4 ($\bowtie^{\emptyset}_{\sigma}$ is a Well-Quasi Order). The embedding relation $\bowtie^{\emptyset}_{\sigma}$ is a *well-quasi order* on any sequence of LTSs that are encountered during transformation.

PROOF 5.5. Nash-Williams has previously shown that the infinite trees are well-quasi ordered under the topological minor relation [15]. The sequence of LTSs that are encountered during transformation obviously belong to the topological minor relation, as each successive LTS is a sub-tree of the previous one. The embedding relation $\lesssim^{\emptyset}_{\sigma}$ is therefore a well-quasi order. To show that $\bowtie^{\emptyset}_{\sigma}$ is also a well-quasi order, we additionally need to show that in the sequence of LTSs encountered during transformation, the transition from the root state must eventually re-occur. This must be the case since in any infinite sequence of transitions, either a function unfolding or a substitution transition must re-occur.

When we compare LTSs for possible embeddings or renamings, only a finite portion of the LTSs are inspected, namely the portion up to a re-occurrence of a function unfolding or substitution transition. We call this portion the *core component* of the LTS. In order to prove that the distillation algorithm terminates, we need to ensure that there is a bound on the size of the core components that are used to compare for possible embeddings. If there is such a bound, then a renaming of a previous core component must eventually be encountered at which point folding can be applied and the transformation will terminate.

DEFINITION 5.6 (Size of Core Component of LTS). The *size* of the core component of LTS t, denoted by $|t|$, is defined as follows:

$$|t| = \mathcal{S}[\![t]\!]\ \emptyset$$

$$\mathcal{S}[\![e \to \tau_f t]\!]\ \rho = \begin{cases} 0, & \text{if } f \in \rho \\ 1 + \mathcal{S}[\![t]\!]\ (\rho \cup \{f\}), & \text{otherwise} \end{cases}$$
$$\mathcal{S}[\![e \to \tau_{\downarrow e'} t]\!]\ \rho = \begin{cases} 0, & \text{if } (\downarrow e') \in \rho \\ 1 + \mathcal{S}[\![t]\!]\ (\rho \cup \{\downarrow e'\}), & \text{otherwise} \end{cases}$$
$$\mathcal{S}[\![e \to (\alpha_1, t_1), \ldots, (\alpha_n, t_n)]\!]\ \rho = 1 + \Sigma_{i=1}^{n} \mathcal{S}[\![t_i]\!]\ \rho$$

LEMMA 5.7. There is a bound on the size of the core components of the LTSs that are encountered during transformation.

PROOF 5.8. We define an extraction operation $\mathcal{X}$ which removes **let**s from a LTS as follows:

$$\begin{array}{ll} \mathcal{X}[\![e \to (\mathbf{let}, t_0), (x, t_1)]\!] & = \mathcal{X}[\![t_0]\!] \\ \mathcal{X}[\![e \to (\alpha_1, t_1), \ldots, (\alpha_n, t_n)]\!] & = e \to (\alpha_1, \mathcal{X}[\![t_1]\!]), \ldots, (\alpha_n, \mathcal{X}[\![t_n]\!]) \end{array}$$

If a LTS t is encountered where $t' \bowtie^{\emptyset}_{\sigma} t$ for a previously encountered LTS t', then generalization will be performed yielding a generalized LTS tg, where $|\mathcal{X}[\![tg]\!]| \leq |t'|$. There will therefore be a bound on the size of the core components of the LTSs that are encountered during transformation if a point is reached after which no new **let**s are introduced by successive generalization passes. This will be the case if there is a bound on the size of the components that are extracted in each successive generalization pass as extracted components must eventually re-occur, and the same generalization variables will be reused without introducing new **let**s. This must be the case as extracted components which would otherwise grow in size will be generalized with respect to previously occurring extracted components, and can therefore be no larger than them.

5.2 Correctness

In order to show that the LTS resulting from transformation is observationally equivalent to the original LTS, we need to show that they are weakly bisimilar. Weak bisimilarity is used because silent transitions in the original LTS may have been removed by transformation. To prove this, the transformed LTS has to be purified using the operation $\mathcal{P}$ as given in Definition 3.4. The correctness of the algorithm can now be stated as follows.

THEOREM 5.9 (Correctness of Distillation).

$$\forall t \in LTS\ .\ \mathcal{P}[\![t]\!] \sim \mathcal{P}[\![\mathcal{T}[\![t]\!]]\!]$$

PROOF 5.10. The proof is by induction on the transformation rules $\mathcal{T}'$. The rules that do not involve generalization or folding can be proved straightforwardly by application of the inductive hypothesis. For the rules involving generalization and folding, we need to prove the following:

LEMMA 5.11 (Correctness of Generalization).

$$\forall t, t'', \theta, \sigma\ .\ \mathcal{P}[\![t]\!] \sim (\mathcal{P}[\![(\mathcal{G}[\![t'']\!][\![t]\!]\ \theta\ \sigma)\theta]\!])$$

LEMMA 5.12 (Correctness of Folding).

$$\forall t, t'', \sigma\ .\ (t'' \eqsim^{\emptyset}_{\sigma} t) \Rightarrow \mathcal{P}[\![t]\!] \sim (\mathcal{P}[\![\mathcal{F}[\![t'']\!]\ \sigma]\!])$$

These can be shown by straightforward induction on the LTS t.

5.3 Efficiency

We now look at the efficiency gains that can be obtained using distillation, and try to give some intuition as to how superlinear speedups are obtained. Speedups are obtained in distillation through two mechanisms; removing silent transitions from the LTS representation of a program, and the identification of extracted components resulting from generalization. Sørensen has previously shown that removing silent transitions can only produce a linear speedup in programs [20]. This is because there will only be a constant number of such silent transitions between each recursive call of a function. Hence, only linear speedups are possible. It is therefore the identification of extracted components resulting from generalization that produces superlinear speedups.

DEFINITION 5.13 (Run-Time). We define $t(e\theta)$, the *run-time* of program e with ground data given by the substution θ, as the number of states in the driven LTS $\mathcal{D}[\![e\theta]\!]$.

THEOREM 5.14 (Superlinearity of Speedups).

$$\exists e \in Exp. \mathcal{R}[\![\mathcal{T}[\![\mathcal{D}[\![e]\!]]\!]]\!] = e' \Rightarrow \nexists n. \forall \theta. (n \times t(e'\theta)) \geq t(e\theta)$$

PROOF 5.15. The components that are extracted from between successive function calls and subsequently identified can be of greater than constant complexity. The higher complexity of these components may be entirely due to silent transitions that will be removed by transformation, producing components that can be subsequently identified. For example, the components that are extracted from between successive calls in the naive reverse example running through this paper correspond to the terms $Cons\ x\ Nil$, $app\ Nil\ (Cons\ x\ Nil)$, $app\ Nil\ (app\ Nil\ (Cons\ x\ Nil))$, $\ldots$ and are therefore of linear complexity with respect to the size of the input list. However, all of these components can be transformed to a component corresponding to the term $Cons\ x\ Nil$ and subsequently identified.

If components are extracted from between N recursive calls of a function on input data of size N, and the run-time of the extracted and remaining components within each function call are given by $f_1(N)$ and $f_2(N)$ respectively, then the run-time of the function before generalization will be $t_1 = \Sigma_{i=0}^{N}(f_1(i) + f_2(i))$. If the extracted components are of greater than constant complexity then $\nexists n. \forall i. f_1(i) \leq n$. If these components are transformed to become of complexity $f_1(0)$ and subsequently identified, then the run-time of the function after generalization will be $t_2 = f_1(0) + \Sigma_{i=0}^{N} f_2(i)$. Therefore $\nexists n. n \times t_2 \geq t_1$.

6. Conclusion and Related Work

We have presented a semantic basis for the distillation program transformation algorithm and explained why superlinear speedups can be achieved that cannot be obtained by other fully automatic program transformation techniques such as deforestation, positive supercompilation and partial evaluation.

Previous works [1, 2, 13, 20, 26] have noted that the unfold/fold transformation methodology is incomplete; some programs cannot be synthesized from each other. This is partly due to the linear relationship that must exist between the original and transformed programs. This paper has shown how to overcome this restriction.

Sands' theory of local improvement [18] was seminal in proving correctness of unfold/fold program transformations. However, it is complicated by the use of folding, which causes a loss of efficiency locally, but not globally if always done in conjunction with a corresponding unfold. Also, tying the correctness of transformations to an improvement in efficiency restricts this approach to particular program semantics and transformation techniques.

Distillation can be considered as a method that works on the meta-level above supercompilation, where instead of sets of states ("configurations") the results of supercompilation ("residual graphs") are manipulated. There have been several attempts to do this, the first one by Turchin himself using *walk grammars* [23]. In this approach, traces through residual graphs are represented by regular grammars that are subsequently analysed and simplified. This approach is also capable of achieving superlinear speedups, but no automatic procedure is defined for it; the outlined heuristics and strategies may not terminate.

The most recent work on building a meta-level above supercompilation is by Klyuchnikov and Romanenko [12]. They construct a hierarchy of supercompilers in which lower level supercompilers are used to prove lemmas about term equivalences, and higher level supercompilers utilise these lemmas by rewriting according to the term equivalences (similar to the "second order replacement method" defined by Kott [14]). This approach is also capable of achieving superlinear speedups, but again no automatic procedure is defined for it; the need to find and apply appropriate lemmas introduces infinite branching into the search space, and various heuristics have to be used to used to try to limit this search.

Logic program transformation is closely related, and the equivalence of partial deduction and driving has been argued by Glück and Sørensen [4]. Superlinear speedups can be achieved in logic program transformation by *goal replacement* [16, 17]: replacing one logical clause with another to facilitate folding. Techniques similar to the notion of "higher level supercompilation" [12] have been used to prove correctness of goal replacement, but have similar problems regarding the search for appropriate lemmas.

Acknowledgements

This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre (www.lero.ie).

References

[1] Amtoft, T.: Sharing of Computations. PhD thesis. DAIMI, Aarhus University. (1993)

[2] Andersen, L.O., and Gomard, C.K.: Speedup Analysis in Partial Evaluation: Preliminary Results. In: Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM). pp. 1–7 (1992)

[3] Burstall, R., Darlington, J.: A Transformation System for Developing Recursive Programs. Journal of the ACM 24(1), 44–67 (1977)

[4] Glück, R., Sørensen, M.H.: Partial Deduction and Driving are Equivalent. In: Proceedings of the 6th International Symposium on Programming Language Implementation and Logic Programming (PLILP). pp. 165–181 (1994)

[5] Gordon, A.D.: Bisimilarity as a Theory of Functional Programming. Theoretical Computer Science 228(1–2), 5–47 (1999)

[6] Hamilton, G.W.: Distillation: Extracting the Essence of Programs. In: Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM). pp. 61–70 (2007)

[7] Hamilton, G.W.: Extracting the Essence of Distillation. In: Proceedings of the Seventh International Andrei Ershov Memorial Conference: Perspectives of System Informatics (PSI). Lecture Notes in Computer Science 5947, 151–164 (2009)

[8] Hamilton, G.W., Mendel-Gleason, G.: A Graph-Based Definition of Distillation. In: Proceedings of the Second International Workshop on Metacomputation in Russia (2010)

[9] Hamilton, G.W., Jones, N.D.: Proving the Correctness of Unfold/Fold Program Transformations Using Bisimulation. In: Proceedings of the Eighth International Andrei Ershov Memorial Conference: Perspectives of System Informatics (PSI). Lecture Notes in Computer Science 7162, 150–166 (2011)

[10] Howe, D.J.: Proving Congruence of Bisimulation in Functional Programming Languages. Information and Computation 124(2), 103–112 (1996)

[11] Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice Hall (1993)

[12] Klyuchnikov, I., Romanenko, S.: Towards Higher-Level Supercompilation. In: Proceedings of the Second International Workshop on Metacomputation in Russia (2010)

[13] Kott, L.: A System for Proving Equivalences of Recursive Programs. In: Proceedings of the Fifth Conference on Automated Deduction (CADE). pp 63–69 (1980)

[14] Kott, L.: Unfold/Fold Transformations. In: Nivat, M., Reynolds, J., eds: Algebraic Methods in Semantics. ch. 12 pp. 412–433. CUP. (1985)

[15] Nash-Williams, C. St.J. A.: On Well-Quasi-Ordering Infinite Trees. Proceedings of the Cambridge Philosophical Society 61, 697–720 (1965)

[16] Pettorossi, A., Proietti, M.: A Theory of Totally Correct Logic Program Transformations In: Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM). pp. 159–168 (2004)

[17] Roychoudhury, A., Kumar, K.N., Ramakrishnan, C.R., Ramakrishnan, I.V.: An Unfold/Fold Transformation Framework for Definite Logic Programs. ACM Transactions on Programming Language Systems 26(3), 464–509 (2004)

[18] Sands, D.: Proving the Correctness of Recursion-Based Automatic Program Transformations. Theoretical Computer Science 167(1–2), 193–233 (1996)

[19] Sangiorgi,D., Kobayashi, N., Sumii, E.: Environmental Bisimulations for Higher-Order Languages. ACM Transactions on Programming Languages and Systems 33(1):5 (2011)

[20] Sørensen, M.H.: Turchin's Supercompiler Revisited. Master's thesis, Department of Computer Science, University of Copenhagen (1994), DIKU-Rapport 94/17

[21] Sørensen, M.H., Glück, R., Jones, N.D.: A Positive Supercompiler. Journal of Functional Programming 6(6), 811–838 (1996)

[22] Turchin, V.F.: The Concept of a Supercompiler. ACM Transactions on Programming Languages and Systems 8(3), 90–121 (1986)

[23] Turchin, V.F.: Program Transformation With Metasystem Transitions. Journal of Functional Programming 3(3), 283–313 (1993)

[24] Wadler, P.L.: The Concatenate Vanishes (Dec 1987), FP Electronic Mailing List

[25] Wadler, P.L.: Deforestation: Transforming Programs to Eliminate Trees. Lecture Notes in Computer Science 300, 344–358 (1988)

[26] Zhu, H.: How Powerful are Folding/Unfolding Transformations? Journal of Functional Programming 4(1), 89–112 (1994)