A Comparison of Program Transformation Systems

Michael Dever and G.W. Hamilton

Dublin City University
{mdever, hamilton}@computing.dcu.ie

Abstract. Program transformation is a popular technique for attempting to improve the efficiency of a program. At a high level, program transformation is the process of taking an input program and transforming it into an improved version of the original, bearing the same constraints as the original, e.g. termination constraints. In this paper, we focus on three fold/unfold [3] based transformation systems, *positive supercompilation* [26,25,21,2,12] and *distillation* [8,9,10,11] and *HOSC* [19,18]. We focus on derivatives of both that use labelled transition systems [12,11] and we present these systems, their underlying theory, and implementations. Based upon these implementations we will present an analysis of how they compare to each other, and another transformation system, HOSC[19], when applied to a sample of real-world programs.

1 Introduction

Program transformation describes the process of taking an input program and transforming it via various methodologies, discussed below, to a semantically equivalent program [23] that is bounded by the same constraints as the original program. The goal of such a transformation is to enhance and improve the original program, whether the improvements be scalability, efficiency, concurrency or other measures of improvement. This goal exists as programming, in any language, can be an arduous task, with many aspects that have to be taken in to consideration by the developer.

As skill, knowledge and experience can vary greatly from programmer to programmer, program transformation has the potential to prove an immense aid to developers. If a developer can write a version of software, and have it improved for them by another program (the transformer), this will result in better software coming from that developer. There are, however, downsides to applications of program transformation in the field. Program transformers don't always produce intuitive, comprehendible output; if they did, and the results were intuitive, there would not be a need for the transformer.

There exist many different techniques for the application of program transformation techniques to functional programs; Burstall & Darlington's *fold/unfold* algorithms [3], which are the cornerstones of the other transformations detailed here; Pettorossi and Proietti's *transformation rules (fold/unfold based)*

[23], which further the techniques of Burstall & Darlington; Wadler's *deforestation* [31] and its derivatives [5,7], Turchin's *supercompilation* [28], and its derivatives in the form of *positive supercompilation* [26,25,21,2], *two-level supercompilation*[19,18], Hamilton's *distillation* [8,9,10,11] and many others.

While we focus on program transformation applied to functional languages in this paper, it is worth noting that program transformation is applicable to other types of language, such as logic languages [20], and sequential languages [17]. There are a number of reasons why we focus on functional languages, but the most important reasons are that functional languages are easier to analyze, reason about, and to manipulate using program transformation techniques. The lack of side-effects in pure functional languages is a major benefit, as these do not have to be taken into consideration during the transformation process.

A key note to be made about functional languages is that due to their nature, a lot of functions use intermediate data structures to generate results. As an example, the naive definition of *reverse* below relies heavily upon using intermediate lists, in its call to *append*. Another key feature of some functional languages is the ability to use lazy evaluation, where results are evaluated as they are needed. Even within this context, the use of intermediate structures can be a hindrance, as each allocation requires space and time for allocation etc. [31], and the transformations examined here are aimed at reducing/eliminating usage of intermediate data. To show these reductions, we will present our implementations of the transformation systems, and an analysis of how they compare to each other.

$$
\begin{array}{lll}
reverse = \lambda xs.\textbf{case} & xs \textbf{ of} & \\
 & [] & \rightarrow [] \\
 & (x : xs) & \rightarrow append\ (reverse\ xs)\ [x]
\end{array}
$$

$$
\begin{array}{lll}
append = \lambda xs\ ys.\textbf{case} & xs \textbf{ of} & \\
 & [] & \rightarrow ys \\
 & (x : xs) & \rightarrow (x : append\ xs\ ys)
\end{array}
$$

In this paper, we focus on fold/unfold based transformations using *labelled transition systems* [12,11], that depict a programs run-time behavior, and use *weak bisimulation* to prove correctness. The remainder of this paper is structured as follows: in Section 2 we define the higher-order language to be used, and some constraints on it. In Section 3, we define a labelled transition system, and weak bisimilarity. In Section 4 we define both supercompilation and distillation using a labelled transition system, and in Section 5 we present the results of using these to optimize a sample of programs.

2 Language

The simple higher-order language to be used throughout this paper is shown below:

Within this language, a program consists of an expression to be evaluated, e_0, and a set of function definitions, $\Delta = f_1 = e_1 \ldots f_k = e_k$. Constructors must be

$prog$	$::= e_0$ **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
	$\mid \lambda x.e$	Lambda Abstraction
	$\mid e_0\ e_1$	Application
	$\mid$ **case** e_0 *of* $p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k$	Case Expression
p	$::= c\ x_1 \ldots x_k$	Pattern

Fig. 1. Language Definition

of a fixed arity, and within $c\ e_1 \ldots e_k$, k must be equal to constructor c's arity. *Bound* variables are those introduced by λ-abstraction or **case** patterns, and all other variables are *free*. Two expressions, e_1 and e_2, are equivalent, $e_1 \equiv e_2$, if the only difference between the two is the naming of their bound variables. Case expressions may only have non-nested patterns, and if nested patterns are present, they must be transformed into equivalent non-nested versions [1,30].

The language shown uses a standard call-by-name operational semantics, in which there exists an evaluation relation $\Downarrow$ between closed expressions and *values* (expressions in weak head normal form [14]). The one-step reduction relation, $\overset{r}{\rightsquigarrow}$, is shown in Figure 2, and defines three reductions, f, c and β, where f represents an unfolding of function f, c represents a constructor elimination of constructor c and β represents β-substitution.

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

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

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

Fig. 2. One-Step Reduction Relation

$e \overset{r}{\rightsquigarrow}$ denotes the reduction of an expression, e, by rule r, $e \Uparrow$ denotes e diverging, and $e \Downarrow$ denotes e converging. $e \Downarrow v$ can be used to denote e evaluating to the value v. These notations are defined in below, where $\overset{r}{\rightsquigarrow}^*$ denotes the reflexive transitive closure of $\overset{r}{\rightsquigarrow}$.

$e \overset{r}{\rightsquigarrow}$, iff $\exists e'.e \overset{r}{\rightsquigarrow} e'$ $\quad$ $e \Downarrow$, iff $\exists v.e \Downarrow v$

$e \Downarrow v$, iff $e \overset{r}{\rightsquigarrow}^* v \wedge \neg(v \overset{r}{\rightsquigarrow})$ $\quad$ $e \Uparrow$, iff $\forall e'.e \overset{r}{\rightsquigarrow}^* e' \Rightarrow e' \overset{r}{\rightsquigarrow}$

Definition (Substitution) If e is an expression, and there exists a substitution, $\theta = \{x_1 \to e_1, \ldots x_n \to e_n\}$, then $e\theta = e\{x_1 \to e_1, \ldots x_n \to e_n\}$ denotes the simultaneous substitution of e_i for the variable x_i in e, while ensuring name capture cannot happen. □

Definition (Renaming) If there exists a bijective mapping, σ, such that $\sigma = \{x_1 \to x'_1, \ldots, x_n \to x'_n\}$, and there exists an expression e, then $e\{x_1 \to x'_1, \ldots, x_n \to x'_n\}$ denotes the simultaneous substitution of the variable x_i for x'_i in e. □

Definition (Context) A context, $\mathcal{C}$, is an expression that contains a hole $[]$, where one sub-expression should be, and $\mathcal{C}[e]$ denotes the replacing the hole in $\mathcal{C}$ with the sub-expression e. □

Definition (Evaluation Context, Redex and Observable) Evaluation contexts, $\mathcal{E}$, redexes, $\mathcal{R}$, and observables, $\mathcal{O}$ are as defined below.

$$
\begin{array}{rl}
\mathcal{E} ::= & [] \\
| & \mathcal{E}\ e \\
| & \textbf{case}\ \mathcal{E}\ \textbf{of}\ p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k
\end{array}
$$

$$
\begin{array}{rl}
\mathcal{R} ::= & f \\
| & (\lambda x.e_0)\ e_1 \\
| & \textbf{case}\ (x\ e_1 \ldots e_n)\ \textbf{of}\ p_1 \Rightarrow e'_1 \mid \ldots \mid p_k \Rightarrow e'_k \\
| & \textbf{case}\ (c\ e_1 \ldots e_n)\ \textbf{of}\ p_1 \Rightarrow e'_1 \mid \ldots \mid p_k \Rightarrow e'_k
\end{array}
$$

$$
\begin{array}{rl}
\mathcal{O} ::= & x\ e_1 \ldots e_n \\
| & c\ e_1 \ldots e_n \\
| & \lambda x.e
\end{array}
$$

□

Definition (Observational Equivalence) Observational equivalence, $\simeq$, equates two expressions if and only if they exhibit the same termination behavior in all closing contexts, i.e. $e_1 \simeq e_2$ iff $\forall \mathcal{C}.\mathcal{C}[e_1] \Downarrow$ iff $\mathcal{C}[e_2] \Downarrow$. □

3 Labelled Transition Systems

As per Gordon [6], Hamilton and Jones [12,11], define and extend a labelled transition system, that depicts immediate observations that can be made on expressions to determine their equivalence. Their extension is to allow free variables in both expressions and actions, and a knock on effect of this is that observational equivalence will now require that both free and bound variables in actions match.

Definition (Labelled Transition System) A *driven* LTS associated with the program e_0 is represented by $t = (\mathcal{E}, e_0, \rightarrow, Act)$ where:

- $\mathcal{E}$ represents the set of states of the LTS. Each state can be either an expression or the *end-of-action* state **0**.
- t contains as root the expression e_0, denoted $root(t)$.
- $\rightarrow \subseteq \mathcal{E}\ X\ Act\ X\ \mathcal{E}$ is a *transition relation* relating pairs of states by actions according to the driving rules.
- If $e \in \mathcal{E}$ and $e \xrightarrow{\alpha} e'$ then $e' \in \mathcal{E}$.
- *Act* is a set of actions, α, each of which can either be *silent* or *non-silent*. Non-silent actions are one of: a variable, a constructor, the i^{th} argument of an application, a λ-abstraction over a variable x, **case** selector, a case branch pattern or a **let** abstraction. Silent actions are one of: τ_f, the unfolding of a function f, τ_c, the elimination of a constructor c or τ_β, β-substitution.

□

λ-abstractions, **case** pattern variables and **let** variables that are within the actions of an LTS, t, are bound, denoted by $bv(t)$, while all other variables are free, denoted by $fv(t)$. **let** transitions do not appear in a driven LTS, and are only introduced later on due to generalization, via **let** transitions. The authors note that the LTS notation allows for identifying program behavior just by looking at the labels on the transitions, and that transitions from constructors or variables lead to the end-of-action state, **0**. In addition to the above notation, Hamilton et. al. provide some additional notation for working with an LTS:

- $e \xrightarrow{\alpha} e'$ represents $(e, \alpha, e') \in \rightarrow$.
- $e \rightarrow (\alpha_1, t_1), ..., (\alpha_n, t_n)$ represents an LTS containing root state e where $t_1, ..., t_n$ are LTSs obtained by following transitions labelled $\alpha_1, ..., \alpha_n$ from e.
- $e \Rightarrow e'$ can be used if and only if there exists a potentially empty set of silent transitions from e to e'.
- In the case of a non-silent action α, $e_1 \stackrel{\alpha}{\Rightarrow} e_2$ can be used if and only if there exists e_1 and e_2 such that $e_1 \Rightarrow e_1' \xrightarrow{\alpha} e_2' \Rightarrow e_2$

Comparisons of program behavior can be completed using by using weak bisimilarity, defined below:

Definition (Weak Simulation) The binary relation $\mathcal{R} \subseteq \mathcal{E}_1 X \mathcal{E}_2$ is a weak simulation of a pure LTS $(\mathcal{E}_1, e_0^1, \rightarrow_1, Act_1)$ by another 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 $(e_1, e_2) \in \mathcal{R}, \alpha \in Act_1, e_1' \in \mathcal{E}_1$ it holds that if $e_1 \stackrel{\alpha}{\Rightarrow} e_1'$ then
$\exists e_2' \in \mathcal{E}_2 . e_2 \stackrel{\alpha}{\Rightarrow} e_2' \wedge (e_1', e_2') \in \mathcal{R}$ □

Definition (Weak Bisimulation) A *weak bisimulation* is a binary relation $\mathcal{R}$ such that itself and its inverse are weak simulations. □

Definition (Weak Bisimilarity) If a weak bisimulation $\mathcal{R}$ exists between two pure LTSs, then there exists a unique maximal one, denoted $\sim$. □

4 Transformation Techniques using Labelled Transformation Systems

Supercompilation, [12], and distillation [11], are both program transformation techniques aimed at reducing the use of intermediate data during the evaluation of programs. The goal of the present paper is to compare distillation with positive supercompilation. For this purpose the paper presents a formulation of positive supercompilation in LTS terms, shown in Figure 4, and compares it with an LTS formulation of distillation, shown in Figure 5. At the core of both of these techniques, is a process known as driving, which is essentially a forced unfolding, applied in a top-down fashion, to construct potentially infinite trees of states and transitions. Within driving, all function applications are removed, and will only be re-introduced due to generalization, $\mathcal{G}$, via **let** transitions at a later point. These driving rules, $\mathcal{D}$, are the transformation rules that define the technique, and are applicable to all terms that satisfy the language definition above.

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

$$\mathcal{D}'[\![e = x\ e_1 \ldots e_n]\!]\ \theta = \begin{cases} e \rightarrow (\tau_{\downarrow\theta(x)}, \mathcal{D}'[\![\theta(x)\ e_1 \ldots e_n]\!]\ \theta), & \text{if } x \in dom(\theta) \\ e \rightarrow (x, \mathbf{0}), (\#1, \mathcal{D}'[\![e_1]\!]\ \theta), \ldots, (\#n, \mathcal{D}'[\![e_n]\!]\ \theta), & \text{otherwise} \end{cases}$$

$$\mathcal{D}'[\![e = c\ e_1 \ldots e_n]\!]\ \theta = e \rightarrow (c, \mathbf{0}), (\#1, \mathcal{D}'[\![e_1]\!]\ \theta), \ldots, (\#n, \mathcal{D}'[\![e_n]\!]\ \theta)$$

$$\mathcal{D}'[\![e = \lambda x.e]\!]\ \theta = e \rightarrow (\lambda x, \mathcal{D}'[\![e]\!]\ \theta)$$

$$\mathcal{D}'[\![e = E[f]]\!]\ \theta = e \rightarrow (\tau_f, \mathcal{D}'[\![E[e]]\!]\ \theta) \quad \text{where } (f = e) \in \Delta$$

$$\mathcal{D}'[\![e = E[(\lambda x.e_0)\ e_1]]\!]\ \theta = \mathcal{D}'[\![E[e_0]]\!]\ (\theta \cup \{x \mapsto e_1\})$$

$$\mathcal{D}'[\![e = E[\mathbf{case}\ x\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k]]\!]\ \theta = \begin{cases} e \rightarrow (\tau_\beta, \mathcal{D}'[\![E[\mathbf{case}\ \theta(x)\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k]]\!]\ \theta), & \text{if } x \in dom(\theta) \\ e \rightarrow (\mathbf{case}, \mathcal{D}'[\![x]\!]\ \theta), (p_1, \mathcal{D}'[\![E[e_1]]\!]\ (\theta \cup \{x \mapsto p_1\})), & \\ \qquad \ldots, & \\ \qquad (p_k, \mathcal{D}'[\![E[e_k]]\!]\ (\theta \cup \{x \mapsto p_k\})), & \text{otherwise} \end{cases}$$

$$\mathcal{D}'[\![e = E[\mathbf{case}\ (x\ e_1 \ldots e_n)\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k]]\!]\ \theta = e \rightarrow (\mathbf{case}, \mathcal{D}'[\![x\ e_1 \ldots e_n]\!]\ \theta), (p_1, \mathcal{D}'[\![E[e'_1]]\!]\ \theta), \ldots, (p_k, \mathcal{D}'[\![E[e'_k]]\!]\ \theta)$$

$$\mathcal{D}'[\![e = E[\mathbf{case}\ (c\ e_1 \ldots e_n)\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k]]\!]\ \theta = e \rightarrow (\tau_c, \mathcal{D}'[\![E[e'_i]]\!]\ (\theta \cup \{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\})) \quad \text{where } p_i = c\ x_1 \ldots x_n$$

Fig. 3. Driving Rules

Within both of these transformation systems, driving is performed on an input expression, performing a normal order reduction, in which wherever possible, silent transitions are generated. Whenever an expression without a reduction is encountered, a non-silent transition is generated. If **case** expressions cannot be evaluated, then LTS transitions are generated for their branches with infor-

mation propagated according to each branches pattern. As driving results in a *potentially infinite* labelled transition system, obviously the transformation process cannot stop here, as its results so far may be infinite, and as such, an important issue in both systems is that of termination. Each system approaches this in a similar, but importantly, different manner.

Both make use of both *folding* and *generalization* to guide termination. Generalization is performed when the risk of there being a potentially infinite unfolding has been detected. The distinction between the approach of the two systems is that supercompilation performs both of these on previously encountered *expressions*, while distillation performs these on previously encountered *labelled transition systems*. This difference is quite significant as an LTS obviously contains a lot more information than just a sole expression.

In supercompilation, folding, $\mathcal{F}_s$, is performed upon encountering a renaming of a previously encountered expression, and generalization, $\mathcal{G}_s$, is performed upon encountering an embedding of a previously encountered expression. In distillation, folding, $\mathcal{F}_d$, is performed upon encountering a renaming of a previously encountered LTS, and generalization, $\mathcal{G}_d$, is performed upon encountering an embedding of a previously encountered LTS. In both cases, an embedding is defined by a homeomorphic embedding relation.

In both systems, given an input expression, e, the driving rules above, $\mathcal{D}$ are applied resulting in an LTS, $\mathcal{D}[\![e]\!]$. Next, transformation rules, $\mathcal{T}$, for which distillation is shown in Figure 5 and supercompilation in Figure 4, are applied resulting in an LTS, $\mathcal{T}[\![\mathcal{D}[\![e]\!]]\!]$. Finally, once termination has been guaranteed via generalization, residualization rules $\mathcal{R}$, shown in Figure 6 are applied, resulting in a residualized program, $\mathcal{R}[\![\mathcal{T}[\![\mathcal{D}[\![e]\!]]\!]]\!]$.

$$
\begin{array}{ll}
\mathcal{T}_s[\![e]\!] & = \mathcal{T}'_s[\![e]\!]\ \emptyset \\
\mathcal{T}'_s[\![e \to (\tau_f, e')]\!]\ \rho & = \begin{cases} \mathcal{F}_s[\![e'']\!]\ \rho, & \text{if } \exists e'' \in \rho . e'' \equiv e \\ \mathcal{T}'_s[\![\mathcal{G}_s[\![e'']\!]\ [\![e]\!]\ \sigma]\!]\ \rho\ , & \text{if } \exists e'' \in \rho . e'' \bowtie_s e \\ e \to (\tau_f, \mathcal{T}'_s[\![e']\!]\ (\rho \cup \{e\})\ ), & \text{otherwise} \end{cases} \\
\mathcal{T}'_s[\![e \to (\tau_\beta, e')]\!]\ \rho & = \begin{cases} \mathcal{F}_s[\![e'']\!]\ \rho, & \text{if } \exists e'' \in \rho . e'' \equiv e \\ \mathcal{T}'_s[\![\mathcal{G}_s[\![e'']\!]\ [\![e]\!]\ \sigma]\!]\ \rho\ , & \text{if } \exists e'' \in \rho . e'' \bowtie_s e \\ e \to (\tau_\beta, \mathcal{T}'_s[\![e']\!]\ (\rho \cup \{e\})\ ), & \text{otherwise} \end{cases} \\
\mathcal{T}'_s[\![e\ \to (\alpha_1, e_1), \ldots, (\alpha_n, e_n)]\!]\ \rho & = e\ \to (\alpha_1, \mathcal{T}'_s[\![e_1]\!]\ \rho\ ), \ldots, (\alpha_n, \mathcal{T}'_s[\![e_n]\!]\ \rho\ ) \\
\mathcal{T}'_s[\![e \to (\mathbf{let}, e_0), (x, e_1)]\!]\ \rho & = e \to (\mathbf{let}, \mathcal{T}'_s[\![e_0]\!]\ \rho\ ), (x, \mathcal{T}'_s[\![e_1]\!]\ \rho\ )
\end{array}
$$

Fig. 4. Transformation Rules for Supercompilation

The supercompilation transformation system takes the results of $\mathcal{D}[\![e]\!]$ which is a labelled transition system. As $\mathcal{D}[\![e]\!]$ can be infinite, the transformation rules $\mathcal{T}_s$ only traverse a finite portion lazily from the root. Generalization rules $\mathcal{G}_s$ are applied if the danger of an infinite unfolding (due to recursive function calls) is detected, and a "whistle is blown". When a whistle is blown it indicates the detection of a homeomorphic embedding of a previously encountered *expression*,

and application of these rules results in an LTS with no danger of infinite folding, and a finite set of expressions on any path from it's root. Once the system has been generalized, folding rules, $\mathcal{F}_s$ are applied. Folding takes a generalized LTS and produces a bisimilar LTS, one with a finite number of states that can be residualized into an expression using the residualization rules $\mathcal{R}$.

$$\mathcal{T}_d[\![e]\!] = \mathcal{T}'_d[\![e]\!]\ \emptyset\ \emptyset$$

$$\mathcal{T}'_d[\![t = e \to (\tau_f, t')]\!]\ \rho\ \theta = \begin{cases} \mathcal{F}_s[\![t'']\!]\ \sigma, & \text{if } \exists t'' \in \rho, \sigma.t'' \approx t\sigma \\ \mathcal{T}'_d[\![\mathcal{G}_d[\![t'']\!]\ [\![t]\!]\ \theta\ \sigma]\!]\ \rho\ \phi, & \text{if } \exists t'' \in \rho, \sigma.t'' \bowtie_d t\sigma \\ e \to (\tau_f, \mathcal{T}'_d[\![t']\!]\ (\rho \cup \{t\})\ \theta), & \text{otherwise} \end{cases}$$

$$\mathcal{T}'_d[\![t = e \to (\tau_\beta, t')]\!]\ \rho\ \theta = \begin{cases} \mathcal{F}_s[\![t'']\!]\ \sigma, & \text{if } \exists t'' \in \rho, \sigma.t'' \approx t\sigma \\ \mathcal{T}'_d[\![\mathcal{G}_d[\![t'']\!]\ [\![t]\!]\ \theta\ \sigma]\!]\ \rho\ \phi, & \text{if } \exists t'' \in \rho, \sigma.t'' \bowtie_d t\sigma \\ e \to (\tau_\beta, \mathcal{T}'_d[\![t']\!]\ (\rho \cup \{t\})\ \theta), & \text{otherwise} \end{cases}$$

$$\mathcal{T}'_d[\![e\ \to (\alpha_1, t_1), \ldots, (\alpha_n, t_n)]\!]\ \rho\ \theta = e\ \to (\alpha_1, \mathcal{T}'_d[\![t_1]\!]\ \rho\ \theta), \ldots, (\alpha_n, \mathcal{T}'_d[\![t_n]\!]\ \rho\ \theta)$$

$$\mathcal{T}'_d[\![t = e \to (\mathbf{let}, t_0), (x, t_1)]\!]\ \rho\ \theta$$
$$= \begin{cases} \mathcal{T}'_d[\![t_0\{x \mapsto x'\}]\!]\ \rho\ \theta, & \text{if } \exists (x' \mapsto t_2) \in \theta.t_1 \approx t_2 \\ e \to (\mathbf{let}, \mathcal{T}'_d[\![t_0]\!]\ \rho\ (\theta \cup \{x \mapsto t_1\})), & \\ \qquad (x, \mathcal{T}'_d[\![t_1]\!]\ \rho\ \theta), & \text{otherwise} \end{cases}$$

Fig. 5. Transformation Rules for Distillation

The distillation transformation system proceeds in a similar manner, taking the results of $\mathcal{D}[\![e]\!]$ and applying $\mathcal{T}_d$ to this LTS resulting in another LTS with a finite set of states. In distillation however the whistle is blown when an embedding of a previously encountered *LTS* is detected and generalization, $\mathcal{G}_d$, then ensures that a renaming of a previously encountered LTS will be found. Folding, $\mathcal{F}_d$ is then applied to the LTS resulting in one with a finite set of states and then this can be residualized into a program using the residualization rules $\mathcal{R}$. The main difference between the two systems is that in supercompilation the comparisons are performed on *expressions*, and in distillation they are performed on *LTS*'s.

The questions related to correctness and termination, as well as the details of folding and generalization, are out of the scope of the present paper, because they have been earlier addressed in [12,11].

As mentioned previously, both transformation systems aim to remove intermediate data from their inputs, resulting in a more efficient output. This is done through removing silent transitions, defined previously. Briefly, these are either function unfoldings, β-reductions or constructor eliminations, and these are strongly linked to how powerful each system is. In the case of supercompilation, as it looks at expressions, there can only be a constant number of silent transitions between recursive calls of a function, the removal of which leads to a potentially linear increase in efficiency [27]. Distillation however, allows for an increasing number of silent transitions between recursive calls of a function, allowing for a potentially super-linear increase in efficiency [11]. This is more complex than supercompilation, as the LTS's used for comparison can be infi-

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

$$\begin{array}{ll}
\mathcal{R}'[\![e \to (x, \mathbf{0}), (\#1, t_1), \ldots, (\#n, t_n)]\!]\ \varepsilon & = x\ (\mathcal{R}'[\![t_1]\!]\ \varepsilon) \ldots (\mathcal{R}'[\![t_n]\!]\ \varepsilon) \\
\mathcal{R}'[\![e \to (c, \mathbf{0}), (\#1, t_1), \ldots, (\#n, t_n)]\!]\ \varepsilon & = c\ (\mathcal{R}'[\![t_1]\!]\ \varepsilon) \ldots (\mathcal{R}'[\![t_n]\!]\ \varepsilon) \\
\mathcal{R}'[\![e \to (\lambda x, t)]\!]\ \varepsilon & = \lambda x.(\mathcal{R}'[\![t]\!]\ \varepsilon) \\
\mathcal{R}'[\![e \to (\mathbf{case}, t_0)(p_1, t_1), \ldots, (p_n, t_k)]\!]\ \varepsilon & = \mathbf{case}\ (\mathcal{R}'[\![t_0]\!]\varepsilon)\ \mathbf{of}\ p_1 \Rightarrow (\mathcal{R}'[\![t_1]\!]\varepsilon) \\
 & \qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
 & \qquad\qquad\qquad\qquad\qquad\qquad p_k \Rightarrow (\mathcal{R}'[\![t_k]\!]\varepsilon) \\
\mathcal{R}'[\![e \to (\mathbf{let}, t_0), (x, t_1)]\!]\ \varepsilon & = (\mathcal{R}'[\![t_0]\!]\ \varepsilon)\{x \mapsto (\mathcal{R}'[\![t_1]\!]\ \varepsilon)\} \\
\mathcal{R}'[\![e \to (\tau_c, t)]\!]\ \varepsilon & = \mathcal{R}'[\![t]\!]\ \varepsilon
\end{array}$$

$$\mathcal{R}'[\![e \to (\tau_f, t)]\!]\ \varepsilon = \begin{cases} f'\ x_1 \ldots x_n, \text{ if } \exists(f'\ x_1 \ldots x_n = e) \in \varepsilon \\ f'\ x_1 \ldots x_n\ \mathbf{where}\ f' = \lambda x_1 \ldots x_n.(\mathcal{R}'[\![t]\!]\ (\varepsilon \cup \{f'\ x_1 \ldots x_n = e\})), \\ \qquad\qquad \text{otherwise } (f' \text{ is fresh}, \{x_1 \ldots x_n\} = fv(t)) \end{cases}$$

$$\mathcal{R}'[\![e \to (\tau_\beta, t)]\!]\ \varepsilon = \begin{cases} f\ x_1 \ldots x_n, \text{ if } \exists(f\ x_1 \ldots x_n = e) \in \varepsilon \\ f\ x_1 \ldots x_n\ \mathbf{where}\ f = \lambda x_1 \ldots x_n.(\mathcal{R}'[\![t]\!]\ (\varepsilon \cup \{f\ x_1 \ldots x_n = e\})), \\ \qquad\qquad \text{otherwise } (f \text{ is fresh}, \{x_1 \ldots x_n\} = fv(t)) \end{cases}$$

Fig. 6. Rules For Residualization

nite, whereas the expression compared in supercompilation is not, however distillation benefits from the fact that as any infinite sequence of transitions must contain either a function unfolding or substitution, once one of these is detected, no further comparison needs to be done.

5 Two-Level Supercompilation

There are many other approaches to our supercompilation technique, such as *supero* [22,21]. However, we expect such supercompilation systems to be similar in nature and success as our system. Another more powerful approach to the removal of intermediate data is that of *two-level supercompilation* [19,18]. The authors of this approach supercompilation as a multi-level program transformation technique, using a lower and an upper level supercompiler to transform an input program. Like distillation, this technique is capable of obtaining a super-linear increase in efficiency, but was originally intended as an analysis tool.

Using a multi-level approach is based upon the concept of *meta-system transition* presented by Turchin in [29]. Briefly, a meta-system S' is a system composed of copies and/or variations of another system S and means of controlling these copies of S. If there exists another meta-system S'' composed of copies of S', then the hierarchy of control in this system is obvious.

In two-level supercompilation, S' is a hierarchical meta-system composed of copies of a 'classic' supercompiler S, that each can control each other. Lower level supercompilers generate improvement lemmas [24] - improved expressions equivalent to an input expression - guided by the expressions labeling the nodes of its partial process tree [19]. These improvement lemmas can then be used to guide the upper level supercompiler in its search for improvement lemmas.

The supercompilers at the different levels of a two-level supercompiler can differ in behavior, allowing for an upper and lower level supercompiler to produce different results. Using its hierarchy of possibly differing supercompilers, it, like distillation is capable of obtaining a super-linear increase in efficiency.

6 Comparison of Techniques

Based upon the fact that distillation is significantly more powerful than supercompilation, we have implemented both techniques for comparison using the popular functional language Haskell [15]. As two-level supercompilation is also more powerful than supercompilation and can obtain results similar to that of distillation, we would have liked to included it in our comparison, however we had difficulties getting the tool functioning, so we have included the output of the HOSC [19] single level supercompiler. To perform these comparisons, we represent parsed Haskell programs into a simple representation of the language similar to that shown in Figure 1. It is worth noting that our transformation tool is still a work in progress, and as such our comparison is on a set of relatively 'simpler' programs, that match up pretty closely to the language definition, with more advanced features of the Haskell language being disallowed.

The benchmarks we used are as follows: *sumsquare* [4], a function that calculates the sum of the squares of two lists; *wordcount*, *linecount* and *charcount* [22] are functions that respectively count the number of words, lines and characters in a file; *factorial*, *treeflip* and *raytracer* are programs used to benchmark previous supercompilation work [16]; *nrev* represents a naive list reversal program; and a sample of programs from the well known Haskell nofib benchmark suite[13]. We present the results obtained after supercompiling, distilling and applying HOSC to our set of benchmarks, with focus on both execution time and memory usage. The results of the transformation systems are shown as a percentage of the measure used for the unoptimized version, and a lower percentage will obviously indicate an improvement.

6.1 Findings

In Figure 7, we present our findings of the execution time performance of our set of benchmark programs based on seconds taken to complete execution. From left to right, the columns describe the name of the sample program, its unoptimized execution time (in seconds), the time taken by the supercompiled version, the time taken by the output of HOSC, and finally, the time taken by the distilled version.

In Figure 8 we present our findings of the memory performance of our set of benchmark programs based on maximum memory allocated on the heap during execution time. From left to right, the columns describe the name of the sample program, its unoptimized total memory usage (in MB), the memory used by the supercompiled version, the memory used by the output of HOSC, and finally, the memory used by the distilled version.

Name	Unoptimized	Supercompilation	HOSC	Distillation
nrev	62.5	53.3	68.7	0.1
charcount	0.01	0.01	0.01	0.01
exp3_8	45.9	32.4	52.1	-
factorial	2.6	2.5	2.8	-
linecount	28.7	0.01	0.01	0.01
primes	79.2	75.9	104.5	-
raytracer	12.7	10.0	10.4	10.0
rfib	57.7	35.3	37.7	-
sumsquare	81.9	72.7	76.9	-
treeflip	51.2	29.9	32.2	-
wordcount	29.8	0.01	0.01	0.01

Fig. 7. Execution Time

Name	Unoptimized	Supercompilation	HOSC	Distillation
nrev	8	6	11	3
charcount	3	3	3	3
exp3_8	6	4	6	-
factorial	3	3	3	-
linecount	6	1	1	1
primes	2	2	2	-
raytracer	1011	730	732	732
rfib	2073	1061	1047	-
sumsquare	2313	2391	2221	-
treeflip	2176	1083	1069	-
wordcount	6	1	1	1

Fig. 8. Memory Usage

There are some interesting conclusions that can be drawn from both Figure 7 and Figure 8. A quick review of our execution time and memory usage findings show that, while the distillation tool is incomplete at this point it in no case underperforms either of the other two optimization techniques with respect to execution time. It also reveals that our implementation of supercompilation fares better than that of HOSC in terms of execution time. In the following sections we present the following: firstly a comparison of our implementation of supercompilation and HOSC, and secondly a comparison of the supercompilation techniques and our implementation of distillation. It is worth noting at this point that, as mentioned above, our implementation of distillation is a work in progress, and as a result of this there are some programs that we were unable to optimize.

Supercompilation Techniques With respect to execution time, and the test cases involved, we find that our implementation of supercompilation results in

more outputs with improved efficiency than that of HOSC. We also find that our supercompiler results in a reduced execution time, whereas HOSC does not and in some cases results in a drastic increase in execution time.

This reduction in performance is most pronounced in the case of *primes*, where for the same input, the original unoptimized version has an execution time of 79.2 seconds, our super-compiled version takes 75.9 seconds and HOSC takes 104.5 seconds. Compared with the original our implementation results in a 4.16% reduction in execution time, and HOSC results in a 31.94% increase in execution time over the original and a 37.68% increase in execution time over our implementation of supercompilation. In the cases of *charcount*, *linecount* and *wordcount* both implementations resulted in the same decrease of execution time, with *charcount* seeing no increase in efficiency, which was expected and *linecount* and *wordcount* seeing a 99.96% increase in execution time.

With respect to memory usage, and the test cases involved, we find that our implementation of supercompilation often results in more efficient outputs than that of HOSC. In the cases of *nrev*, *exp*3_8 and *raytracer* our supercompiler gives a greater reduction in memory usage than that of HOSC, which in the case of *nrev* actually has a negative impact on memory usage. The original program has a maximum allocation of 8 MB, our supercompiled version uses 6 MB, a reduction of 25%, and the HOSC supercompiled version uses 11 MB, an increase of 37.5%.

However there are also some cases where HOSC seems to obtain a slight decrease in memory usage when compared to both the original and our supercompiler, i.e. $rfib$, and $treeflip$. In the case of *sumsquare* our supercompiled version results in an output that is less efficient than both the original program and the output of HOSC with respect to memory usage. The original program uses 2313 MB, the output of HOSC uses 2221 MB and our supercompiled version uses 2391 MB. Our supercompiled version represents a 3.37% increase in memory usage, and the HOSC supercompiled version represents 3.98% decrease in memory usage over the original and a 7.65% decrease in memory usage over our supercompiled version.

Distillation vs. Supercompilation As our sample set of distillable programs is limited, we have few benchmarks to draw conclusions from, namely *nrev*, *charcount*, *linecount*, *raytracer* and *wordcount*. With respect to execution time for these test cases, *nrev* is probably the most interesting example with the original program taking 62.5 seconds, our supercompiled version taking 53.3 seconds, the HOSC supercompiled version taking 68.7 seconds and our distilled version taking 0.1 seconds, representing a 14.72% increase, a 9.92% decrease and a 99.84% increase in efficiency respectively. An increase of this order was expected for distillation as it is capable of obtaining a super-linear increase in efficiency.

With respect to memory usage, and the same set of benchmarks, *nrev* and *raytracer* are the most interesting examples. In the case of *nrev*, the original program uses 8 MB, our supercompiled version uses 6 MB, the HOSC super-

compiled version uses 11 MB and our distilled version uses 3 MB, representing a 25% reduction, an 11% increase and a 62.5% reduction in memory usage respectively. Again this is to be expected with supercompilation. In the case of *raytracer*, the original program uses 1011 MB, our supercompiled version uses 730 MB, and both the distilled and the HOSC supercompiled versions use 732 MB, representing a 27.79% decrease and a 27.6% reduction respectively. What is interesting about this is that our supercompiled version is more efficient with respect to memory usage than our distilled version, when the opposite would be expected.

7 Future Work

Work is progressing at present on extension of the distillation tool to allow it to handle some of the more powerful features of the Haskell programming language. Once it is capable of handling these more advanced features it will be applied to the benchmarks that weren't possible for this comparison. The results of these optimizations will be published, alongside those of the multi-level supercompiler mentioned previously for comparison. We aim to also support all programs in the Nofib benchmark suite, and some more real-world programs.

8 Acknowldgements

This work was supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero - the Irish Software Engineering Research Centre

References

1. Augustsson, L.: Compiling pattern matching. Functional Programming Languages and Computer Architecture (1985)
2. Bolingbroke, M., Jones, S.P.: Supercompilation by evalutation. Proceedings of the 2010 ACM SIGPLAN Haskell Symposium (2010)
3. Burstall, R.M., Darlington, J.: A transformation system for developing recursive programs. Journal of the Association for Computing Machinery 24(1), 44–67 (January 1977)
4. Coutts, D., Leshchinskiy, R., Stewart, D.: Stream fusion. from lists to streams to nothing at all. In: ICFP'07 (2007)
5. Gill, A., Launchbury, J., Jones, S.P.: A shortcut to deforestation. FPCA: Proceedings of the conference on Functional programming languages and computer architecture pp. 223–232 (1993)
6. Gordon, A.D.: Bisimilarity as a theory of functional programming. Electronic Notes in Theoretical Computer Science 1, 232 – 252 (1995)
7. Hamilton, G.W.: Higher order deforestation. Fundamenta Informaticae 69(1-2), 39–61 (July 2005)
8. Hamilton, G.W.: Distillation: Extracting the essence of programs. Proceedings of the ACM Workshop on Partial Evaluation and Program Manipulation (2007)

9. Hamilton, G.W.: Extracting the essence of distillation. Proceedings of the Seventh International Andrei Ershov Memorial Conference: Perspectives of System Informatics (2009)
10. Hamilton, G.W., Mendel-Gleason, G.: A graph-based definition of distillation. Proceedings of the Second International Workshop on Metacomputation in Russia (2010)
11. Hamilton, G., Jones, N.: Distillation and labelled transition systems. Proceedings of the ACM Workshop on Partial Evaluation and Program Manipulation pp. 15–24 (January 2012)
12. Hamilton, G., Jones, N.: Proving the correctness of unfold/fold program transformations using bisimulation. Lecture Notes in Computer Science 7162, 153–169 (2012)
13. Haskell-Community: Nofib benchmarking suite (2012), `http://darcs.haskell.org/nofib/`
14. Jones, S.P.: The Implementation of Functional Programming Languages. Prentice-Hall (1987)
15. Jones, S.P.: Haskell 98 language and libraries - the revised report. Tech. rep. (2002)
16. Jonsson, P.A., Nordlander, J.: Positive supercompilation for a higher order call-by-value language. SIGPLAN Not. 44(1), 277–288 (Jan 2009)
17. Klimov, A.V.: An approach to supercompilation for object-oriented languages: the java supercompiler case study (2008)
18. Klyuchnikov, I., Romanenko, S.: Towards higher-level supercompilation. In: Second International Valentin Turchin Memorial Workshop on Metacomputation in Russia. pp. 82–101. Ailamazyan University of Pereslavl (2010)
19. Klyuchnikov, I.G.: Towards effective two-level supercompilation (2010)
20. Lloyd, J.W., Shepherdson, J.C.: Partial evaluation in logic programming. J. Log. Program. 11(3-4), 217–242 (1991)
21. Mitchell, N.: Rethinking supercompilation. In: ICFP '10: Proceedings of the 15th ACM SIGPLAN Internation Conference on Functional Programming. pp. 309–320. ACM (September 2010)
22. Mitchell, N., Runciman, C.: A supercompiler for core Haskell. In: et al., O.C. (ed.) IFL 2007. LNCS, vol. 5083, pp. 147–164. Springer-Verlag (May 2008)
23. Pettorossi, A., Proietti, M.: Rules and strategies for transforming functional and logic programs. ACM Comput. Surv. 28(2), 360–414 (Jun 1996)
24. Sands, D.: Total correctness by local improvement in the transformation of functional programs. ACM Transactions on Programming Languages and Systems 18, 175–234 (1996)
25. Sørensen, M., Glück, R.: An algorithm of generalization in positive supercompilation. International Logic Programming Symposium pp. 465–479 (1995)
26. Sørensen, M., Glück, R., Jones, N.: A positive supercompiler. Journal of Functional Programming 1(1) (January 1993)
27. Sørensen, M.H.: Turchin's supercompiler revisited - an operational theory of positive information propagation (1996)
28. Turchin, V.F.: The concept of a supercompiler. ACM Transactions on Programming Languages and Systems 8(3), 292–325 (June 1986)
29. Turchin, V.F.: Metacomputation: Metasystem transitions plus supercompilation. In: Selected Papers from the Internaltional Seminar on Partial Evaluation. pp. 481–509. Springer-Verlag, London, UK, UK (1996)
30. Wadler, P.: Efficient compilation of pattern matching. In: Jones, S.P. (ed.) The Implementation of Functional Programming Languages., pp. 78–103. Prentice-Hall (1987)

31. Wadler, P.: Deforestation: Transforming programs to eliminate trees. Theoretical Computer Science 73, 231–248 (1990)