A Hierarchy of Program Transformers

G.W. Hamilton

School of Computing
Dublin City University
Dublin 9
Ireland
e-mail: *hamilton@computing.dcu.ie*

Abstract. In this paper, we describe a hierarchy of program transformers in which the transformer at each level of the hierarchy builds on top of the transformers at lower levels. The program transformer at the bottom of the hierarchy corresponds to positive supercompilation, and that at the next level corresponds to the first published definition of distillation [4]. We then show how the more recently published definition of distillation [5] can be described using this hierarchy. We see that this moves up through the levels of the transformation hierarchy until no further improvements can be made. The resulting definition of distillation uses only finite data structures, as opposed to the definition in [5], and we therefore argue that it is easier to understand and to implement.

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* [7], *deforestation* [17] and *supercompilation* [15]. *Positive supercompilation* [14] 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 [13].

The *distillation* algorithm was originally motivated by the need for automatic techniques for obtaining superlinear speedups in programs. The original definition of distillation [4] was very similar in its formulation to positive supercompilation; the main difference being that in positive supercompilation, generalization and folding are performed with respect to *expressions*, while in distillation, they are performed with respect to *graphs*. The graphs which were used in distillation for this purpose were in fact those produced by positive supercompilation, so we can see that this definition of distillation was built on top of positive supercompilation. This suggests the existence of a hierarchy of program transformers, where the transformer at each level is built on top of those at lower levels, and higher

level transformers are more powerful. In this paper, we define such a hierarchy inductively, with positive supercompilation as the base case at the bottom level, and each new level defined in terms of the previous ones. The original definition of distillation is therefore at the second level in this hierarchy, while the more recently published definition of distillation [5] does not actually belong to any single level of the hierarchy, but in fact moves up through the levels until no further improvements can be made. We also define this more recent version of distillation using our program transformer hierarchy.

The remainder of this paper is structured as follows. In Section 2, we define the syntax and semantics of the higher-order functional language on which the described transformations are performed. In Section 3, we define labelled transition systems, which are used to represent the results of transformations. In Section 4, we define the program transformer hierarchy, where the transformer at the bottom level corresponds to positive supercompilation, and each successive transformer is defined in terms of the previous ones. In Section 5, we describe the more recent definition of distillation using the program transformer hierarchy, and show how it moves up through the levels of this hierarchy until no further improvements can be made. Section 6 concludes and considers 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 1 (Language Syntax). The syntax of this language is as shown in Fig. 1.

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

Fig. 1. Language Grammar

A program in the language is an expression which can be a variable, constructor application, λ-abstraction, function call, application, **case**, **let** or **where**. Variables introduced by λ-abstraction, **let** 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.

It is assumed that the input program contains no **let** expressions; these are only introduced during transformation. 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. 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 $\textbf{case}\ (\lambda x.e)\ \textbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k$ cannot occur.

Example 1. An example program in our language which calculates the nth fibonacci number is shown in Fig. 2.

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

Fig. 2. Example Program

Definition 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 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 4 (Shallow Reduction Context). A shallow reduction context $\mathcal{R}$ is an expression containing a single hole $\bullet$ in the place of the redex, which can have one of the two following possible forms:

$$\mathcal{R} ::= \bullet\ e \mid (\textbf{case}\ \bullet\ \textbf{of}\ p_1 \Rightarrow e_1 \mid \ \ldots \ \mid p_k \Rightarrow e_k)$$

Definition 5 (Evaluation Context). An evaluation context $\mathcal{E}$ is represented as a sequence of shallow reduction contexts (known as a *zipper* [6]), representing

the nesting of these contexts from innermost to outermost within which the expression redex is contained. An evaluation context can therefore have one of the two following possible forms:

$$\mathcal{E} ::= \langle\rangle \mid \langle \mathcal{R} : \mathcal{E} \rangle$$

Definition 6 (Insertion into Evaluation Context). The insertion of an expression e into an evaluation context κ, denoted by $\kappa \bullet e$, is defined as follows:

$$\begin{array}{l} \langle\rangle \bullet e \qquad\qquad = e \\ \langle (\bullet\ e') : \kappa \rangle \bullet e = \kappa \bullet (e\ e') \\ \langle (\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k) : \kappa \rangle \bullet e \\ \qquad\qquad\qquad = \kappa \bullet (\mathbf{case}\ e\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k) \end{array}$$

Free variables within the expression e may become bound within $\kappa \bullet e$; if $\kappa \bullet e$ is closed then we call κ a *closing context* for e.

Definition 7 (Unfolding). The unfolding of a function in the redex of expression e with function environment Δ is defined as shown in Fig. 3.

$$\begin{array}{ll} \mathcal{U}[\![e]\!]\ \Delta = \mathcal{U}'[\![e]\!]\ \langle\rangle\ \varnothing\ \Delta & \\ & \\ \mathcal{U}'[\![x]\!]\ \kappa\ \rho\ \Delta & = \begin{cases} \mathcal{U}'[\![\rho(x)]\!]\ \kappa\ \rho\ \Delta, & \text{if } x \in dom(\rho) \\ \kappa \bullet x, & \text{otherwise} \end{cases} \\ \mathcal{U}'[\![c\ e_1 \ldots e_k]\!]\ \kappa\ \rho\ \Delta & = \kappa \bullet (c\ e_1 \ldots e_k) \\ \mathcal{U}'[\![\lambda x.e]\!]\ \kappa\ \rho\ \Delta & = \kappa \bullet (\lambda x.e) \\ \mathcal{U}'[\![f]\!]\ \kappa\ \rho\ \Delta & = \kappa \bullet e \text{ where } (f = e) \in \Delta \\ \mathcal{U}'[\![e_0\ e_1]\!]\ \kappa\ \rho\ \Delta & = \mathcal{U}'[\![e_0]\!]\ \langle (\bullet\ e_1) : \kappa \rangle\ \rho\ \Delta \\ \mathcal{U}'[\![\mathbf{case}\ e_0\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k]\!]\ \kappa\ \rho\ \Delta & \\ & = \mathcal{U}'[\![e_0]\!]\ \langle (\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k) : \kappa \rangle\ \rho\ \Delta \\ \mathcal{U}'[\![\mathbf{let}\ x = e_0\ \mathbf{in}\ e_1]\!]\ \kappa\ \rho\ \Delta & = \mathbf{let}\ x = e_0\ \mathbf{in}\ \mathcal{U}'[\![e_1]\!]\ \kappa\ (\rho \cup \{x \mapsto e_0\})\ \Delta \\ \mathcal{U}'[\![e_0\ \mathbf{where}\ f_1 = e_1 \ldots f_n = e_n]\!]\ \kappa\ \rho\ \Delta & \\ & = \mathcal{U}'[\![e_0]\!]\ \kappa\ \rho\ (\Delta \cup \{f_1 = e_1, \ldots, f_n = e_n\}) \\ & \quad \mathbf{where}\ f_1 = e_1 \ldots f_n = e_n \end{array}$$

Fig. 3. Function Unfolding

Within these rules, the context around the redex is built up within κ, the values of **let** variables are stored in ρ and the set of function definitions are stored in Δ. If the redex is a variable which has a value within ρ, then that value is substituted into the redex position. If the redex can itself be divided into an inner redex and shallow reduction context, then the shallow reduction context is added to the overall context and the inner redex is further unfolded. If the innermost redex is a function then it is replaced by its definition within Δ; otherwise this innermost redex is simply inserted back into its context.

The call-by-name operational semantics of our language is standard: we 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 $\overset{r}{\leadsto}$ inductively as shown in Fig. 4, where the reduction r can be f (unfolding of function f), c (elimination of constructor c) or β (β-substitution).

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

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

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

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

Fig. 4. One-Step Reduction Relation

We use the notation $e \overset{r}{\leadsto}$ 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 $\overset{r}{\leadsto}^*$ denotes the reflexive transitive closure of $\overset{r}{\leadsto}$:

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

We assume that all expressions are typable under system F, and that types are *strictly positive*. This ensures that all infinite sequences of reductions must include the unfolding of a function.

Definition 8 (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 $\forall \kappa \bullet (\kappa \bullet e_1 \Downarrow \text{ iff } \kappa \bullet e_2 \Downarrow)$.

3 Labelled Transition Systems

In this section, we define the labelled transition systems (LTSs) which are used to represent the results of our transformations.

Definition 9 (Labelled Transition System). The LTS associated with program e is given by $t = (\mathcal{E}, e, \rightarrow, Act)$ where:

- $\mathcal{E}$ is the set of *states* of the LTS each of which is either an expression or the end-of-action state $\mathbf{0}$.
- t always contains as root the expression e, denoted by $root(t) = e$.

- $\rightarrow \subseteq \mathcal{E} \times Act \times \mathcal{E}$ is a *transition relation* that relates pairs of states by actions according to Fig. 5. We 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.
- If $e \in \mathcal{E}$ and $(e, \alpha, e') \in \rightarrow$ 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; @, the function in an application; $\#i$, the ith 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 τ_β, β-substitution.

$$
\begin{array}{ll}
\mathcal{L}[\![x]\!]\ \rho\ \Delta & = x \rightarrow (x, \mathbf{0}) \\
\mathcal{L}[\![c\ e_1 \ldots e_n]\!]\ \rho\ \Delta & = (c\ e_1 \ldots e_n) \rightarrow (c, \mathbf{0}), (\#1, \mathcal{L}[\![e_1]\!]\ \rho\ \Delta), \ldots, (\#n, \mathcal{L}[\![e_n]\!]\ \rho\ \Delta) \\
\mathcal{L}[\![\lambda x.e]\!]\ \rho\ \Delta & = (\lambda x.e) \rightarrow (\lambda x, \mathcal{L}[\![e]\!]\ \rho\ \Delta) \\
\mathcal{L}[\![f]\!]\ \rho\ \Delta & = \begin{cases} f \rightarrow (\tau_f, \mathbf{0}), & \text{if } f \in \rho \\ f \rightarrow (\tau_f, \mathcal{L}[\![e]\!]\ (\rho \cup \{f\})\ \Delta), & \text{otherwise where } (f = e) \in \Delta \end{cases} \\
\mathcal{L}[\![e_0\ e_1]\!]\ \rho\ \Delta & = (e_0\ e_1) \rightarrow (@, \mathcal{L}[\![e_0]\!]\ \rho\ \Delta), (\#1, \mathcal{L}[\![e_1]\!]\ \rho\ \Delta) \\
\multicolumn{2}{l}{\mathcal{L}[\![e = (\mathbf{case}\ e_0\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k)]\!]\ \rho\ \Delta} \\
 & = e \rightarrow (\mathbf{case}, \mathcal{L}[\![e_0]\!]\ \rho\ \Delta), (p_1, \mathcal{L}[\![e_1]\!]\ \rho\ \Delta), \ldots, (p_k, \mathcal{L}[\![e_k]\!]\ \rho\ \Delta) \\
\multicolumn{2}{l}{\mathcal{L}[\![e = (\mathbf{let}\ x = e_0\ \mathbf{in}\ e_1)]\!]\ \rho\ \Delta} \\
 & = e \rightarrow (\mathbf{let}, \mathcal{L}[\![e_1]\!]\ \rho\ \Delta), (x, \mathcal{L}[\![e_0]\!]\ \rho\ \Delta) \\
\multicolumn{2}{l}{\mathcal{L}[\![e_0\ \mathbf{where}\ f_1 = e_1 \ldots f_n = e_n]\!]\ \rho\ \Delta} \\
 & = \mathcal{L}[\![e_0]\!]\ \rho\ (\Delta \cup \{f_1 = e_1, \ldots, f_n = e_n\})
\end{array}
$$

Fig. 5. LTS Representation of a Program

Within the rules $\mathcal{L}$ shown in Fig. 5 for converting a program to a corresponding LTS, the parameter ρ is the set of previously encountered function calls and the parameter Δ is the set of function definitions. If a function call is re-encountered, no further transitions are added to the constructed LTS. Thus, the constructed LTS will always be a finite representation of the program.

Example 2. The LTS representation of the program in Fig. 2 is shown in Fig. 6.

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.

Definition 10 (Extraction of Residual Program from LTS). A residual program can be constructed from a LTS using the rules $\mathcal{R}$ as shown in Fig. 7. Within these rules, the parameter ε contains the set of new function calls that have been created, and associates them with the expressions they replaced. On encountering a renaming of a previously replaced expression, it is also replaced by the corresponding renaming of the associated function call.

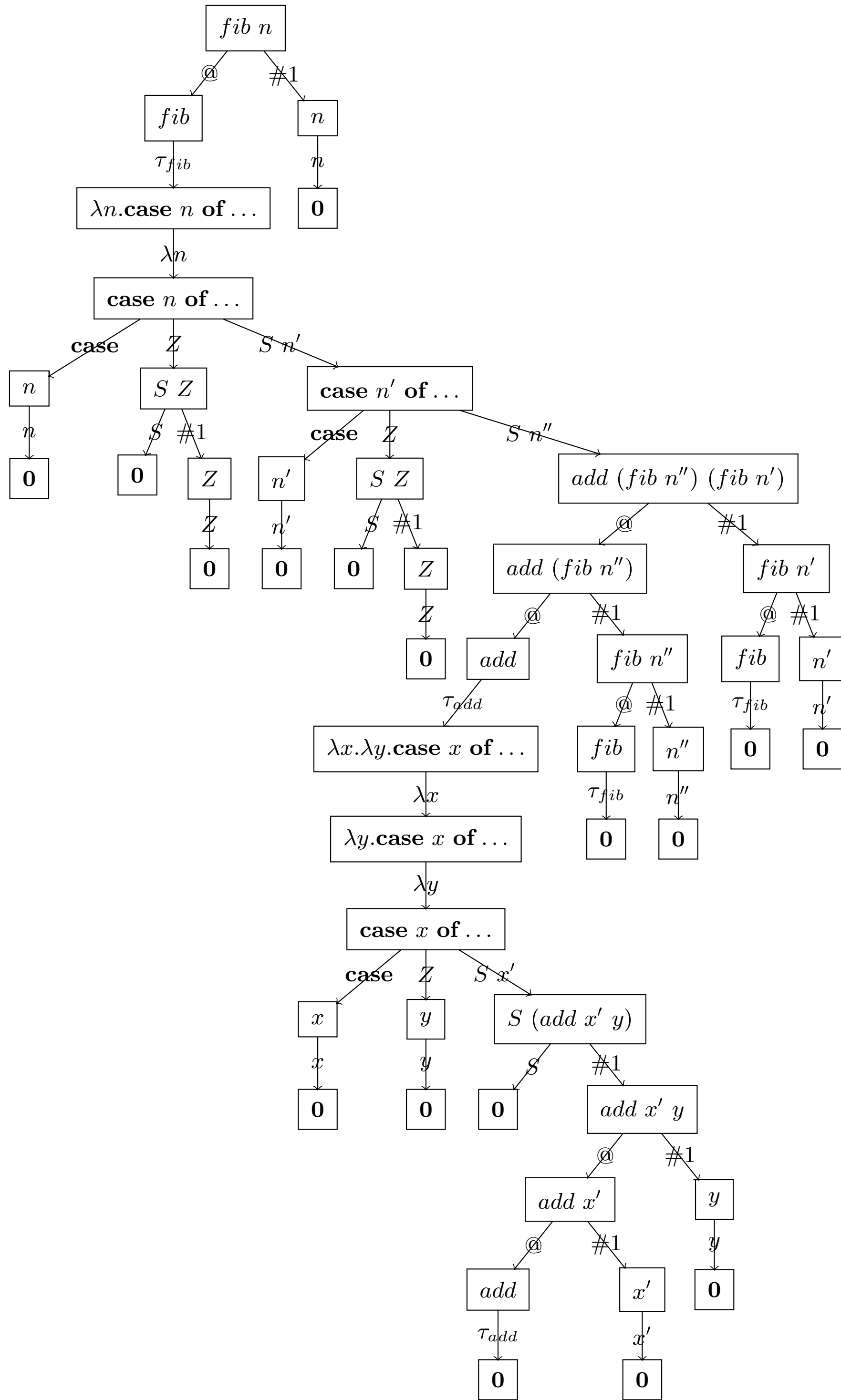

Fig. 6. LTS Corresponding to *fib n*

$$\mathcal{R}[\![t]\!] = \mathcal{R}'[\![t]\!]\ \varnothing$$

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

Fig. 7. Rules For Residualization

Example 3. The residual program constructed from the LTS in Fig. 6 is esentially that shown in Fig. 2 (modulo renaming of functions).

4 A Hierarchy of Program Transformers

In this section, we define a hierarchy of program transformers in which the transformer at each level of the hierarchy makes use of those at lower levels. Each transformer takes as its input the original program and produces as its output a labelled transition system, from which a new (hopefully improved) program can be residualized. In all the transformers, LTSs corresponding to previously encountered terms are compared to the LTS for the current term. If a *renaming* of a previously encountered LTS is detected, then *folding* is performed. If an *embedding* of a previously encountered LTS is detected, then *generalization* is performed. The use of LTSs rather than expressions when checking for renaming or embedding allows us to abstract away from the specific function names which are used within expressions and to focus on their underlying recursive structure.

Definition 11 (LTS Renaming). LTS t_1 is a *renaming* of LTS t_2 iff there is a renaming σ such that $t_1 \eqsim_\sigma^\varnothing t_2$, where the reflexive, transitive and symmetric relation $\eqsim_\sigma^\rho$ is defined as follows:

$$\begin{array}{l}
(x \rightarrow (x, \mathbf{0})) \eqsim_\sigma^\rho (x' \rightarrow (x', \mathbf{0})), \ \text{ if } x\sigma = x' \\
(e \rightarrow (\tau_f, t)) \eqsim_\sigma^\rho (e' \rightarrow (\tau_{f'}, t')), \text{ if } ((f, f') \in \rho) \vee (t \eqsim_\sigma^{\rho \cup \{(f,f')\}} t') \\
(e \rightarrow (\alpha_1, t_1), \ldots, (\alpha_n, t_n)) \eqsim_\sigma^\rho (e' \rightarrow (\alpha'_1, t'_1), \ldots, (\alpha'_n, t'_n)), \\
\qquad\qquad\qquad\qquad \text{if } \forall i \in \{1 \ldots n\} \bullet (\exists \sigma' \bullet (\alpha_i\ \sigma' = \alpha'_i \wedge t_i \eqsim_{\sigma \cup \sigma'}^\rho t'_i))
\end{array}$$

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 keep track of the corresponding function names which have been matched with each other.

Definition 12 (LTS Embedding). LTS t_1 is *embedded* within LTS t_2 iff there is a renaming σ such that $t_1 \lesssim_\sigma^\varnothing t_2$, where the reflexive, transitive and anti-symmetric relation $\lesssim_\sigma^\rho$ is defined as follows:

$$
\begin{array}{ll}
t \lesssim_\sigma^\rho t', & \text{if } (t \lhd_\sigma^\rho t') \vee (t \bowtie_\sigma^\rho t') \\
t \lhd_\sigma^\rho (e \to (\alpha_1, t_1), \ldots, (\alpha_n, t_n)), & \text{if } \exists i \in \{1 \ldots n\} \bullet t \lesssim_\sigma^\rho t_i \\
(x \to (x, \mathbf{0})) \bowtie_\sigma^\rho (x' \to (x', \mathbf{0})), & \text{if } x\sigma = x' \\
(e \to (\tau_f, t)) \bowtie_\sigma^\rho (e' \to (\tau_{f'}, t')), & \text{if } ((f, f') \in \rho) \vee (t \lesssim_\sigma^{\rho \cup \{(f,f')\}} t') \\
(e \to (\alpha_1, t_1), \ldots, (\alpha_n, t_n)) \bowtie_\sigma^\rho (e' \to (\alpha_1', t_1'), \ldots, (\alpha_n', t_n')), & \\
 & \text{if } \forall i \in \{1 \ldots n\} \bullet (\exists \sigma' \bullet (\alpha_i\ \sigma' = \alpha_i' \wedge t_i \lesssim_{\sigma \cup \sigma'}^\rho t_i'))
\end{array}
$$

The rules for embedding are applied in top-down order, where the final rule is a catch-all. One LTS is embedded within another by this relation if either *diving* (denoted by $\lhd_\sigma^\rho$) or *coupling* (denoted by $\bowtie_\sigma^\rho$) 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 keep track of the corresponding function names which have been coupled with each other.

Definition 13 (Generalization of LTSs). The function $\mathcal{G}[\![t]\!][\![t']\!]\ \theta$ that generalizes LTS t with respect to LTS t' is defined in Fig. 8, where θ is the set of previous generalizations which can be reused. The result of this function is the generalization of the LTS t, in which some sub-components have been extracted from t using **let**s.

Within the rules $\mathcal{G}'$, γ is the set of bound variables within the LTS being generalized and ρ is used to keep track of the corresponding function names which have been matched with each other. The rules are applied in top-down order. 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 introduce bound variables (λ, **case** and **let**); in these cases the bound variables are added to the set of bound variables γ. Unmatched LTS components are extracted into a substitution and replaced by variable applications. The arguments of the variable applications

$$\mathcal{G}[\![t]\!][\![t']\!]\ \theta = \mathcal{A}[\![t^g]\!]\ \theta'$$
where
$$(t^g, \theta') = \mathcal{G}'[\![t]\!][\![t']\!]\ \theta\ \varnothing\ \varnothing$$

$$\mathcal{G}'[\![e \rightarrow (x, \mathbf{0})]\!][\![e' \rightarrow (x', \mathbf{0})]\!]\ \theta\ \gamma\ \rho = ((e \rightarrow (x, \mathbf{0})), \varnothing)$$

$$\begin{aligned}
&\mathcal{G}'[\![e \rightarrow (c, \mathbf{0}), (\#1, t_1), \ldots, (\#n, t_n)]\!][\![e' \rightarrow (c, \mathbf{0}), (\#1, t_1'), \ldots, (\#n, t_n')]\!]\ \theta\ \gamma\ \rho \\
&\quad = ((e \rightarrow (c, \mathbf{0}), (\#1, t_1^g), \ldots, (\#n, t_n^g)), \textstyle\bigcup_{i=1}^{n} \theta_i) \\
&\quad\quad \text{where} \\
&\quad\quad \forall i \in \{1 \ldots n\} \bullet (t_i^g, \theta_i) = \mathcal{G}'[\![t_i]\!][\![t_i']\!]\ \theta\ \gamma\ \rho
\end{aligned}$$

$$\begin{aligned}
&\mathcal{G}'[\![e \rightarrow (\lambda x, t)]\!][\![e' \rightarrow (\lambda x', t')]\!]\ \theta\ \gamma\ \rho = ((e \rightarrow (\lambda x, t^g)), \theta') \\
&\quad\quad \text{where} \\
&\quad\quad (t^g, \theta') = \mathcal{G}'[\![t]\!][\![t']\!]\ (\gamma \cup \{x\})\ \rho
\end{aligned}$$

$$\begin{aligned}
&\mathcal{G}'[\![e \rightarrow (@, t_0), (\#1, t_1)]\!][\![e' \rightarrow (@, t_0'), (\#1, t_1')]\!]\ \theta\ \gamma\ \rho \\
&\quad = ((e \rightarrow (@, t_0^g), (\#1, t_1^g)), \theta_0 \cup \theta_1) \\
&\quad\quad \text{where} \\
&\quad\quad \forall i \in \{0, 1\} \bullet (t_i^g, \theta_i) = \mathcal{G}[\![t_i]\!][\![t_i']\!]\ \theta\ \gamma\ \rho
\end{aligned}$$

$$\begin{aligned}
&\mathcal{G}'[\![e \rightarrow (\mathbf{case}, t_0), (p_1, t_1), \ldots, (p_n, t_n)]\!][\![e' \rightarrow (\mathbf{case}, t_0'), (p_1', t_1'), \ldots, (p_n', t_n')]\!]\ \theta\ \gamma\ \rho \\
&\quad = ((e \rightarrow (\mathbf{case}, t_0^g), (p_1, t_1^g), \ldots, (p_n, t_n^g)), \textstyle\bigcup_{i=0}^{n} \theta_i) \\
&\quad\quad \text{where} \\
&\quad\quad \forall i \in \{1 \ldots n\} \bullet (\exists \sigma \bullet p_i \equiv p_i'\sigma) \\
&\quad\quad (t_0^g, \theta_0) = \mathcal{G}'[\![t_0]\!][\![t_0']\!]\ \theta\ \gamma\ \rho \\
&\quad\quad \forall i \in \{1 \ldots n\} \bullet (t_i^g, \theta_i) = \mathcal{G}'[\![t_i]\!][\![t_i']\!]\ \theta\ (\gamma \cup fv(p_i))\ \rho
\end{aligned}$$

$$\begin{aligned}
&\mathcal{G}'[\![e \rightarrow (\mathbf{let}, t_0), (x_1, t_1), \ldots, (x_n, t_n)]\!][\![e' \rightarrow (\mathbf{let}, t_0'), (x_1', t_1'), \ldots, (x_n', t_n')]\!]\ \theta\ \gamma\ \rho \\
&\quad = ((e \rightarrow (\mathbf{let}, t_0^g), (x_1, t_1^g), \ldots, (x_n, t_n^g)), \textstyle\bigcup_{i=0}^{n} \theta_i) \\
&\quad\quad \text{where} \\
&\quad\quad (t_0^g, \theta_0) = \mathcal{G}'[\![t_0]\!][\![t_0']\!]\ \theta\ \gamma\ \rho \\
&\quad\quad \forall i \in \{1 \ldots n\} \bullet (t_i^g, \theta_i) = \mathcal{G}'[\![t_i]\!][\![t_i']\!]\ \theta\ \gamma\ \rho
\end{aligned}$$

$$\mathcal{G}'[\![e \rightarrow (\tau_f, t)]\!][\![e' \rightarrow (\tau_{f'}, t')]\!]\ \theta\ \gamma\ \rho = \begin{cases} ((e \rightarrow (\tau_f, t)), \varnothing), & \text{if } (f, f') \in \rho \\ ((e \rightarrow (\tau_f, t^g)), \theta'), & \text{otherwise} \\ \text{where} & \\ (t^g, \theta') = \mathcal{G}'[\![t]\!][\![t']\!]\ \theta\ \gamma\ (\rho \cup \{(f, f')\}) & \end{cases}$$

$$\mathcal{G}'[\![e \rightarrow (\tau_\beta, t)]\!][\![e' \rightarrow (\tau_\beta, t')]\!]\ \theta\ \gamma\ \rho = \mathcal{G}'[\![t]\!][\![t']\!]\ \theta\ \gamma\ \rho$$

$$\mathcal{G}'[\![e \rightarrow (\tau_c, t)]\!][\![e' \rightarrow (\tau_c, t')]\!]\ \theta\ \gamma\ \rho = \mathcal{G}'[\![t]\!][\![t']\!]\ \theta\ \gamma\ \rho$$

$$\begin{aligned}
&\mathcal{G}'[\![t]\!][\![t']\!]\ \theta\ \gamma\ \rho = \begin{cases} (\mathcal{B}[\![x \rightarrow (x, \mathbf{0})]\!]\ \gamma', \varnothing), & \text{if } \exists (x, t_1) \in \theta \bullet t_1 \eqsim_{\varnothing}^{\varnothing} t_2 \\ (\mathcal{B}[\![x \rightarrow (x, \mathbf{0})]\!]\ \gamma', \{x \mapsto t_2\}), & \text{otherwise } (x \text{ is fresh}) \end{cases} \\
&\quad\quad \text{where} \\
&\quad\quad \gamma' = fv(t) \cap \gamma \\
&\quad\quad t_2 = \mathcal{C}[\![t]\!]\ \gamma'
\end{aligned}$$

$$\mathcal{A}[\![t]\!]\ \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\} = root(t) \rightarrow (\mathbf{let}, t), (x_1, t_1), \ldots, (x_n, t_n)$$

$$\mathcal{B}[\![t]\!]\ \{x_1 \ldots x_n\} = root(t) \rightarrow (@, (\ldots root(t) \rightarrow (@, t), (\#1, x_1 \rightarrow (x_1, \mathbf{0})) \ldots)), (\#1, x_n \rightarrow (x_n, \mathbf{0}))$$

$$\mathcal{C}[\![t]\!]\ \{x_1 \ldots x_n\} = root(t) \rightarrow (\lambda x_1, \ldots root(t) \rightarrow (\lambda x_n, t) \ldots)$$

Fig. 8. Rules for Generalization

introduced are the free variables of the LTS component which are also contained in the set of overall bound variables γ; this ensures that bound variables are not extracted outside their binders. If an extracted LTS component is contained in the set of previous generalizations θ, then the variable name from this previous generalization is reused. Otherwise, a new variable is introduced which is different and distinct from all other program variables.

4.1 Level 0 Transformer

We now define the level 0 program transformer within our hierarchy, which corresponds closely to the positive supercompilation algorithm. The transformer takes as its input the original program and produces as its output a labelled transition system, from which a new (hopefully improved) program can be residualized.

$$
\begin{array}{ll}
\mathcal{T}_0[\![x]\!]\ \kappa\ \rho\ \theta\ \Delta & = \mathcal{T}'_0[\![x \rightarrow (x, \mathbf{0})]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\mathcal{T}_0[\![e = c\ e_1 \ldots e_n]\!]\ \langle\rangle\ \rho\ \theta\ \Delta & \\
& = e \rightarrow (c, \mathbf{0}), (\#1, \mathcal{T}_0[\![e_1]\!]\ \langle\rangle\ \rho\ \theta\ \Delta), \ldots, (\#n, \mathcal{T}_0[\![e_n]\!]\ \langle\rangle\ \rho\ \theta\ \Delta) \\
\mathcal{T}_0[\![e = c\ e_1 \ldots e_n]\!]\ (\kappa = \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k) : \kappa'\rangle)\ \rho\ \theta\ \Delta & \\
& = (\kappa \bullet e) \rightarrow (\tau_c, \mathcal{T}_0[\![e'_i\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}]\!]\ \kappa'\ \rho\ \theta\ \Delta) \\
& \quad \text{where } p_i = c\ x_1 \ldots x_n \\
\mathcal{T}_0[\![e = \lambda x.e_0]\!]\ \langle\rangle\ \rho\ \theta\ \Delta & \\
& = e \rightarrow (\lambda x, \mathcal{T}_0[\![e_0]\!]\ \langle\rangle\ \rho\ \theta\ \Delta) \\
\mathcal{T}_0[\![e = \lambda x.e_0]\!]\ (\kappa = \langle(\bullet\ e_1) : \kappa'\rangle)\ \rho\ \theta\ \Delta & \\
& = (\kappa \bullet e) \rightarrow (\tau_\beta, \mathcal{T}_0[\![e_0\{x \mapsto e_1\}]\!]\ \kappa'\ \rho\ \theta\ \Delta) \\
\mathcal{T}_0[\![f]\!]\ \kappa\ \rho\ \theta\ \Delta & = \begin{cases} (\kappa \bullet f) \rightarrow (\tau_f, \mathbf{0}), & \text{if } \exists t' \in \rho, \sigma \bullet t' \eqsim^{\varnothing}_{\sigma} t \\ \mathcal{T}_0[\![\mathcal{R}[\![\mathcal{G}[\![t]\!][\![t']\!]\ \theta]\!]]\!]\ \langle\rangle\ \rho\ \theta\ \varnothing, & \text{if } \exists t' \in \rho, \sigma \bullet t' \bowtie^{\varnothing}_{\sigma} t \\ (\kappa \bullet f) \rightarrow (\tau_f, \mathcal{T}_0[\![\mathcal{U}[\![\kappa \bullet f]\!]\ \Delta]\!]\ \langle\rangle\ (\rho \cup \{t\})\ \theta\ \Delta), & \text{otherwise} \end{cases} \\
& \quad \text{where } t = \mathcal{L}[\![\kappa \bullet f]\!]\ \varnothing\ \Delta \\
\mathcal{T}_0[\![e_0\ e_1]\!]\ \kappa\ \rho\ \theta\ \Delta & = \mathcal{T}_0[\![e_0]\!]\ \langle(\bullet\ e_1) : \kappa\rangle\ \rho\ \theta\ \Delta \\
\mathcal{T}_0[\![\mathbf{case}\ e_0\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k]\!]\ \kappa\ \rho\ \theta\ \Delta & \\
& = \mathcal{T}_0[\![e_0]\!]\ \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k) : \kappa\rangle\ \rho\ \theta\ \Delta \\
\mathcal{T}_0[\![e = \mathbf{let}\ x = e_0\ \mathbf{in}\ e_1]\!]\ \kappa\ \rho\ \theta\ \Delta & \\
& = (\kappa \bullet e) \rightarrow (\mathbf{let}, \mathcal{T}_0[\![e_1]\!]\ \kappa\ (\rho \cup \{x \mapsto e_0\})\ \theta\ \Delta), (x, \mathcal{T}_0[\![e_0]\!]\ \langle\rangle\ \rho\ \theta\ \Delta) \\
\mathcal{T}_0[\![e_0\ \mathbf{where}\ f_1 = e_1 \ldots f_n = e_n]\!]\ \kappa\ \rho\ \theta\ \Delta & \\
& = \mathcal{T}_0[\![e_0]\!]\ \kappa\ \rho\ \theta\ (\Delta \cup \{f_1 = e_1, \ldots, f_n = e_n\})
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{T}'_0[\![t]\!]\ \langle\rangle\ \rho\ \theta\ \Delta & = t \\
\mathcal{T}'_0[\![t]\!]\ \langle(\bullet\ e) : \kappa\rangle\ \rho\ \theta\ \Delta & \\
& = \mathcal{T}'_0[\![(t\ e) \rightarrow (@, t), (\#1, \mathcal{T}_0[\![e]\!]\ \langle\rangle\ \rho\ \theta\ \Delta)]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\mathcal{T}'_0[\![x \rightarrow (x, \mathbf{0})]\!]\ \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k) : \kappa\rangle\ \rho\ \theta\ \Delta & \\
& = (\mathbf{case}\ x\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k) \rightarrow \\
& \quad (\mathbf{case}, t), (p_1, \mathcal{T}_0[\![(\kappa \bullet e'_1)\{x \mapsto p_1\}]\!]\ \langle\rangle\ \rho\ \theta\ \Delta), \ldots, (p_k, \mathcal{T}_0[\![(\kappa \bullet e'_k)\{x \mapsto p_k\}]\!]\ \langle\rangle\ \rho\ \theta\ \Delta) \\
\mathcal{T}'_0[\![t]\!]\ \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k) : \kappa\rangle\ \rho\ \theta\ \Delta & \\
& = (\mathbf{case}\ (root(t))\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k) \rightarrow \\
& \quad (\mathbf{case}, t), (p_1, \mathcal{T}_0[\![e'_1]\!]\ \kappa\ \rho\ \theta\ ), \ldots, (p_k, \mathcal{T}_0[\![e'_k]\!]\ \kappa\ \rho\ \theta\ \Delta)
\end{array}
$$

Fig. 9. Level 0 Transformation Rules

The level 0 transformer effectively performs a normal-order reduction of the input program. The (LTS representation of) previously encountered terms are 'memoized'. If the (LTS representation of the) current term is a renaming of a memoized one, then folding is performed, and the transformation is complete. If the (LTS representation of the) current term is an embedding of a memoized one, then generalization is performed, and the resulting generalized term is further transformed. Generalization ensures that a renaming of a previously encountered term is always eventually encountered, and that the transformation therefore terminates. The rules for level 0 transformation are as shown in Fig. 9.

The rules $\mathcal{T}_0$ are defined on an expression and its surrounding context, denoted by κ. The parameter ρ contains the LTS representations of memoized terms; for our language it is only necessary to add LTSs to ρ for terms in which the redex is a function, as any infinite sequence of reductions must include such terms. The parameter θ contains terms which have been extracted using a **let** expression as a result of generalization. If an identical term is subsequently extracted as a result of further generalization, then the extraction is removed and the same variable is used as for the previous extraction. The parameter Δ contains the set of function definitions.

The rules $\mathcal{T}'_0$ are defined on an LTS and its surrounding context, also denoted by κ. These rules are applied when the normal-order reduction of the input program becomes 'stuck' as a result of encountering a variable in the redex position. In this case, the context surrounding the redex is further transformed. If the context surrounding a variable redex is a **case**, then information is propagated to each branch of the **case** to indicate that this variable has the value of the corresponding branch pattern.

Example 4. The result of transforming the *fib* program in Fig. 2 using the level 0 transformer is shown in Fig. 10 (due to space constraints, we present the results of transformation in this and further examples as residualized programs rather than LTSs). As we can see, this is no real improvement over the original program.

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

Fig. 10. Result of Level 0 Transformation

4.2 Level $n + 1$ Transformer

We now define the transformers at all levels above 0 within our hierarchy. Each of these transformers makes use of the transformers below it in the hierarchy. The rules for a level $n+1$ transformer are actually very similar to those for the level 0 transformer; the level $n+1$ transformer also takes as its input the original program, performs normal-order reduction on it, and produces as its output a labelled transition system. Where the level $n+1$ transformer differs from that at level 0 is that the LTSs, which are memoized for the purposes of comparison when determining whether to fold or generalize, are those resulting from the level n transformation of previously encountered terms.

$$
\begin{array}{l}
\mathcal{T}_{n+1}[\![x]\!]\ \kappa\ \rho\ \theta\ \Delta = \mathcal{T}'_{n+1}[\![x \rightarrow (x, \mathbf{0})]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\mathcal{T}_{n+1}[\![e = c\ e_1 \ldots e_n]\!]\ \langle\rangle\ \rho\ \theta\ \Delta \\
\qquad = e \rightarrow (c, \mathbf{0}), (\#1, \mathcal{T}_{n+1}[\![e_1]\!]\ \langle\rangle\ \rho\ \theta\ \Delta), \ldots, (\#n, \mathcal{T}_{n+1}[\![e_n]\!]\ \langle\rangle\ \rho\ \theta\ \Delta) \\
\mathcal{T}_{n+1}[\![e = c\ e_1 \ldots e_n]\!]\ (\kappa = \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k) : \kappa'\rangle)\ \rho\ \theta\ \Delta \\
\qquad = (\kappa \bullet e) \rightarrow (\tau_c, \mathcal{T}_{n+1}[\![e'_i\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}]\!]\ \kappa'\ \rho\ \theta\ \Delta) \\
\qquad\quad \text{where } p_i = c\ x_1 \ldots x_n \\
\mathcal{T}_{n+1}[\![e = \lambda x.e_0]\!]\ \langle\rangle\ \rho\ \theta\ \Delta \\
\qquad = e \rightarrow (\lambda x, \mathcal{T}_{n+1}[\![e_0]\!]\ \langle\rangle\ \rho\ \theta\ \Delta) \\
\mathcal{T}_{n+1}[\![e = \lambda x.e_0]\!]\ (\kappa = \langle(\bullet\ e_1) : \kappa'\rangle)\ \rho\ \theta\ \Delta \\
\qquad = (\kappa \bullet e) \rightarrow (\tau_\beta, \mathcal{T}_{n+1}[\![e_0\{x \mapsto e_1\}]\!]\ \kappa'\ \rho\ \theta\ \Delta) \\
\mathcal{T}_{n+1}[\![f]\!]\ \kappa\ \rho\ \theta\ \Delta = \begin{cases} (\kappa \bullet f) \rightarrow (\tau_f, \mathbf{0}), & \text{if } \exists t' \in \rho, \sigma \bullet t' \eqsim^{\varnothing}_{\sigma} t \\ \mathcal{T}_{n+1}[\![\mathcal{R}[\![\mathcal{G}[\![t]\!][\![t']\!]\ \theta]\!]]\!]\ \langle\rangle\ \rho\ \theta\ \varnothing, & \text{if } \exists t' \in \rho, \sigma \bullet t' \bowtie^{\varnothing}_{\sigma} t \\ (\kappa \bullet f) \rightarrow (\tau_f, \mathcal{T}_{n+1}[\![\mathcal{U}[\![\mathcal{R}[\![t]\!]]\!]\ \varnothing]\!]\ \langle\rangle\ (\rho \cup \{t\})\ \theta\ \varnothing), & \text{otherwise} \end{cases} \\
\qquad\quad \text{where } t = \mathcal{T}_n[\![f]\!]\ \kappa\ \varnothing\ \theta\ \Delta \\
\mathcal{T}_{n+1}[\![e_0\ e_1]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\qquad = \mathcal{T}_{n+1}[\![e_0]\!]\ \langle(\bullet\ e_1) : \kappa\rangle\ \rho\ \theta\ \Delta \\
\mathcal{T}_{n+1}[\![\mathbf{case}\ e_0\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\qquad = \mathcal{T}_{n+1}[\![e_0]\!]\ \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k) : \kappa\rangle\ \rho\ \theta\ \Delta \\
\mathcal{T}_{n+1}[\![e = \mathbf{let}\ x = e_0\ \mathbf{in}\ e_1]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\qquad = (\kappa \bullet e) \rightarrow (\mathbf{let}, \mathcal{T}_{n+1}[\![e_1\{x \mapsto e_0\}]\!]\ \kappa\ (\rho \cup \{x \mapsto e_0\})\ \theta\ \Delta), (x, \mathcal{T}_{n+1}[\![e_0]\!]\ \langle\rangle\ \rho\ \theta\ \Delta) \\
\mathcal{T}_{n+1}[\![e_0\ \mathbf{where}\ f_1 = e_1 \ldots f_n = e_n]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\qquad = \mathcal{T}_{n+1}[\![e_0]\!]\ \kappa\ \rho\ \theta\ (\Delta \cup \{f_1 = e_1, \ldots, f_n = e_n\})
\end{array}
$$

$$
\begin{array}{l}
\mathcal{T}'_{n+1}[\![t]\!]\ \langle\rangle\ \rho\ \theta\ \Delta = t \\
\mathcal{T}'_{n+1}[\![t]\!]\ \langle(\bullet\ e) : \kappa\rangle\ \rho\ \theta\ \Delta \\
\qquad = \mathcal{T}'_{n+1}[\![(t\ e) \rightarrow (@, t), (\#1, \mathcal{T}_{n+1}[\![e]\!]\ \langle\rangle\ \rho\ \theta\ )]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\mathcal{T}'_{n+1}[\![x \rightarrow (x, \mathbf{n+1})]\!]\ \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k) : \kappa\rangle\ \rho\ \theta\ \Delta \\
\qquad = (\mathbf{case}\ x\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k) \rightarrow \\
\qquad\quad (\mathbf{case}, t), (p_1, \mathcal{T}_{n+1}[\![(\kappa \bullet e'_1)\{x \mapsto p_1\}]\!]\ \langle\rangle\ \rho\ \theta\ \Delta), \ldots, (p_k, \mathcal{T}_{n+1}[\![(\kappa \bullet e'_k)\{x \mapsto p_k\}]\!]\ \langle\rangle\ \rho\ \theta\ \Delta) \\
\mathcal{T}'_{n+1}[\![t]\!]\ \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k) : \kappa\rangle\ \rho\ \theta\ \Delta \\
\qquad = (\mathbf{case}\ root(t)\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k) \rightarrow \\
\qquad\quad (\mathbf{case}, t), (p_1, \mathcal{T}_{n+1}[\![e'_1]\!]\ \kappa\ \rho\ \theta\ \Delta), \ldots, (p_k, \mathcal{T}_{n+1}[\![e'_k]\!]\ \kappa\ \rho\ \theta\ \Delta)
\end{array}
$$

Fig. 11. Level $n + 1$ Transformation Rules

After the LTS resulting from level n transformation has been memoized, it is residualized, unfolded and further transformed. If a renaming of a memoized LTS is encountered, then folding is performed. If an embedding of a memoized LTS is encountered, then generalization is performed; this generalization will have the effect of adding an extra layer of **let**s around the LTS. Thus, each successive level in the transformer hierarchy will have the effect of adding an extra layer of **let**s around the LTS representation of the current term.

As over-generalization may occur at each level in the transformer hierarchy, the extracted terms at each level are substituted back in at the next successive level. The **let**s themselves are retained as the extracted terms could be re-encountered in further generalization, and the **let** variables reused; if they are not reused, then the **let** can be removed in a post-processing phase. Note that at each successive level in the transformer hierarchy, the size of the components which are extracted by generalization must get smaller as they are sub-components of the result of transformation at the previous level.

The level $n + 1$ transformation rules are defined as shown in Fig. 11. The parameters used within these rules are the same as those for the level 0 transformer, except that the memoization environment ρ contains the LTSs resulting from the level n transformation of previously encountered terms.

Example 5. If the result of the level 0 transformation of the *fib* program shown in Fig. 10 is further transformed within a level 1 transformer, then the level 0 result as shown in Fig. 12 is encountered.

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

Fig. 12. Further Result of Level 0 Transformation

Generalization is therefore performed with respect to the previous level 0 result in Fig. 10 to obtain the level 1 result shown in Fig. 13. We can see that an extra layer of **let**s has been added around this program. If this level 1 program is further transformed within a level 2 transformer, then the level 1 result as shown in Fig. 14 is encountered.

$$
\begin{array}{l}
\textbf{let}\ x_1\ =\ S\ Z\\
\textbf{in let}\ x_2\ =\ S\ S\ Z\\
\quad \textbf{in}\ f\ n\\
\qquad \textbf{where}\\
\qquad f = \lambda n.\,\textbf{case}\ n\ \textbf{of}\\
\qquad\qquad\qquad Z \Rightarrow S\ x_1\\
\qquad\qquad\quad\ \mid S\ n' \Rightarrow \textbf{case}\ n'\ \textbf{of}\\
\qquad\qquad\qquad\qquad\qquad Z \Rightarrow S\ x_2\\
\qquad\qquad\qquad\qquad\ \ \mid S\ n'' \Rightarrow \textbf{case}\ (f\ n'')\ \textbf{of}\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad Z \Rightarrow f\ (S\ n'')\\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mid S\ x \Rightarrow S\ (g\ x\ n'')\\
\qquad g = \lambda x.\lambda n.\,\textbf{case}\ x\ \textbf{of}\\
\qquad\qquad\qquad\quad Z \Rightarrow f\ (S\ n)\\
\qquad\qquad\qquad \mid S\ x' \Rightarrow S\ (g\ x'\ n)
\end{array}
$$

Fig. 13. Result of Level 1 Transformation

$$
\begin{array}{l}
\textbf{let}\ x_3\ =\ S\ S\ S\ Z\\
\textbf{in let}\ x_4\ =\ S\ S\ S\ S\ S\ Z\\
\quad \textbf{in}\ f\ n\\
\qquad \textbf{where}\\
\qquad f = \lambda n.\,\textbf{case}\ n\ \textbf{of}\\
\qquad\qquad\qquad Z \Rightarrow S\ S\ x_3\\
\qquad\qquad\quad\ \mid S\ n' \Rightarrow \textbf{case}\ n'\ \textbf{of}\\
\qquad\qquad\qquad\qquad\qquad Z \Rightarrow S\ S\ S\ x_4\\
\qquad\qquad\qquad\qquad\ \ \mid S\ n'' \Rightarrow \textbf{case}\ (f\ n'')\ \textbf{of}\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad Z \Rightarrow f\ (S\ n'')\\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mid S\ x \Rightarrow S\ (g\ x\ n'')\\
\qquad g = \lambda x.\lambda n.\,\textbf{case}\ x\ \textbf{of}\\
\qquad\qquad\qquad\quad Z \Rightarrow f\ (S\ n)\\
\qquad\qquad\qquad \mid S\ x' \Rightarrow S\ (g\ x'\ n)
\end{array}
$$

Fig. 14. Further Result of Level 1 Transformation

Generalization is therefore performed with respect to the previous level 1 result in Fig. 13 to obtain the level 2 result shown in Fig. 15. Again, we can see that an extra layer of **let**s has been added around this program. If this level 2 program is further transformed within a level 3 transformer, then the level 2 result as shown in Fig. 16 is encountered. Generalization is therefore performed with respect to the previous level 2 result in Fig. 15 to obtain the level 3 result shown in Fig. 17. In this case, we can see that an extra layer of **let**s has not been added around the program; this is because the components which would have been extracted had been extracted previously, so the variables from these previous extractions were reused. If this level 3 program is further transformed within a level 4 transformer, then the level 3 result as shown in Fig. 18 is encountered.

$$
\begin{array}{l}
\textbf{let } x_5 \;=\; \lambda x.S\ x \\
\textbf{in let } x_6 \;=\; \lambda x.S\ S\ x \\
\quad \textbf{in let } x_3 \;=\; S\ x_2 \\
\quad\quad \textbf{in let } x_4 \;=\; S\ S\ x_3 \\
\quad\quad\quad \textbf{in } f\ n \\
\quad\quad\quad\quad \textbf{where} \\
\quad\quad\quad\quad f = \lambda n.\,\textbf{case } n \textbf{ of} \\
\quad\quad\quad\quad\quad\quad\quad\quad Z \quad \Rightarrow S\ (x_5\ x_3) \\
\quad\quad\quad\quad\quad\quad\quad\quad \mid S\ n' \Rightarrow \textbf{case } n' \textbf{ of} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad Z \quad \Rightarrow S\ (x_6\ x_4) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid S\ n'' \Rightarrow \textbf{case } (f\ n'') \textbf{ of} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad Z \quad \Rightarrow f\ (S\ n'') \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid S\ x \Rightarrow S\ (g\ x\ n'') \\
\quad\quad\quad\quad g = \lambda x.\lambda n.\,\textbf{case } x \textbf{ of} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad Z \quad \Rightarrow f\ (S\ n) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid S\ x' \Rightarrow S\ (g\ x'\ n)
\end{array}
$$

Fig. 15. Result of Level 2 Transformation

$$
\begin{array}{l}
\textbf{let } x_9 \;=\; \lambda x.S\ S\ S\ x \\
\textbf{in let } x_{10} \;=\; \lambda x.S\ S\ S\ S\ S\ x \\
\quad \textbf{in let } x_7 \;=\; S\ S\ S\ x_4 \\
\quad\quad \textbf{in let } x_8 \;=\; S\ S\ S\ S\ S\ x_7 \\
\quad\quad\quad \textbf{in } f\ n \\
\quad\quad\quad\quad \textbf{where} \\
\quad\quad\quad\quad f = \lambda n.\,\textbf{case } n \textbf{ of} \\
\quad\quad\quad\quad\quad\quad\quad\quad Z \quad \Rightarrow S\ (x_9\ x_7) \\
\quad\quad\quad\quad\quad\quad\quad\quad \mid S\ n' \Rightarrow \textbf{case } n' \textbf{ of} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad Z \quad \Rightarrow S\ (x_{10}\ x_8) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid S\ n'' \Rightarrow \textbf{case } (f\ n'') \textbf{ of} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad Z \quad \Rightarrow f\ (S\ n'') \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid S\ x \Rightarrow S\ (g\ x\ n'') \\
\quad\quad\quad\quad g = \lambda x.\lambda n.\,\textbf{case } x \textbf{ of} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad Z \quad \Rightarrow f\ (S\ n) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid S\ x' \Rightarrow S\ (g\ x'\ n)
\end{array}
$$

Fig. 16. Further Result of Level 2 Transformation

We can now see that the level 3 result in Fig. 18 is a renaming of the level 3 result in Fig. 17. Folding is therefore performed to obtain the result in Fig. 19 (for the sake of brevity, this program has been compressed, but the actual result can be obtained from this by performing a couple of function unfoldings).

We can see that the original program which contained double recursion has been transformed into one with single recursion. Note that the result we have obtained is not the linear version of the *fib* function which we might have expected. This is because the addition operation within the original program is not a constant-time operation; it is a linear-time operation defined on Peano

$$
\begin{array}{l}
\textbf{let } x_9 \;=\; \lambda x.x_5\ (x_6\ x)\\
\textbf{in let } x_{10} \;=\; \lambda x.x_6\ (x_5\ (x_6\ x))\\
\quad \textbf{in let } x_7 \;=\; x_5\ (x_6\ x_4)\\
\quad\quad \textbf{in let } x_8 \;=\; x_6\ (x_5\ (x_6\ x_4))\\
\quad\quad\quad \textbf{in } f\ n\\
\quad\quad\quad\quad \textbf{where}\\
\quad\quad\quad\quad f = \lambda n.\,\textbf{case } n \textbf{ of}\\
\quad\quad\quad\quad\quad\quad Z \Rightarrow S\ (x_9\ x_7)\\
\quad\quad\quad\quad\quad\quad \mid S\ n' \Rightarrow \textbf{case } n' \textbf{ of}\\
\quad\quad\quad\quad\quad\quad\quad\quad Z \Rightarrow S\ (x_{10}\ x_8)\\
\quad\quad\quad\quad\quad\quad\quad\quad \mid S\ n'' \Rightarrow \textbf{case } (f\ n'') \textbf{ of}\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad Z \Rightarrow f\ (S\ n'')\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid S\ x \Rightarrow S\ (g\ x\ n'')\\
\quad\quad\quad\quad g = \lambda x.\lambda n.\,\textbf{case } x \textbf{ of}\\
\quad\quad\quad\quad\quad\quad Z \Rightarrow f\ (S\ n)\\
\quad\quad\quad\quad\quad\quad \mid S\ x' \Rightarrow S\ (g\ x'\ n)
\end{array}
$$

Fig. 17. Result of Level 3 Transformation

$$
\begin{array}{l}
\textbf{let } x_{13} \;=\; \lambda x.x_9\ (x_{10}\ x)\\
\textbf{in let } x_{14} \;=\; \lambda x.x_{10}\ (x_9\ (x_{10}\ x))\\
\quad \textbf{in let } x_{11} \;=\; x_9\ (x_{10}\ x_8)\\
\quad\quad \textbf{in let } x_{12} \;=\; x_{10}\ (x_9\ (x_{10}\ x_8))\\
\quad\quad\quad \textbf{in } f\ n\\
\quad\quad\quad\quad \textbf{where}\\
\quad\quad\quad\quad f = \lambda n.\,\textbf{case } n \textbf{ of}\\
\quad\quad\quad\quad\quad\quad Z \Rightarrow S\ (x_{13}\ x_{11})\\
\quad\quad\quad\quad\quad\quad \mid S\ n' \Rightarrow \textbf{case } n' \textbf{ of}\\
\quad\quad\quad\quad\quad\quad\quad\quad Z \Rightarrow S\ (x_{14}\ x_{12})\\
\quad\quad\quad\quad\quad\quad\quad\quad \mid S\ n'' \Rightarrow \textbf{case } (f\ n'') \textbf{ of}\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad Z \Rightarrow f\ (S\ n'')\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid S\ x \Rightarrow S\ (g\ x\ n'')\\
\quad\quad\quad\quad g = \lambda x.\lambda n.\,\textbf{case } x \textbf{ of}\\
\quad\quad\quad\quad\quad\quad Z \Rightarrow f\ (S\ n)\\
\quad\quad\quad\quad\quad\quad \mid S\ x' \Rightarrow S\ (g\ x'\ n)
\end{array}
$$

Fig. 18. Further Result of Level 3 Transformation

numbers. If we had used a constant-time addition operator, then we would have obtained the linear version of the *fib* function using transformers at level 1 and upwards. This requires building the addition operator into our language, and defining specific transformation rules for such built-in operators which transform their arguments in sequence.

$$
\begin{array}{l}
\textbf{case}\ n\ \textbf{of} \\
\quad\ \ Z \quad\ \Rightarrow S\ Z \\
\quad \mid S\ n' \Rightarrow \textbf{case}\ n'\ \textbf{of} \\
\qquad\qquad\qquad\quad Z \quad\ \Rightarrow S\ Z \\
\qquad\qquad\qquad \mid S\ n'' \Rightarrow f\ n''\ (\lambda x.S\ x)\ (\lambda x.S\ S\ x)\ Z\ Z \\
\textbf{where} \\
f = \lambda n.\lambda g.\lambda h.\lambda x.\lambda y.\,\textbf{case}\ n\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\quad Z \quad\ \Rightarrow S\ (g\ x) \\
\qquad\qquad\qquad\qquad\qquad \mid S\ n' \Rightarrow \textbf{case}\ n'\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad Z \quad\ \Rightarrow S(h\ y) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid S\ n'' \Rightarrow f\ n''\ (\lambda x.g\ (h\ x))\ (\lambda x.h\ (g\ (h\ x)))\ (g\ x)\ (h\ y)
\end{array}
$$

Fig. 19. Overall Result of Level 4 Transformation

5 Distillation

In this section, we show how distillation can be described within our hierarchy of program transformers. One difficulty with having such a hierarchy of transformers is knowing which level within the hierarchy is sufficient to obtain the desired results. For example, for many of the examples in the literature of programs which are improved by positive supercompilation (level 0 in our hierarchy), no further improvements are obtained at higher levels in the hierarchy. However, only linear improvements in efficiency are possible at this level [13]. Higher levels in the hierarchy are capable of obtaining super-linear improvements in efficiency, but are overkill in many cases.

We therefore give a formulation of distillation which initially performs at level 0 in our hierarchy, and only moves to higher levels when necessary. Moving to a higher level in the hierarchy is only necessary if generalization has to be performed at the current level. Thus, when generalization is performed, the result of the generalization is memoized and is used for comparisons at the next level when checking for renamings or embeddings.

The rules for distillation are shown in Fig. 20. These rules are very similar to those for the transformers within the hierarchy; they take the original program as input, perform normal-order reduction, and produce a labelled transition system as output. The rules differ from those for the transformers within the hierarchy in that when generalization has to be performed, the LTS resulting from generalization at the current level is memoized, residualized, unfolded and then transformed at the next level up in the transformer hierarchy. Thus, each generalization will have the effect of moving up to the next level in the transformer hierarchy in addition to adding an extra layer of **let**s around the LTS representation of the current term.

We have already seen that at each successive level in the transformer hierarchy, the size of the components which are extracted by generalization must get smaller as they are sub-components of the result of transformation at the previous level. A point must therefore always be reached eventually at which extracted

$$
\begin{array}{l}
\mathcal{D}_n\llbracket x\rrbracket\ \kappa\ \rho\ \theta\ \Delta = \mathcal{D}'_n\llbracket x \rightarrow (x,\mathbf{0})\rrbracket\ \kappa\ \rho\ \theta\ \Delta \\
\mathcal{D}_n\llbracket e = c\ e_1 \ldots e_n\rrbracket\ \langle\rangle\ \rho\ \theta\ \Delta \\
\qquad\qquad = e \rightarrow (c,\mathbf{0}), (\#1, \mathcal{D}_n\llbracket e_1\rrbracket\ \langle\rangle\ \rho\ \theta\ \Delta), \ldots, (\#n, \mathcal{D}_n\llbracket e_n\rrbracket\ \langle\rangle\ \rho\ \theta\ \Delta) \\
\mathcal{D}_n\llbracket e = c\ e_1 \ldots e_n\rrbracket\ (\kappa = \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k) : \kappa'\rangle)\ \rho\ \theta\ \Delta \\
\qquad\qquad = (\kappa\bullet e) \rightarrow (\tau_c, \mathcal{D}_n\llbracket e'_i\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}\rrbracket\ \kappa'\ \rho\ \theta\ \Delta) \\
\qquad\qquad\quad \text{where } p_i = c\ x_1 \ldots x_n \\
\mathcal{D}_n\llbracket e = \lambda x.e_0\rrbracket\ \langle\rangle\ \rho\ \theta\ \Delta \\
\qquad\qquad = e \rightarrow (\lambda x, \mathcal{D}_n\llbracket e_0\rrbracket\ \langle\rangle\ \rho\ \theta\ \Delta) \\
\mathcal{D}_n\llbracket e = \lambda x.e_0\rrbracket\ (\kappa = \langle(\bullet\ e_1) : \kappa'\rangle)\ \rho\ \theta\ \Delta \\
\qquad\qquad = (\kappa\bullet e) \rightarrow (\tau_\beta, \mathcal{D}_n\llbracket e_0\{x \mapsto e_1\}\rrbracket\ \kappa'\ \rho\ \theta\ \Delta) \\
\mathcal{D}_n\llbracket f\rrbracket\ \kappa\ \rho\ \theta\ \Delta = \begin{cases} (\kappa\bullet f) \rightarrow (\tau_f, \mathbf{0}), & \text{if } \exists t' \in \rho, \sigma \bullet t' \eqsim^{\emptyset}_{\sigma} t \\ \mathcal{D}_{n+1}\llbracket \mathcal{U}\llbracket \mathcal{R}\llbracket t^g\rrbracket\rrbracket\ \emptyset\rrbracket\ \langle\rangle\ \{t^g\}\ \theta\ \emptyset, & \text{if } \exists t' \in \rho, \sigma \bullet t' \bowtie^{\emptyset}_{\sigma} t \\ \text{where } t^g = \mathcal{G}\llbracket t\rrbracket\llbracket t'\rrbracket\ \theta & \\ (\kappa\bullet f) \rightarrow (\tau_f, \mathcal{D}_n\llbracket \mathcal{U}\llbracket \mathcal{R}\llbracket t\rrbracket\rrbracket\ \emptyset\rrbracket\ \langle\rangle\ (\rho \cup \{t\})\ \theta\ \emptyset), & \text{otherwise} \end{cases} \\
\qquad\qquad\quad \text{where } t = \mathcal{T}_n\llbracket f\rrbracket\ \kappa\ \emptyset\ \theta\ \Delta \\
\mathcal{D}_n\llbracket e_0\ e_1\rrbracket\ \kappa\ \rho\ \theta\ \Delta \\
\qquad\qquad = \mathcal{D}_n\llbracket e_0\rrbracket\ \langle(\bullet\ e_1) : \kappa\rangle\ \rho\ \theta\ \Delta \\
\mathcal{D}_n\llbracket \mathbf{case}\ e_0\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k\rrbracket\ \kappa\ \rho\ \theta\ \Delta \\
\qquad\qquad = \mathcal{D}_n\llbracket e_0\rrbracket\ \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k) : \kappa\rangle\ \rho\ \theta\ \Delta \\
\mathcal{D}_n\llbracket e = \mathbf{let}\ x = e_0\ \mathbf{in}\ e_1\rrbracket\ \kappa\ \rho\ \theta\ \Delta \\
\qquad\qquad = (\kappa\bullet e) \rightarrow (\mathbf{let}, \mathcal{D}_n\llbracket e_1\{x \mapsto e_0\}\rrbracket\ \kappa\ (\rho \cup \{x \mapsto e_0\})\ \theta\ \Delta), (x, \mathcal{D}_n\llbracket e_0\rrbracket\ \langle\rangle\ \rho\ \theta\ \Delta) \\
\mathcal{D}_n\llbracket e_0\ \mathbf{where}\ f_1 = e_1 \ldots f_n = e_n\rrbracket\ \kappa\ \rho\ \theta\ \Delta \\
\qquad\qquad = \mathcal{D}_n\llbracket e_0\rrbracket\ \kappa\ \rho\ \theta\ (\Delta \cup \{f_1 = e_1, \ldots, f_n = e_n\})
\end{array}
$$

$$
\begin{array}{l}
\mathcal{D}'_n\llbracket t\rrbracket\ \langle\rangle\ \rho\ \theta\ \Delta = t \\
\mathcal{D}'_n\llbracket t\rrbracket\ \langle(\bullet\ e) : \kappa\rangle\ \rho\ \theta\ \Delta \\
\qquad\qquad = \mathcal{D}'_n\llbracket (t\ e) \rightarrow (@, t), (\#1, \mathcal{D}_n\llbracket e\rrbracket\ \langle\rangle\ \rho\ \theta\ )\rrbracket\ \kappa\ \rho\ \theta\ \Delta \\
\mathcal{D}'_n\llbracket x \rightarrow (x, \mathbf{n+1})\rrbracket\ \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k) : \kappa\rangle\ \rho\ \theta\ \Delta \\
\qquad\qquad = (\mathbf{case}\ x\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k) \rightarrow \\
\qquad (\mathbf{case}, t), (p_1, \mathcal{D}_n\llbracket (\kappa\bullet e'_1)\{x \mapsto p_1\}\rrbracket\ \langle\rangle\ \rho\ \theta\ \Delta), \ldots, (p_k, \mathcal{D}_n\llbracket (\kappa\bullet e'_k)\{x \mapsto p_k\}\rrbracket\ \langle\rangle\ \rho\ \theta\ \Delta) \\
\mathcal{D}'_n\llbracket t\rrbracket\ \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k) : \kappa\rangle\ \rho\ \theta\ \Delta \\
\qquad\qquad = (\mathbf{case}\ root(t)\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k) \rightarrow \\
\qquad\qquad\qquad (\mathbf{case}, t), (p_1, \mathcal{D}_n\llbracket e'_1\rrbracket\ \kappa\ \rho\ \theta\ \Delta), \ldots, (p_k, \mathcal{D}_n\llbracket e'_k\rrbracket\ \kappa\ \rho\ \theta\ \Delta)
\end{array}
$$

Fig. 20. Distillation Transformation Rules

components re-occur, and the same generalization variables will be reused without introducing new **let**s. At this point, no further generalization will be done and the distiller will not move any further up the transformer hierarchy and must terminate at the current level.

Example 6. The result of transforming the *fib* program in Fig. 2 using distillation is the same as that of the level 4 transformer within our transformation hierarchy shown in Fig. 19.

6 Conclusion and Related Work

We have defined a hierarchy of transformers in which the transformer at each level of the hierarchy makes use of the transformers at lower levels. At the bottom of the hierarchy is the level 0 transformer, which corresponds to positive supercompilation [14], and is capable of achieving only linear improvements in efficiency. The level 1 transformer corresponds to the first published definition of distillation [4], and is capable of achieving super-linear improvements in efficiency. Further improvements are possible at higher levels in the hierarchy, but the difficulty is in knowing the appropriate level at which to operate for an arbitrary input program. We have also shown how the more recently published definition of distillation [5] moves up through the levels of the transformation hierarchy until no further improvements can be made.

Previous works [9, 2, 1, 18, 13] have noted that the unfold/fold transformation methodology is incomplete; some programs cannot be synthesized from each other. This is because the transformation methodologies under consideration correspond to level 0 in our hierarchy; higher levels are required to achieve the desired results.

There have been several attempts to work on a meta-level above supercompilation, the first one by Turchin himself using *walk grammars* [16]. 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 [8]. 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 [10]). 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 [3]. Superlinear speedups can be achieved in logic program transformation by *goal replacement* [11, 12]: replacing one logical clause with another to facilitate folding. Techniques similar to the notion of "higher level supercompilation" [8] have been used to prove correctness of goal replacement, but have similar problems regarding the search for appropriate lemmas.

Acknowledgements

The work presented here owes a lot to the input of Neil Jones, who provided many useful insights and ideas in our collaboration during the sabbatical of the author

at the Department of Computer Science, University of Copenhagen (DIKU). 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. Ph.D. thesis, DAIMI, Aarhus University (1993)
2. Andersen, L., Gomard, C.: 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. Glück, R., Jørgensen, J.: Generating Transformers for Deforestation and Supercompilation. In: Proceedings of the Static Analysis Symposium. Lecture Notes in Computer Science, vol. 864, pp. 432–448. Springer-Verlag (1994)
4. Hamilton, G.W.: Distillation: Extracting the Essence of Programs. In: Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation. pp. 61–70 (2007)
5. Hamilton, G.W., Jones, N.D.: Distillation with Labelled Transition Systems. In: Proceedings of the SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation. pp. 15–24 (2012)
6. Huet, G.: The Zipper. Journal of Functional Programming 7(5), 549–554 (1997)
7. Jones, N., Gomard, C., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice Hall (1993)
8. Klyuchnikov, I., Romanenko, S.: Towards Higher-Level Supercompilation. In: Proceedings of the Second International Workshop on Metacomputation in Russia (META) (2010)
9. Kott, L.: A System for Proving Equivalences of Recursive Programs. In: Proceedings of the Fifth Conference on Automated Deduction (CADE). pp. 63–69 (1980)
10. Kott, L.: Unfold/Fold Transformations. In: Nivat, M., Reynolds, J. (eds.) Algebraic Methods in Semantics, chap. 12, pp. 412–433. CUP (1985)
11. Petterossi, A., Proietti, M.: A Theory of Totally Correct Logic Pro- gram Transformations. In: Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM). pp. 159–168 (2004)
12. Roychoudhury, A., Kumar, K., Ramakrishnan, C., Ramakrishnan, I.: An Unfold/Fold Transformation Framework for Definite Logic Programs. ACM Transactions on Programming Language Systems 26(3), 464–509 (2004)
13. Sørensen, M.H.: Turchin's Supercompiler Revisited. Master's thesis, Department of Computer Science, University of Copenhagen (1994), DIKU-rapport 94/17
14. Sørensen, M.H., Glück, R., Jones, N.: A Positive Supercompiler. Journal of Functional Programming 6(6), 811–838 (1996)
15. Turchin, V.: The Concept of a Supercompiler. ACM Transactions on Programming Languages and Systems 8(3), 90–121 (Jul 1986)
16. Turchin, V.: Program Transformation With Metasystem Transitions. Journal of Functional Programming 3(3), 283–313 (1993)
17. Wadler, P.: Deforestation: Transforming programs to eliminate trees. Lecture Notes in Computer Science 300, 344–358 (1988)
18. Zhu, H.: How Powerful are Folding/Unfolding Transformations? Journal of Functional Programming 4(1), 89–112 (1994)