

# Supercompilation and Normalisation By Evaluation

Gavin E. Mendel-Gleason and Geoff Hamilton

Dublin City University

**Abstract.** It has been long recognised that partial evaluation is related to proof normalisation. Normalisation by evaluation, which has been presented for theories with simple types, has made this correspondance formal. Recently Andreas Abel formalised an algorithm for normalisation by evaluation for System  $F$ . This is an important step towards the use of such techniques on practical functional programming languages such as Haskell which can reasonably be embedded in relatives of System  $F_\omega$ . Supercompilation is a program transformation technique which performs a superset of the simplications performed by partial evaluation. The focus of this paper is to formalise the relationship between supercompilation and normalisation by evaluation.

## 1 Introduction

Partial evaluation has arisen in two rather distinct settings. The first setting is in practical attempts to improve program performance. The second setting is in the attempts to simplify proofs by performing cut-eliminations. The use of recursion however fundamentally restricts the practical use of evaluation as a tool for normalisation. Due to this fact the program transformation community has developed a number of tools for improving the performance of programs, including deforestation [16], fusion [8] and supercompilation [11]

Supercompilation [15] is a program transformation which performs a superset of the optimisations done by fusion and deforestation. Supercompilation is a complex program transformation making use of *folds* [3]. Folds, which introduce new recursive structures, can sometimes introduce non-termination so certain side conditions must be met in order to ensure their correctness.

Bisimilarity, a technique developed by Milner [7] was used by Gordon [5] as an alternative to context equivalence for showing semantic equivalence of programs in all contexts. This is achieved by associating terms with transition systems, and showing bisimilarity of the transition systems. The technique can be usefully applied to automatic program transformations in order to prove correctness.

This paper introduces several novel developments. It demonstrates a transition system framework for System  $F$  with recursive types. This gives allows us to define a bisimulation relation on recursive terms in System  $F$  which demonstrates behavioural equivalence. It uses these transition systems as a semantic domain to present a system which closely resembles Normalisation by Evaluation (NbE)

[2] [1]. The techniques which are already common place in the supercompilation and meta-computation communities of creating partial process trees and then extracting programs is formalised in such a way as to demonstrate the connection with NbE.

## 2 Language

The language we present is a functional programming language, which we will call  $\Lambda_F$  with a type system based on System  $F$  with recursive types. The use of System  $F$  typing allows us to ensure that transitions can be found for any term. Our term language will follow closely on the one used by Abel [1]. We will use two distinct sets of variables for our exposition, term variables  $x, y$  drawn from the set **Var** and type variables  $X$  drawn from the set **TyVar**.

<b>Ty</b> $\ni A, B, C$	$::= X \mid A \rightarrow B \mid \forall X. A \mid A + B \mid A \times B$	Types
	$\mid \mu X. A$	
<b>Tr</b> $\ni r, s, t$	$::= x \mid \lambda x : A. t \mid \Lambda X. t \mid r \ s \mid r \ A$	Terms
	$\mid \text{case } r \text{ of } \text{inl}(x_1) \Rightarrow s \mid \text{inr}(x_2) \Rightarrow t \mid \text{split } r \text{ as } x_1, x_2 \text{ in } s$	
	$\mid \text{inl}(t) \mid \text{inr}(t) \mid (t, s) \mid \text{fold}(t, A) \mid \text{unfold}(t, A)$	
<b>Ctx</b> $\ni \Gamma$	$::= \cdot \mid \Gamma, x : A$	Contexts

We will describe *substitutions* using the map  $\sigma$  which will represent assignment of variables to terms and type variables to types. Extension of a substitution will be written as  $\sigma \cup (x, t)$  or  $\sigma \cup (X, A)$ . We will use a function  $FV(t)$  to obtain the free type and term variables from a term. Substitutions of a single variable will be written  $[X := A]$  or  $[x := t]$  for type and term variables respectively.

In order to make our presentation interesting, we will also need to introduce recursive terms. This will present us with the fundamental difference between standard presentations of NbE and our framework for supercompilation. Function constants will be drawn from a set **F**. We will couple our proofs with a function  $\Delta$  which associates a function constant  $f$  with a term  $e$ ,  $\Delta(f) = e$ , where  $e$  may itself contain the function constant.

The use of  $\Delta$  will allow us to use arbitrary recursive and mutually recursive function definitions. In so doing, however, we will need to add a rule to System  $F$  which will make our type theory potentially unsound in a way which depends on the definition of  $\Delta$ . The simplest example is given by  $\Delta(f) = f$  which will clearly be well typed in our system for an arbitrary type  $A$ . This is the usual case for functional programming languages, and we hope to demonstrate how this can be remedied in a future paper to produce a constructive type theory.

$$\begin{array}{c}
\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \\
\frac{\Gamma \vdash r : A \rightarrow B \quad \Gamma \vdash s : A}{\Gamma \vdash r s : B} \\
\frac{\Gamma \vdash t : \forall A.T}{\Gamma \vdash t B : A[X := B]} \\
\frac{}{\Gamma \vdash () : 1} \\
\frac{\Gamma \vdash t : T}{\Gamma \vdash \text{inl}(t) : (T + S)} \\
\frac{U = \mu X.T \quad \Gamma \vdash t : T}{\Gamma \vdash \text{unfold}(t, U) : [X := U]T} \quad X \notin \mathbf{FV}(\Gamma) \\
\frac{\Gamma \vdash e : T + S \quad \Gamma, x : T \vdash t : U \quad \Gamma, y : S \vdash s : U}{\Gamma \vdash (\text{case } e \text{ of } \text{inl}(x) \Rightarrow t \mid \text{inr}(y) \Rightarrow s) : U} \\
\frac{\Gamma \vdash s : T \times S \quad \Gamma, x : T, y : S \vdash t : U}{\Gamma \vdash (\text{split } s \text{ as } x_1, x_2 \text{ in } t) : U} \\
\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash (\lambda x : A.t) : A \rightarrow B} \\
\frac{\Gamma \vdash t : A}{\Gamma \vdash \Lambda X.t : \forall X.A} \quad X \notin \mathbf{FV}(\Gamma) \\
\frac{\Gamma, f : A \vdash \Delta(f) : A}{\Gamma \vdash f : A} \\
\frac{\Gamma \vdash r : A \quad \Gamma \vdash s : B}{\Gamma \vdash (r, s) : A \times B} \\
\frac{\Gamma \vdash t : S}{\Gamma \vdash \text{inr}(t) : (T + S)} \\
\frac{U = \mu X.T \quad \Gamma \vdash t : [X := U]T}{\Gamma \vdash \text{fold}(t, U) : U}
\end{array}$$

Because of the strong normalisation property of System  $F$ , we can carefully construct our evaluation relation in such a way that we can separate out potential non-termination due to function constant unfolding. This ensures that we never encounter an infinite number of *transient* reductions [10].

$$\begin{array}{c}
(\lambda x : T.t) s \rightsquigarrow_{\beta} t[x := s] \qquad (\Lambda A : \kappa.t) T \rightsquigarrow_{\tau} t[A := T] \\
\frac{t \rightsquigarrow t'}{t s \rightsquigarrow_{\alpha} t' s} \qquad \text{unfold}(\text{fold}(s, T), T) \rightsquigarrow_{\mu} s \\
\frac{f \triangleq e \in \Delta}{f \rightsquigarrow_{\delta} e} \qquad \text{split}(r, s) \text{ as } x, y \text{ in } e \rightsquigarrow_{\pi} e[x := r, y := s] \\
\text{case } \text{inr}(t) \text{ of } \text{inl}(x) \Rightarrow e_1 \mid \text{inr}(y) \Rightarrow e_2 \rightsquigarrow_{\iota} e_2[y := t] \\
\text{case } \text{inl}(t) \text{ of } \text{inl}(x) \Rightarrow e_1 \mid \text{inr}(y) \Rightarrow e_2 \rightsquigarrow_{\iota} e_1[x := t] \\
\rightsquigarrow \equiv \rightsquigarrow_{\beta} \cup \rightsquigarrow_{\alpha} \cup \rightsquigarrow_{\tau} \cup \rightsquigarrow_{\iota} \cup \rightsquigarrow_{\pi}
\end{array}$$

Notice the  $\rightsquigarrow$  relation does not make use of function unfolding. The reason for this is that the  $\rightsquigarrow_{\delta}$  relation may not reduce. This is easy to see from the

$$\begin{array}{ll}
t \rightsquigarrow^+ t & \text{iff } t \rightsquigarrow t' \vee t \rightsquigarrow t'' \wedge t'' \rightsquigarrow^+ t \\
t \rightsquigarrow^* t' & \text{iff } t = t' \wedge t \rightsquigarrow^+ t \\
t \not\rightsquigarrow & \text{iff } \neg \exists s. t \rightsquigarrow s \\
t \downarrow h & \text{iff } t \rightsquigarrow^* h \wedge h \not\rightsquigarrow
\end{array}$$

program  $f \triangleq f$ , which yields no normal form under evaluation. By omitting  $\rightsquigarrow_\delta$  we ensure that we can always obtain a kind of normal form.

Here  $\rightsquigarrow^+$ , the transitive closure of  $\rightsquigarrow$  is taken to be the least fixed point of the recursive equation. The transitive reflexive closure  $\rightsquigarrow^*$  is defined in terms of the transitive closure.

Associated with each term of active type is a some set of *experiments* which can be applied in a type directed fashion. These experiments are listed in Table ??.

$$\begin{array}{l}
E ::= - x : A \mid - A \mid (\text{case } - \text{ of } \text{inl}(x) \Rightarrow e_1 \mid \text{inr}(y) \Rightarrow e_2) \\
\mid \text{split } - \text{ as } x, y \text{ in } e
\end{array}$$

Rather than quantifying over infinite contexts, we can use these experiments  $C[-]$  in a type directed way to assemble arbitrary contexts by  $C_0[\dots C_n[-]]$ . One can see by inspection that arbitrary type-correct contexts can be construct in this manner.

We will need one further property of contexts and reduction, namely that a term either does not reduce, or it has a unique reduct.

**Lemma 1 (Unique Decomposition Property).** *For any term  $t$  which reduces under the  $\rightsquigarrow_\delta$  relation, there is a unique reduct such that we can rewrite  $t = C[f]$ ,  $t = C[\text{case } x \xrightarrow{t} \text{ of } \text{inl}(x_1) \Rightarrow e_1 \mid \text{inr}(x_2) \Rightarrow e_2]$  or  $t = C[\text{split } x \xrightarrow{t} \text{ as } x_1, x_2 \text{ in } e]$*

*Proof.* This lemma follows by induction on the structure of terms and the property that evaluation for System  $F$  restricted to  $\rightsquigarrow$  will terminate. Since  $\lambda$  and  $\Lambda$  will not reduce, we need only deal with the two forms of application. There can be no applications of  $\lambda$  or  $\Lambda$  since these would have reduced under the evaluation relation. This means only function constants, cases or splits are left. The uniqueness is derived from the fact that the evaluation relation always chooses the left branch of an application for reduction, leading to a unique term for reduction, which may be a function constant  $f$ , a variable of type  $A + B$  or a variable of type  $A \times B$ . Hence we may write this term  $t = C[f]$ ,  $t = C[\text{case } x \xrightarrow{t} \text{ of } \text{inl}(x_1) \Rightarrow e_1 \mid \text{inr}(x_2) \Rightarrow e_2]$  or  $t = C[\text{split } x \xrightarrow{t} \text{ as } x_1, x_2 \text{ in } e]$

### 3 Transition System

A transition system is a structure which consists of a collection of states and actions and a relation which associates states via some action. Formally such a system is described by a tuple as follows:

$$\mathcal{T} = (\mathcal{S}, \mathcal{A}, \delta : \mathcal{S} \otimes \mathcal{A} \otimes \mathcal{S})$$

Where  $\mathcal{S}$  is a set of states,  $\mathcal{A}$  is a set of actions and  $\delta(s, \alpha, s')$  is a relation representing potential transitions from a state  $s$  to some state  $s'$  by way of some action  $\alpha$ .

For our purposes, sets of states will be represented by programs, and transitions will be generated according to type directed experiments.

Transition resulting from delta-reductions are not *observable*. They are effectively *transient* reductions. We will however explicitly notate them for book keeping, in order to help us reason about termination behaviour.

We will define the function  $\Xi : A_F \rightarrow \mathcal{T}$  as a function taking terms in our language into transition systems. We use the maps  $\rho$  and  $\psi$  to associate variables with terms. We assume that variable names are renamed to avoid capturing as is the case with application of lambda terms. The function is defined explicitly as follows:

$$\begin{aligned}
\Xi[\lambda x : A.t]_\rho & ::= \lambda x : A.t \xrightarrow{x:A} \Xi[t]_\rho \\
\Xi[\Lambda A.t]_\rho & ::= \Lambda A.t \xrightarrow{A} \Xi[t]_\rho \\
\Xi[1]_\rho & ::= 1 \\
\Xi[(r, s)]_\rho & ::= (r, s) \xrightarrow{\text{fst}} \Xi[r]_\rho, (r, s) \xrightarrow{\text{snd}} \Xi[s]_\rho \\
\Xi[\text{inl}(t)]_\rho & ::= \text{inl}(t) \xrightarrow{\text{inl}} \Xi[t]_\rho \\
\Xi[\text{inr}(t)]_\rho & ::= \text{inr}(t) \xrightarrow{\text{inr}} \Xi[t]_\rho \\
\Xi[\text{fold}(t, A)]_\rho & ::= \text{fold}(t, A) \xrightarrow{\text{fold}} \Xi[t]_\rho \\
\Xi[\text{unfold}((, t)A)]_\rho & ::= \text{unfold}(t, A) \xrightarrow{\text{unfold}} \Xi[t]_\rho \\
\Xi[C[f]]_\rho & ::= C[f] \xrightarrow{\delta f} \Xi[C[\Delta(f)]]
\end{aligned}$$

$$\begin{aligned}
\Xi[C[\text{case } x \vec{t} \text{ of } \text{inl}(x_1) \Rightarrow e_1 \mid \text{inr}(x_2) \Rightarrow e_2]] & ::= \\
C[\text{case } x \vec{t} \text{ of } \text{inl}(x_1) \Rightarrow e_1 \mid \text{inr}(x_2) \Rightarrow e_2] & \xrightarrow{x \vec{t} := \text{inl}(x_1)} C[e_1[x \vec{t} := \text{inl}(x_1)]], \\
C[\text{case } x \vec{t} \text{ of } \text{inl}(x_1) \Rightarrow e_1 \mid \text{inr}(x_2) \Rightarrow e_2] & \xrightarrow{x \vec{t} := \text{inr}(x_2)} C[e_2[x \vec{t} := \text{inr}(x_2)]] \\
\Xi[C[\text{split } x \vec{t} \text{ as } x_1, x_2 \text{ in } e]] & ::= C[\text{split } x \vec{t} \text{ as } x_1, x_2 \text{ in } e] \xrightarrow{x \vec{t} := (x_1, x_2)} e
\end{aligned}$$

Once transition systems are given for terms, we can proceed to define bisimilarity. Bisimilarity is a coinductive equality relation. If two terms are bisimilar, we should not be able to distinguish them by any number of *experiments* on the terms. This is effectively identical to contextual equivalence, but allows us to look directly at the transition systems to establish bisimilarity, rather than having to cope with quantification over contexts. The technical machinery is consequently less complex.

Bisimilarity is defined as a relation between two transition systems with the following definition.

**Definition 1 (Bisimilarity).** *A term  $s$  is bisimilar to a term  $t$ , written  $s \sim t$ , if the following two conditions hold:*

- $s \sim t \rightarrow \forall (s, \alpha, s') \in \delta \rightarrow \exists (t \xrightarrow{\alpha} t') \in \delta \wedge s' \sim t'$
- $s \sim t \rightarrow \forall (t, \alpha, t') \in \delta \rightarrow \exists (s, \alpha, s') \in \delta \wedge s' \sim t'$

*Or, if the terms are syntactically identical.*

In order to make use of transition systems for our theory however, we will also need to make use of a notion of composition. This will allow us to generalise transition systems and to make them parametric. The basic idea is to make explicit a notion of composition of transition systems such that the following theorem holds. This notion of composition is similar to the idea of composition for normalisation by partial evaluation [1].

**Definition 2 (Composition).** *Composition of trees is achieved by replacement of states in the transition system or replacement of labels on transitions.*

$$\begin{aligned} (r \xrightarrow{x:T} \Xi[s]_\rho) \cdot \Xi[t]_\psi &= \Xi[s]_{\rho \cup (x,t)} \\ (r \xrightarrow{A::\kappa} \Xi[s]) \cdot \Xi[T]_\psi &= \Xi[s]_{\rho \cup (A,T)} \\ (C[f] \xrightarrow{\delta f} \Xi[C[\Delta(f)]]) \cdot \Xi[r] &= C[f] \xrightarrow{\delta f} (\Xi[C[\Delta(f)]] \cdot \Xi[r]) \end{aligned}$$

The composition operator as defined here can potentially fail to associate further transitions for function unfolding. This is ok however, since non-terminating behaviour is described as a transition system with no available transitions.

**Theorem 1 (Composition).**  $\Xi[t]_\rho \cdot \Xi[s]_\rho$  is defined whenever  $\Xi[t s]$  is defined and enjoys the property that  $\Xi[t s] = \Xi[t] \cdot \Xi[s]$ .

The proof of this theorem follows by construction since we use variable substitution in essentially the same way.

We would now like to provide a reification of transition systems back into terms. However, in general these terms may be of infinite size. In order to ensure finite terms we will need to make use of *folding* and *generalisation* and hence, we must now describe the supercompilation algorithm.

## 4 Supercompilation

Supercompilation is a program transformation framework first developed by Turchin. Sørensen and Glück defined positive supercompilation [12], which is an algorithm for program transformation. We will present a system modeled on the positive supercompilation algorithm extended to deal explicitly with types in System  $F$ . We then show the correctness of this algorithm using bisimilarity.

Supercompilation uses the concepts of *driving*, *generalisation* and *folding*. *Driving* is the production of a *process tree* by way of normal order evaluation. For those familiar with supercompilation, the above descriptions of transition systems will look very familiar.

The difference between the two is largely in the explicit labeling of transitions, allowing bisimilarity to be defined, and the use of folding. To simplify the presentation we will not use the traditional formulation of driving, but we will proceed to define supercompilation directly in terms of transition systems.

Since process trees are potentially infinite, we will require some mechanism of creating a finitary representation. *Folding* involves describing a transition

system in terms of states prior in the transition system which are  $\alpha$ -equivalent, that is, equivalent modulo variable renaming. Instead of representing the entire potentially infinite unfolding, we can now *point back* to a prior state set in the transition system.

**Theorem 2 (Folding).** *If a node in the transition system  $t$  is a renaming of former term  $s$ , such that  $t\sigma = s$ , then  $\Xi[t]_\rho \Xi[s]_{\rho\cup\sigma}$ .*

In order to ensure that we can find folds, we will need the composition law provided previously and a notion of generalisation. Generalisation can be considered the dual of unification [9]. In general, the least general generalisation of terms is undecidable, so we must make do with an over-generalisation, of some form.

**Definition 3 (Generalisation).** *A generalisation operator  $r \sqcap s = (t, \theta_1, \theta_2)$  is defined such that  $t\theta_1 = r$  and  $t\theta_2 = s$ .*

Not any generalisation procedure which meets the above criterion will be sufficient for our purposes. Higher order generalisation does not necessarily lead to an upper bound as arbitrarily large terms can result from generalisation [6]. Some additional requirement must be made to ensure upper bounds, for instance, requiring that terms are strictly syntactically smaller.

After generalisation of a term, we make use of the composition property to continue driving on the generalised term. This step is called abstraction.

**Definition 4 (Abstraction).** *From a generalisation  $r \sqcap s = (t, \theta_1, \theta_2)$  we can reproduce the transition system by the composition property:*

$$Xi[s]_\rho = Xi[\lambda x_1 \dots \lambda x_n.t] \cdot \Xi[\theta_2(x_1)] \dots Xi[\theta_2(x_n)]_\rho$$

*Provided  $x_i$  are fresh, and not in  $\rho$ .*

Now, to control the process of generating the process tree, we need to use the composition property and some relation that ensures we can find a finite representation of our potentially infinite transition system. In order to do this, we will make use of the homeomorphic embedding [4]. The homomorphic embedding is defined in Table ???. It is a well quasi-order and ensures that there are no infinite sequences of terms which can not be ordered. Once we encounter a term which is *smaller* we will cease unfolding.

Positive supercompilation can now be described as follows.

**Definition 5 (Positive Supercompilation).** *The positive supercompilation of a term  $t$  can be produced by lazily producing the transition system  $\Xi[t]$ . When a term  $s$  is encountered which is a homeomorphic embedding of a former term, generalisation is applied and we abstract to:  $\Xi[t'] \cdot \Xi[\theta_1(x_1)] \dots \Xi[\theta_2(x_n)]$  and continue the algorithm on the strictly smaller  $\Xi[t]$ . If a term is encountered which is a renaming of a former term, we note the substitution and terminate.*

## 5 Reification

Now that we have a suitable definition of bisimulation, which captures the notion of even infinite program behaviours being identical, we can give a definition for the reification of a term. This reflection back into terms is usually called program extraction in the meta-computation community.

We define the function  $K(\tau)$  to return the set of transition systems starting at child nodes.

**Definition 6 (Reification).** *Reification is defined on the structure of process trees in the following way.*

$$\text{if } \tau\sigma \in K(\tau) \text{ then } \Delta = (f, \Pi[\tau]_{\Delta})$$

$$\Pi[r \xrightarrow{x:T} \tau]_{\Delta} = \lambda x : T. \Pi[\tau]_{\Delta}$$

$$\Pi[r \xrightarrow{A} \tau]_{\Delta} = \Lambda A. \Pi[\tau]_{\Delta}$$

$$\begin{aligned} \Pi[r \xrightarrow{x \vec{t} := \text{inl}(x_1)} \tau_1, \\ r \xrightarrow{y \vec{t} := \text{inr}(x_2)} \tau_2]_{\Delta} = \text{case } (x \vec{t}) \text{ of } \text{inl}(x) \Rightarrow \Pi[\tau_1]_{\Delta} \mid \text{inr}(y) \Rightarrow \Pi[\tau_2]_{\Delta} \end{aligned}$$

$$\Pi[r \xrightarrow{x \vec{t} := (x_1, x_2)} \tau]_{\Delta} = \text{split } (x \vec{t}) \text{ as } x_1, x_2 \text{ in } \Pi[\tau]_{\Delta}$$

$$\Pi[r \xrightarrow{\text{fst}} \tau, r \xrightarrow{\text{snd}} \psi] = (\Pi[\tau]_{\Delta}, \Pi[\psi]_{\Delta})$$

$$\Pi[r \xrightarrow{\kappa} s] = \kappa(s)$$

Where  $\kappa \in \{\text{inl}, \text{inr}, \text{fold}, \text{unfold}\}$

**Theorem 3 (Reification).** *The reification function  $\Pi$  of a transition system  $\tau$  associates a term and function constant relation  $\Delta$  with the transition system such that the following holds:*

$$\Xi[\Pi[\Xi[t]]] \sim \Xi[t]$$

This follows by construction by the definition of  $\sim$ ,  $\Pi$  and  $\Xi$ .

## 6 Example

The following program which represents the double append problem is by now well known [10]. The following however is slightly different than former presentations in that the types are explicitly represented, and the semantics are intended to be captured by the transition labels.

$$\begin{aligned}
List &= \Lambda A. \mu Y. 1 + (A \times Y) \\
\Delta &= \{ \\
& (app, \\
& \Lambda A. \lambda xs : List A. \lambda ys : List A. \\
& \quad \text{case unfold}(xs, 1 + A \times Y) \text{ of} \\
& \quad \quad \text{inl}(u) \Rightarrow ys \\
& \quad \quad | \text{inr}(p) \Rightarrow \text{fold}(\text{inr}(\text{split } p \text{ as } x, xs' \text{ in } (x, app \ xs' \ ys)), Y) \\
& ) \\
& \}
\end{aligned}$$

Now, we wish to find the map  $\Delta$  and term for  $\rho[\Xi[t]]$  where  $t = app \ A \ (app \ A \ xs \ ys) \ zs$ .  $\Xi[t]$  This yields the transition system given in Table 1, with the following program yielded by  $\rho[\Xi[t]]$ .

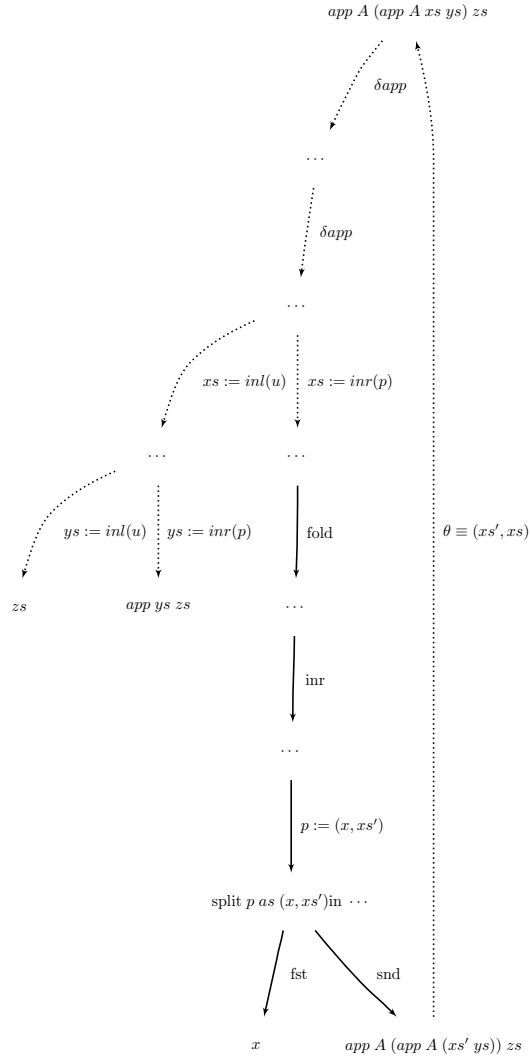
$$\begin{aligned}
t &= (appapp \ xs \ ys \ zs) \ \Delta = \{ \\
& (appapp, \\
& \Lambda A. \lambda xs : List A. \lambda ys : List A. \lambda zs : List A. \\
& \quad \text{case unfold}(xs, 1 + A \times Y) \text{ of} \\
& \quad \quad \text{inl}(u) \Rightarrow app \ ys \ zs \\
& \quad \quad | \text{inr}(p) \Rightarrow \text{case unfold}(ys, 1 + A \times Y) \text{ of} \\
& \quad \quad \quad \text{inl}(u) \Rightarrow zs \\
& \quad \quad \quad | \text{inr}(p) \Rightarrow \text{fold}(\text{inr}(\text{split } p \text{ as } x, xs' \text{ in } (x, appapp \ xs' \ ys \ zs)), Y) \\
& ) \\
& \Lambda A. \lambda xs : List A. \lambda ys : List A. \\
& \quad \text{case unfold}(xs, 1 + A \times Y) \text{ of} \\
& \quad \quad \text{inl}(u) \Rightarrow ys \\
& \quad \quad | \text{inr}(p) \Rightarrow \text{fold}(\text{inr}(\text{split } p \text{ as } x, xs' \text{ in } (x, app \ xs' \ ys)), Y) \\
& ) \\
& \}
\end{aligned}$$

Here we can see that  $\Xi[\rho[\Xi[t]]] \sim \Xi[t]$  by inspection.

## 7 Conclusion and Related Work

Normalisation by evaluation for a simple type theory is presented in [2]. A similar system defined for System F is given by Abel in [1]. Our approach differs in that we introduce a (potentially unsound) type theory based on System  $F$  which uses transition systems as the semantic domain. Our system does not produce true normal forms as these NbE systems do, but is schematically quite similar.

Supercompilation was first described by Turchin [14]. The system we use here is modeled on a description of positive supercompilation given by Sørensen and Glück [12]. Instead of using the traditional process tree or partial process tree, we present transition systems as a semantic domain. This serves much the same purpose as a process tree, however the presentation varies slightly such that it provides us with a direct means of showing equivalence in the semantic domain. This is used to motivate the notion of our reflection operator.



**Table 1.** Double Append

In future work we hope to describe conditions which ensure that the type system is sound with respect to the function  $\Delta$ . In addition it would be useful to extend the system to System  $F_\omega$  to bring the system closer to being of direct use for the Haskell Core which uses a variant of System  $F_\omega$  [13]. We would also like to include  $\eta$  normalisation in the scheme, which has not been dealt with in this work.

## References

1. Andreas Abel. Typed applicative structures and normalization by evaluation for system  $f^{mega}$ . In Erich Grädel and Reinhard Kahle, editors, *CSL*, volume 5771 of *Lecture Notes in Computer Science*, pages 40–54. Springer, 2009.
2. T. Altenkirch, P. Dybjer, M. Hofmannz, and P. Scott. Normalization by evaluation for typed lambda calculus with coproducts. In *LICS '01: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, page 303, Washington, DC, USA, 2001. IEEE Computer Society.
3. Rod M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.
4. Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (B)*, pages 243–320. 1990.
5. Andrew D. Gordon. Bisimilarity as a theory of functional programming. *Theor. Comput. Sci.*, 228(1-2):5–47, 1999.
6. Jianguo Lu, Masateru Harao, and Masami Hagiya. Higher order generalization. In *JELIA '98: Proceedings of the European Workshop on Logics in Artificial Intelligence*, pages 368–381, London, UK, 1998. Springer-Verlag.
7. Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.
8. Y. Onoue, Z. Hu, H. Iwasaki, and M Takeichi. A calculational fusion system  $hylo$ . In *Algorithmic Languages and Calculi*, pages 76–106. Chapman and Hall, 1997.
9. G.D. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970.
10. Morten H. Sørensen and Robert Glück. Introduction to supercompilation. In *Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School*, June 1998.
11. Morten Heine Sørensen. Turchin’s Supercompiler Revisited. Master’s thesis, Department of Computer Science, University of Copenhagen, 1994. DIKU-rapport 94/17.
12. Morten Heine Sørensen, Robert Glück, and Neil D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
13. Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System  $f$  with type equality coercions. In *TLDI '07: Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 53–66, New York, NY, USA, 2007. ACM.
14. Valentin F. Turchin. *The Algorithm of Generalization in the Supercompiler*, pages 531–548. Elsevier Science Publishers B.V. (North-Holland), 1988.
15. V.F. Turchin. The Use of Metasystem Transition in Theorem Proving and Program Optimization. *Lecture Notes in Computer Science*, 85:645–657, 1980.
16. P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Theoretical Computer Science*, 73:231–248, 1990.