Supercompilation and Normalisation By Evaluation

Gavin E. Mendel-Gleason and Geoff Hamilton

Lero@DCU
School of Computing
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 correspondence 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_ω. Supercompilation is a program transformation technique which performs a super-set of the simplifications performed by partial evaluation. The focus of this paper is to formalise the relationship between supercompilation and normalisation by evaluation for System F with recursive types and terms.

1 Introduction

Partial evaluation has arisen in two rather distinct settings. The first setting is in practical attempts to improve program performance. The second is the use of evaluation to produce normal forms for proofs in a Curry Howard setting [20].

The use of a term language with recursion, however, puts fundamental limits on the practical use of evaluation as a tool for normalisation or optimisation. Due to this fact the program transformation community has developed a number of tools for improving the performance of programs, including deforestation [27], fusion [14] and supercompilation [18].

Supercompilation [24] is a program transformation which performs a super-set of the optimisations performed by fusion and deforestation. Supercompilation is a complex program transformation making using 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.

When using program transformations for languages which are not strongly normalising, it is important to ensure that the transformation preservers the meaning of the program. Bisimilarity, a technique developed by Milner, [13] was used by Gordon [7] 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 allows us to define a bisimilarity relation on recursive terms in System F which demonstrates behavioural equivalence. We use 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 are formalised in such a way as to demonstrate the connection with NbE.

2 Normalisation By Evaluation

NbE makes use of two functions: a semantic interpretation function $f : Syntax \rightarrow Semantics$, and a reification function $g : Semantics \rightarrow Syntax$. The idea behind the technique is that we can arrive at a normal form for a term t (a unique syntactic description) by transforming into the semantic domain $f(t)$ and then reifying as a term $t' = g(f(t))$. This normalisation corresponds with eliminations of *cuts* from the type tree. We will see this in more detail later.

In our presentation, the semantic interpretation function is the supercompilation algorithm, which presents a transition system as the semantic representative. The reason for this is that labelled transition systems serve as a compact representation of potential program traces. When we want to establish bisimilarity, we need a formalism in which to show that we have an observable trace equivalence between programs. This idea was expressed by Turchin in [22]. This trace equivalence can be established between two systems using a bisimulation of transition systems.

Reification is performed by *program extraction* which provides us, again, with a term in our original source language.

While the techniques given here are very similar to ones used in NbE, it must be stressed that positive supercompilation will not provide unique normal forms for arbitrary terms. While it provides normal forms for any finitary expression without function symbols, it can not provide a *unique* finite representation for all transition systems. A simple proof based on the Full-Employment theorem suffices to show that any attempt to do so, except for sub-Turing complete languages will fail.

Theorem 1. *There is no normal form, giving syntactic equivalence modulo α-renaming and function symbol renaming, for arbitrary terms t for a Turing complete functional programming language.*

Proof. Assume a normalisation function f. We can apply this function to a program Ω, known not to terminate, to obtain a canonical term $f(\Omega) = \Omega_c$. We may now test any term t for halting by applying f and comparing to Ω_c syntactically, violating the Halting theorem. □

So if we can not obtain normal forms, the question naturally arises, why should we view more general program transformation through a similar lens to the one given by NbE? One answer is that in some instances the technique can be used for deciding the equality of terms as in [10] [11]. The fragment of applicability may in fact be quite large and could include languages with infinite transition systems. In addition, the use of transition systems for the semantic domain means that we can use bisimulation equivalence of transition systems as a means by which to justify the substitution of programs generally enabling us to use it as a general tool for showing semantic equivalence of program transformations.

3 Language

The language we present is a functional programming language, which we will call Λ_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**.

$\mathbf{Fun} \ni f, g$		Function Symbols
$\mathbf{Ty} \ni A, B, C$	$::= 1 \mid X \mid A \to B \mid \forall X.A \mid A + B \mid A \times B$	Types
	$\mid \nu X.A$	
$\mathbf{Tr} \ni r, s, t$	$::= x \mid f \mid () \mid \lambda x : A.t \mid \Lambda X.t \mid r\ s \mid r\ A$	Terms
	$\mid \mathrm{inl}(t) \mid \mathrm{inr}(t) \mid (t, s)$	
	$\mid \mathrm{in}(t, A) \mid \mathrm{out}(t, A)$	
	$\mid \mathrm{case}\ r\ \mathrm{of}\ \mathrm{inl}(x_1) \Rightarrow s\ ;\ \mathrm{inr}(x_2) \Rightarrow t$	
	$\mid \mathrm{split}\ r\ \mathrm{as}\ x_1, x_2\ \mathrm{in}\ s$	
$\mathbf{Ctx} \ni \Gamma$	$::= \cdot \mid \Gamma, X \mid \Gamma, x : A$	Contexts

We will describe *substitutions* using the map σ 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 simplify our presentation, we will also need to introduce recursive terms. This change is the point of departure between this work and standard presentations of NbE and our framework for supercompilation.

Recursive terms will be represented using function constants. Function constants will be drawn from a set **F**. We will couple our terms with a function Δ which associates a function constant f with a term e, $\Delta(f) = e$, where e may itself contain any function constants in the domain of Δ.

The use of Δ 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 Δ. 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 unsoundness can sometimes be remedied to produce a constructive type theory in a future paper.

For a term t with type T in a context Γ we will write $\Gamma \vdash t : T$. The type derivation is given by the following rules:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash (\lambda x : A.t) : A \to B}$$

$$\frac{\Gamma \vdash r : A \to B \qquad \Gamma \vdash s : A}{\Gamma \vdash r\ s : B} \qquad \frac{\Gamma, X \vdash t : A}{\Gamma \vdash \Lambda X.t : \forall X.A}\ X \notin \mathbf{FV}(\Gamma)$$

$$\frac{\Gamma \vdash t : \forall A.T}{\Gamma \vdash t\ B : A[X := B]} \qquad \frac{\Gamma, f : A \vdash \Delta(f) : A}{\Gamma \vdash f : A}$$

$$\frac{}{\Gamma \vdash () : 1} \qquad \frac{\Gamma \vdash r : A \qquad \Gamma \vdash s : B}{\Gamma \vdash (r, s) : A \times B}$$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \mathrm{inl}(t) : (T + S)} \qquad \frac{\Gamma \vdash t : S}{\Gamma \vdash \mathrm{inr}(t) : (T + S)}$$

$$\frac{U = \nu X.T \qquad \Gamma \vdash t : T}{\Gamma \vdash \mathrm{out}(t, U) : T[X := U]}\ X \notin \mathbf{FV}(\Gamma) \qquad \frac{U = \nu X.T \qquad \Gamma \vdash t : T[X := U]}{\Gamma \vdash \mathrm{in}(t, U) : U}$$

$$\frac{\Gamma \vdash e : T + S \qquad \Gamma, x : T \vdash t : U \qquad \Gamma, y : S \vdash s : U}{\Gamma \vdash (\mathrm{case}\ e\ \mathrm{of}\ \mathrm{inl}(x) \Rightarrow t\ ;\ \mathrm{inr}(y) \Rightarrow s) : U}$$

$$\frac{\Gamma \vdash s : T \times S \qquad \Gamma, x : T, y : S \vdash t : U}{\Gamma \vdash (\mathrm{split}\ s\ \mathrm{as}\ x_1, x_2\ \mathrm{in}\ t) : U}$$

System F without recursive types is strongly normalising, however due to the inclusion of infinite types via the $\nu X.\phi(X)$ type constructor, we can lose the strong normalisation property, even in the absence of function constant unfolding if our types are not restricted [26]. This can be seen by the simple example of the data type $D := \nu X.X \to X$ which. A concrete representative of this type which does not terminate, despite being well typed and having no function constants, is given by the term:

$$(\lambda f.\mathrm{out}(f\ \mathrm{in}(f, D), D))(\mathrm{in}((\lambda f.\mathrm{out}(f\ \mathrm{in}(f, D), D)), D))$$

We can, however, recover the normalisation property by imposing a positivity restriction on types [6].

With this restriction we can be assured that never encounter an infinite number of *transient* reductions [17], where *transient* reductions are those which do not result in a transition. This effectively allows us to ensure that all infinite

behaviour will be present in the graph only, allowing us to carefully segregate out non-termination by other means.

$$(\lambda x : T.t)\ s \rightsquigarrow_\beta t[x := s] \qquad (\Lambda A : \kappa.t)\ T \rightsquigarrow_\tau t[A := T]$$

$$\frac{t \rightsquigarrow s}{\text{split } t\ \cdots \rightsquigarrow_\alpha \text{ split } s\ \cdots} \qquad \frac{t \rightsquigarrow s}{\text{case } t\ \cdots \rightsquigarrow_\alpha \text{ case } s\ \cdots}$$

$$\frac{t \rightsquigarrow t'}{t\ s \rightsquigarrow_\alpha t'\ s} \qquad \text{out}(\text{in}(s, T), T) \rightsquigarrow_\nu s$$

$$\frac{f \triangleq e \in \Delta}{f \rightsquigarrow_\delta e} \qquad \text{split } (r, s) \text{ as } x, y \text{ in } t \rightsquigarrow_\pi t[x := r, y := s]$$

$$\text{case inr}(t) \text{ of inl}(x) \Rightarrow r\ ;\ \text{inr}(y) \Rightarrow s \rightsquigarrow_\iota s[y := t]$$

$$\text{case inl}(t) \text{ of inl}(x) \Rightarrow r\ ;\ \text{inr}(y) \Rightarrow s \rightsquigarrow_\iota r[x := t]$$

$$\rightsquigarrow \equiv \rightsquigarrow_\beta \cup \rightsquigarrow_\alpha \cup \rightsquigarrow_\tau \cup \rightsquigarrow_\iota \cup \rightsquigarrow_\pi \cup \rightsquigarrow_\nu$$

We define a reduction relation which will allow us to simplify our proofs of a given type using a relation $\rightsquigarrow$. Notice the $\rightsquigarrow$ relation does not make use of function unfolding. The reason for this is that the $\rightsquigarrow_\delta$ relation may not reduce finitely, as with the example given previously $\Delta(f) = f$. By omitting $\rightsquigarrow_\delta$ we ensure that we can always obtain a head normal form which we will describe more completely later.

$t \rightsquigarrow^+ t'$	iff	$t \rightsquigarrow t' \vee t \rightsquigarrow t'' \wedge t'' \rightsquigarrow^+ t'$
$t \rightsquigarrow^* t'$	iff	$t = t' \vee t \rightsquigarrow^+ t'$
$t \not\Downarrow$	iff	$\neg\exists s.t \rightsquigarrow s$
$t \Downarrow h$	iff	$t \rightsquigarrow^* h$
$t \Downarrow$	::=	h where $t \Downarrow h$

Table 1: Derived Relations

In Table 1 we give some relations which are derived from the $\rightsquigarrow$ evaluation relation. 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.

Because of our careful definition of $\rightsquigarrow$, we can use the relation $\Downarrow$ to arrive at a *head normal form*. This consists of an outermost syntactic form which has is either a value or a context.

Formally, divide the grammar of our language into the following classes:

$$
\begin{array}{lll}
\mathbb{O} \ni o, p := () \mid \lambda x : A.t \mid \Lambda A.t \mid (r, s) & & \text{Observable} \\
\qquad\qquad \mid \mathrm{inl}(t) \mid \mathrm{inr}(t) \mid \mathrm{in}(t, A) & & \\
\mathbb{V} \ni v \quad := x \mid f \mid v\ t \mid v\ A & & \text{Irreducible} \\
\mathbb{E} \ni e \quad := - \mid \mathrm{out}(e, A) & & \text{Context} \\
\qquad := \mathrm{case}\ e\ \mathrm{of}\ \mathrm{inl}(x) \Rightarrow t\ ;\ \mathrm{inr}(y) \Rightarrow s & & \\
\qquad := \mathrm{split}\ e\ \mathrm{as}\ x, y\ \mathrm{in}\ s & &
\end{array}
$$

Where we view $e[t]$ as shorthand for the replacement of the privileged variable $-$ with t in e, $e[- := t]$. Using this we can derive the following decomposition lemma.

Lemma 1 (Unique Decomposition). *A term s such that $t \Downarrow s$ can be written as $e[v]$ or $s \in \mathbb{O}$.*

Proof (Proof Sketch). We proceed by induction on type derivations for a term t, starting with the empty context.

If $t \in \mathbb{O}$ then the context is empty. This follows from the fact that any type correct context for an element of $\mathbb{O}$ would lead to a reduction.

If $t \in \mathbb{V}$ then we are done.

For the case $t = \mathrm{case}\ t'\ \mathrm{of}\ \mathrm{inl}(x) \Rightarrow s\ ;\ \mathrm{inr}(y) \Rightarrow r$: By the inductive hypothesis, $t' = e[v]$ since otherwise $t' \in \mathbb{O}$ which would reduce. Therefore $t = (\mathrm{case}\ e\ \mathrm{of}\ \mathrm{inl}(x) \Rightarrow s\ ;\ \mathrm{inr}(y) \Rightarrow r)[v]$, and $(\mathrm{case}\ e\ \mathrm{of}\ \mathrm{inl}(x) \Rightarrow s\ ;\ \mathrm{inr}(y) \Rightarrow r) \in \mathbb{E}$.

The remaining cases follow similarly. □

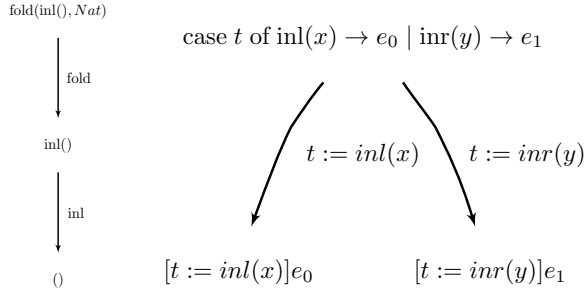
4 Transition System

For the purposes of reasoning about functional programs, and indeed the meaning of types themselves it is useful to use transition systems as a semantic domain. This approach is related to the infinite histories approach taken by Turchin [23] and is also quite close to the approach taken in process calculi such as CCS [12] and CSP [8]. It is also similar to the account given by monoidal histories [5]. The framework given here is based on the one given by Gordon in [7].

The basic idea behind the approach presented here is that terms are potentially infinite trees, and *variables* represent parametrisation with respect to an unknown transition system of appropriate type. Types themselves can be interpreted as restrictions on the form of the transition system.

An example of a value term which is represented by a finite tree is the inhabitant of the type $Nat := \nu X.1 + X$ which we can call zero, $\mathrm{in}(\mathrm{inl}(()), Nat)$, given in a Church-numeral style encoding, which is demonstrated in Fig. 1a.

In the interpretation of a term with variables, we will view the variables' semantics as being a parametrisation of transition system at the type of that variable. The parametrisation with respect to a transition system may be seen as an *external choice* non-determinism. That is, operationally, an equivalent program must have the same behaviour given the same external choices. Our various available reductions which perform substitution are an internalisation of



(a) Zero (b) Case

Fig. 1: Transition Systems

a choice which has become known. We see an example of this in the transition system in Fig. 1b.

Formally, 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:

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

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 $\alpha \in A$.

For our purposes, sets of states will be represented by programs, and transitions will be generated according to the operational behaviour of the program. Intuitively, we mean that the behaviour of a program makes *choices* for a calling program.

Transitions resulting from δ-reductions are not *observable* in the sense that they do not have any visible operational behaviour to the caller. This might seem odd in that a non-terminating program is certainly different than one which terminates. However, an infinite number of δ transitions is a failure to make a choice and is equivalent (that is, bisimilar) to a transition system with no further edges.

Formally we mean that we will not distinguish two transition systems which have transitions with arbitrarily many different transitions resulting from *delta*-reductions. This will be made explicit when we talk about *bisimiliarity* later. We will however explicitly notate them in our graphs for book keeping, in order to help us to reason about termination behaviour.

We define L_Γ as the set of terms in some typing context Γ having type derivations.

We define the function $\Xi : L_\Gamma \to \mathcal{T}$ as a function taking derivations in our language to a transition system. We will abbreviate the application of Ξ to some context Γ and a well typed term $\Gamma \vdash t : T$ as $\Xi[t]_\Gamma$.

We assume that variables are renamed to avoid capturing as is the case with application of lambda terms. In practice this can be done using a locally nameless representation.

We will overload the meaning of the $\cup$ operator in order that we can use the notation $(r, a, s) \cup \Xi[t]$ to denote the transition system $(\{t, t'\} \cup S, \{a\} \cup A, \{(r, a, s)\} \cup \delta)$ where $\Xi[t] = (S, A, \delta)$. The application of $\cup$ to two transition systems will be given by their component-wise unions. We will write the empty transition system as Ω.

We will also abuse the notation for substitution, writing $[v := t]$ for the replacement of some irreducible term v with a term t. This replacement is justified since we will retain the same termination behaviour as v is in a reduct position.

We can now generate the transition system of a term t by applying Ξ to the irreducible term $t' = t \Downarrow$. Ξ will convert terms in head normal form to transition systems.

$$\Xi[\lambda x : A.t]_\Gamma ::= (\lambda x : A.t, x : A, t \Downarrow) \cup \Xi[\Downarrow]_{\Gamma, x:T}$$

$$\Xi[\Lambda A.t]_\Gamma ::= (\Lambda A.t, A, t \Downarrow) \cup \Xi[t \Downarrow]_{\Gamma, A}$$

$$\Xi[()]_\Gamma ::= \Omega$$

$$\Xi[(r, s)]_\Gamma ::= ((r, s), \mathrm{fst}, r \Downarrow) \cup ((r, s), \mathrm{snd}, s \Downarrow) \cup \Xi[r \Downarrow]_\Gamma \cup \Xi[s \Downarrow]_\Gamma$$

$$\Xi[\mathrm{inl}(t)]_\Gamma ::= \mathrm{inl}(t) \xrightarrow{\mathrm{inl}} \Xi[t \Downarrow]_\Gamma$$

$$\Xi[\mathrm{inr}(t)]_\Gamma ::= inr(t) \xrightarrow{\mathrm{inr}} \Xi[t \Downarrow]_\Gamma$$

$$\Xi[\mathrm{in}(t, A)]_\Gamma ::= \mathrm{in}(t, A) \xrightarrow{out} \Xi[t \Downarrow]_\Gamma$$

$$\Xi[\mathrm{out}(t, A)]_\Gamma ::= (\mathrm{out}(t, A), out, t \Downarrow) \cup \Xi[t \Downarrow]_\Gamma$$

$$\Xi[e[v]]_\Gamma ::= (\{v\}, v, \emptyset) \cup \Omega$$

$$\Xi[e[f]]_\Gamma ::= (e[f], \delta f, e[\Delta(f)]) \cup \Xi[e[\Delta(f)]]_\Gamma$$

$$\begin{aligned}\Xi[e[\text{case } v \text{ of } \mathrm{inl}(x) \Rightarrow r \ ; \ \mathrm{inr}(y) \Rightarrow s]]_\Gamma ::= \\ (e[\text{case } v \text{ of } \mathrm{inl}(x) \Rightarrow r \ ; \ \mathrm{inr}(y) \Rightarrow s], (v := \mathrm{inl}(x_1)), e[r[v := \mathrm{inl}(x)]]) \cup \\ (e[\text{case } v \text{ of } \mathrm{inl}(x) \Rightarrow r \ ; \ \mathrm{inr}(y) \Rightarrow s], (v := \mathrm{inr}(x_2)), e[s[v := \mathrm{inr}(y)]]) \cup \\ \Xi[e[r[v := \mathrm{inl}(x)]]]_\Gamma \cup \Xi[e[s[v := \mathrm{inl}(y)]]]_\Gamma\end{aligned}$$

$$\begin{aligned}\Xi[e[\text{split } v \text{ as } x, y \text{ in } r]]_\Gamma ::= \\ (e[\text{split } v \text{ as } x_1, x_2 \text{ in } e], (v := (x_1, x_2)), r[v := (x_1, x_2)]) \cup \\ \Xi[r[v := (x_1, x_2)]]_\Gamma\end{aligned}$$

Notice the addition of the δ transition for a function variable f. This transition will not be directly observable, but will be used for book keeping, so that we can avoid the problem of non-termination at nodes. Now that we have a transition system for each term in the above described normal form, we can be assured of a having a (potentially infinite) transition system for every program.

We carry the context Γ through the computation because it is needed later for generalisation and abstraction. In an implementation it is convenient to perform the transformation on derivations rather than terms, such that the appropriate context is always present.

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 [7].

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:*

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

Since bisimilarity is the greatest fixed point of such a relation, we need only to produce a relation that demonstrates these properties, and it will be a subset of the bisimilarity relation.

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.*

$$\Xi[\lambda x : T.s]_\Gamma \cdot \Xi[t] ::= \Xi[s[x := t] \Downarrow]_\Gamma$$
$$\Xi[\Lambda A.s]_\Gamma \cdot T ::= \Xi[s[A := T] \Downarrow]_\Gamma$$
$$\Xi[v]_\Gamma \cdot \Xi[r]_\Gamma ::= (v\ r, v\ r, \emptyset) \cup \Omega$$
$$\begin{aligned}\Xi[e[&case\ v\ of\ inl(x) \Rightarrow r\ ;\ inr(y) \Rightarrow s]]_\Gamma \cdot \Xi[t]_\Gamma ::= \\ &(e[case\ v\ of\ inl(x) \Rightarrow r\ ;\ inr(y) \Rightarrow s]\ t, (v := inl(x)), (e[r[v := inl(x)]]\ t) \Downarrow) \cup \\ &\Xi[(e[r[v := inl(x)]]\ t) \Downarrow]_{\Gamma,x:A} \cup \\ &(e[case\ v\ of\ inl(x) \Rightarrow r\ ;\ inr(y) \Rightarrow s]\ t, (v := inr(y)), (e[s[v := inr(y)]]\ t) \Downarrow) \cup \\ &\Xi[(e[s[v := inr(y)]]\ t) \Downarrow]_{\Gamma,y:B}\end{aligned}$$
$$\begin{aligned}\Xi[e[&split\ v\ as\ x, y\ in\ s]]_\Gamma \cdot \Xi[r]_\Gamma ::= \\ &(e[split\ v\ as\ x, y\ in\ e]\ r, (v := (x, y)), (e[v := (x, y)]\ r) \Downarrow) \cup \\ &\Xi[(e[v := (x, y)]\ r) \Downarrow]_{\Gamma,x:A,y:B}\end{aligned}$$
$$\Xi[e[f]]_\Gamma \cdot \Xi[r]_\Gamma ::= (e[f], \delta f, e[\Delta(f)]) \cup (\Xi[e[\Delta(f)]]_\Gamma \cdot \Xi[r]_\Gamma)$$

Theorem 2 (Composition is Bisimilar). *$\Xi[t] \cdot \Xi[s]$ is defined whenever $\Xi[t\ s]$ is defined and enjoys the property that $\Xi[t\ s] \sim \Xi[t] \cdot \Xi[s]$.*

Proof. The proof is by construction making use of substitution and the λ and Λ cases are therefore trivial. The case and split cases also provide identical transitions in both cases. The remaining case is function constant unfolding in a context.

If we have $t = e[f]$ then $t\ s$ is $e[f]\ s$. Unfolding f in either situation must lead to either a new function constant to unfold, $t' = e[g]$ and $t'\ s$ is $e[g]\ s$ (with f not necessarily different than g). Either we eventually encounter a λ or Λ leading to a reduction, or we encounter an infinite number of function constant un-foldings in either case, which, since delta transitions are not distinguished by bisimilarity, are bisimilar.

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 find a finite representation of our transition system. Essentially this requires the production of a graph using our transition system function Ξ and composition. Practically this can be achieved using supercompilation.

5 Supercompilation

Supercompilation is a program transformation framework first developed by Turchin [25]. Sørensen, Glück and Jones defined positive supercompilation [19], 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 in the transition system which are α-equivalent, that is, equivalent modulo bound variable renaming. Instead of representing the entire potentially infinite unfolding, we can now *point back* to a prior state set in the transition system.

Definition 3 (Abstraction). *Given a type derivation $\Gamma \vdash t : T$, We can form a term, $\vdash \lambda\Gamma.s : \overrightarrow{S} \to T$ by taking each variable $(x, S) \in \Gamma$ and abstracting over t with $x : S$ and for each type variable X with $\Lambda X.t$.*

Theorem 3 (Abstraction is Bisimilar). *If a term t is a renaming of a term s, such that $t\sigma = s$ where σ are variables drawn from a typing context Γ, then $\Xi[t] \sim \Xi[\lambda\Gamma.t\ \overrightarrow{x}] \sim \Xi[\lambda\Gamma.t] \cdot \Xi[x_0] \cdot \cdots \cdot \Xi[x_n]]$ where x_i is the ith variable in $\overrightarrow{x}$.*

The proof of this theorem follows directly from the composition lemma. This will allow us to re-use elements of the transition system previously encountered.

Generalisation can be considered the dual of unification and is sometimes called anti-unification [16]. Generalisation is defined over the semi-lattice induced by the *instantiation* ordering.

Definition 4 (Instantiation Ordering). *A term $t : A$ is said to be an instance of a term $s : B$, if there is a substitution θ such that $t : A = (s : B)\theta$. We write that t is an instance of s or $t : A \preceq s : B$. Similarly, a typing context Γ is said to be an instance of a typing context Γ' if each variable is an instance, or $\Gamma \preceq \Gamma'$.*

The least general generalisation of terms for System F is undecidable [9]. In our implementation we have restricted ourselves to generalisation of higher order patterns using a method similar to the one described in [15].

It is the case that any generalisation can be used for the purposes of assisting in the creation of a finite graph, but the particulars of the generalisation will affect the final form of our transition system.

Definition 5 (Generalisation). *A generalisation operator $r \sqcup s = (t, \theta_1, \theta_2)$ is defined such that $t\theta_1 = r$ and $t\theta_2 = s$.*

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. It is typical to make use of the homeomorphic embedding [4] for this purpose. It is a well quasi-order and ensures that there are no infinite sequences of terms which can not be ordered provided that the set of function constants is finite. Any relation which

ensures that unfolding is restricted to finite sequences is sufficient. We will call this our *whistle* relation.

It is useful to have a reflection to the original syntax which reproduces the original term, modulo naming of functions and variables. This will be used to ensure that we can recover a finite transition system even if we are unable to find new folds. We will use this in our supercompilation algorithm in order to *give up* when we may no longer proceed.

Definition 6 (Identity). *$Id[t]$ is defined as the transition system $\Xi[t]_\Gamma$ where every function call in reduct position is replaced with a composition $\Xi[f]_\Gamma \cdot \Xi[s]_\Gamma$. We may use this composition to produce a finite graph for any term, since the original term is itself finite.*

Definition 7 (Reachable). *A term t is said to be reachable from s if there is a sequence of terms t_i such that $(s, a, t_0) \in \delta$ and $(t_n, a, t) \in \delta$ and $(t_i, a, t_{i+1}) \in \delta$.*

Definition 8 (Ancestor). *A term t is said to be an* ancestor *of a term s if s is reachable by delta from t.*

The general form of supercompilation can now be described as follows.

Definition 9 (Supercompilation). *The positive supercompilation of a term t can be produced by lazily producing the transition system $\Xi[t']$ where $t' = t \Downarrow$. We will write the resulting finite representation of the transition system as $S[t']$.*

When a term s is encountered which has an ancestor r which satisfies the whistle relation, we have a number of cases:

If $s \preceq r$, then we abstract $\Xi[s]$ to obtain $\Xi[r] \cdot \Xi[\theta(x_0)] \cdot \cdots \cdot \Xi[\theta(x_n)]$ where θ is the substitution that proves the instantiation and x_i are the variables in theta. We continue the algorithm on each of the $\Xi[\theta(x_0)]$.

Otherwise, generalisation is applied to s and the term r, $s \sqcup r = (e_g, \theta_1, \theta_2)$. We use abstraction to write: $\Xi[r]_\Gamma \ \Xi[\lambda\Gamma.e_g]_\Gamma \cdot \Xi[\theta_1(x_0)]_\Gamma \cdot \cdots \cdot \Xi[\theta_1(x_n)]_\Gamma$ and continue the algorithm on $\Xi[e_g]$ and each of the $\theta_1(x_i)$.

If generalisation fails, we can give up *using $Id[s]$. Otherwise we proceed on each term reachable from s by δ.*

6 Reification

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

We define the function $K(\tau)$ to return the set of transition systems starting at child nodes for some transition system τ. We will use the notation $r \xrightarrow{a} \tau$ to mean that τ is the transition system starting from $\delta(r, a)$.

Definition 10 (Reification). *Reification is defined on the structure of process trees in the following way.*

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

$$\Pi[\Xi[t] \cdot \Xi[s]] = \Pi[\Xi[t]]\ \Pi[\Xi[s]]$$

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

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

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

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

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

$$\begin{aligned}&\Pi[r \xrightarrow{\kappa} s] = \kappa(s)\\ &\textit{Where } \kappa \in \{inl, inr, in, out\}\end{aligned}$$

Theorem 4 (Reification). *The reification function Π of a transition system τ associates a term and function constant relation Δ with the transition system such that the following holds:*

$$S[\Pi[S[t]]] \sim S[t]$$

This follows by construction by the definition of $\sim$, Π and S.

7 Example

The following program which represents the double append problem is by now well known [17]. Our example, 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.

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

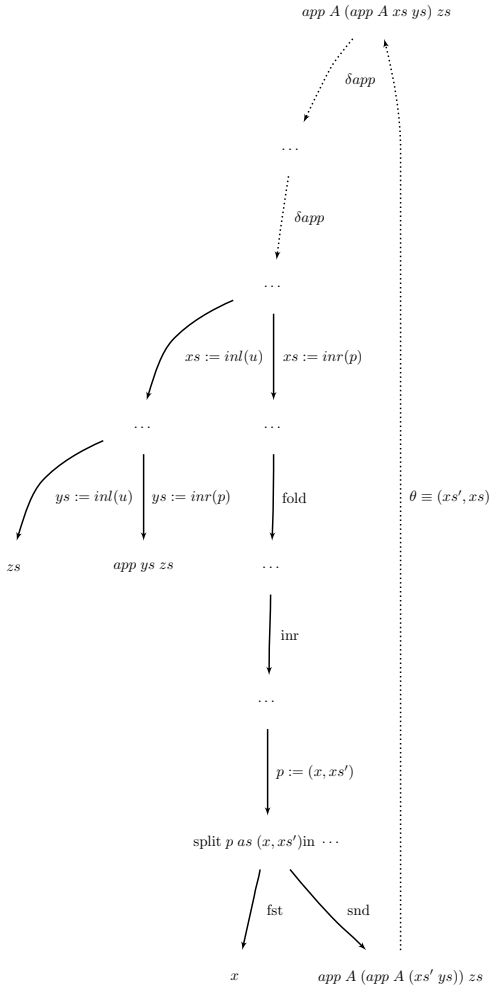


Fig. 2: Double Append

Now, we wish to find the map Δ and term for $\Pi[\Xi[t]]$ where

$$t = app\ A\ (app\ A\ xs\ ys)\ zs.\Xi[t]$$

This yields the transition system given in Fig. 2, with the following program yielded by $\Pi[\Xi[t]]$.

$t = (appapp\ xs\ ys\ zs)\ \Delta = \{$
$(appapp,$
$\Lambda A.\lambda xs : List\ A.\lambda ys : List\ A.\lambda zs : List\ A.$
  case out$(xs, List\ A)$ of
    inl$(u) \Rightarrow app\ ys\ zs$
  ; inr$(p) \Rightarrow$ case out$(ys, List\ A)$ of
      inl$(u) \Rightarrow zs$
    ; inr$(p) \Rightarrow$ in(inr(split p as x,xs' in $(x$,$appapp\ xs'\ ys\ zs))), Y)$
$),$
$(app,$
$\Lambda A.\lambda xs : List\ A.\lambda ys : List\ A.$
  case out$(xs, List\ A)$ of
    inl$(u) \Rightarrow ys$
  ; inr$(p) \Rightarrow$ in(inr(split p as x, xs' in $(x, app\ xs'\ ys))), Y)$
$)$
$\}$

Here we can see that $S[\Pi[S[t]]] \sim S[t]$ by inspection, a feature that follows from the idempotence of the composition $S \circ \Pi$.

8 Conclusion and Related Work

We have demonstrated a parallel between NbE and supercompilation for a System F with sums, products and recursive types. The intent is to clarify exactly in what way it can serve as a form of normalisation. In addition we have used bisimilarity of transition systems, which we use as the semantic domain, to show the correctness of our transformations. This simplifies the presentation, but also allows us to think more generally about term equivalence and the establishment of correct program transformations.

This work also makes a first step to including languages with polymorphic type systems directly into the supercompilation procedure while including type information. As we can see from the generalisation procedure, type information can not be ignored in the context of polymorphic types as it has a bearing on the form that generalisations will take and when generalisation can occur. If supercompilation is to be applied in the context of languages such as the Calculus of Constructions this will be even more critical.

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 [22]. Turchin is the first to recognise program configurations as representing some number of (potentially infinite) states, and process trees as being representations of potentially infinite traces. The system we use here is modeled on a description of positive supercompilation given by Sørensen, Glück and Jones [19].

The similarity of NbE and supercompilation have been described by Lisitsa and Webster [11] as well as Romanenko and Kluchnikov [10].

This work differs in spelling out the correspondence more completely. In addition it includes explicit type information. 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 in 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.

In addition, to make explicit the connections with NbE it is important that supercompilation is done on terms with type information included. This paper is a preliminary step in this direction. In developing the use of supercompilation for constructive type theories [6], this will be particularly important.

In future work we hope to describe conditions which ensure that the type system is sound with respect to the function Δ. In addition it would be useful to extend the system to System F_ω to bring it closer to being of direct use for the Haskell Core which uses a variant of System F_ω [21].

It is also conceivable that normal forms of transition systems actually *do* exist in the context of constructive type theories with infinite terms, at least since the languages are necessarily sub-Turing complete and no simple application of the Full Employment Theorem can be made. It would be of value to explore this potentiality.

In the presentation given here, iso-recursive infinite types are given using explicit constructors representing the isomorphism. This may not be strictly necessary as it is possible to encode least and greatest fixed points directly in System F using universal quantification and impredicativity [28]. Indeed, the introduction and elimination rules for sums and products can also be encoded leaving only β, τ-reduction. While, for efficiency reasons and convenience, it is useful to have these constructors and destructors represented explicitly, it would be interesting to see how an exposition without them compares.

It would also be useful to include η-normalisation and variants which include function-symbol application, into the scheme, which has not been dealt with in this work. The inclusion of η would increase the number of programs which could find a *normal form* for the purpose of deducing equality [10].

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

References

1. Andreas Abel. Typed applicative structures and normalization by evaluation for system f^{ω}. 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. N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 243–320. Elsevier, MIT Press, 1990.
5. Volker Diekert and Yves Métivier. Partial commutation and traces. pages 457–533, 1997.
6. Eduardo Giménez. Structural recursive definitions in type theory. In *ICALP '98: Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, pages 397–408, London, UK, 1998. Springer-Verlag.
7. Andrew D. Gordon. Bisimilarity as a theory of functional programming. *Theor. Comput. Sci.*, 228(1-2):5–47, 1999.
8. C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.
9. G. Huet. The undecidability of unification in third order logic. *Information and Control*, 22:257–267, 1973.
10. Ilya Klyuchnikov and Sergei Romanenko. Proving the equivalence of higher-order terms by means of supercompilation. In *Perspectives of Systems Informatics*, volume 5947, pages 193–205. Springer, 2009.
11. Alexei Lisitsa and Matt Webster. Supercompilation for equivalence testing in metamorphic computer viruses detection. In *Proceedings of First International Workshop on Metacomputation in Russia, META'2008*, pages 113–118, 2009.
12. R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.
13. Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.
14. 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.
15. Frank Pfenning. Unification and anti-unification in the calculus of constructions. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science (LICS 1991)*, pages 74–85. IEEE Computer Society Press, July 1991.
16. G.D. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970.
17. Morten H. Sørensen and Robert Glück. Introduction to supercompilation. In *Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School*, June 1998.
18. Morten Heine Sørensen. Turchin's Supercompiler Revisited. Master's thesis, Department of Computer Science, University of Copenhagen, 1994. DIKU-rapport 94/17.
19. Morten Heine Sørensen, Robert Glück, and Neil D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

20. Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism, Volume 149 (Studies in Logic and the Foundations of Mathematics)*. Elsevier Science Inc., New York, NY, USA, 2006.
21. 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.
22. Valentin F. Turchin. The algorithm of generalization in the supercompiler. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 531–548. Elsevier Science Publishers B.V. (North–Holland), 1988.
23. V.F. Turchin. The language refal–the theory of compilation and metasystem analysis. Technical Report 18, Curant Institute of Mathematics, 1980.
24. V.F. Turchin. The Use of Metasystem Transition in Theorem Proving and Program Optimization. *Lecture Notes in Computer Science*, 85:645–657, 1980.
25. V.F. Turchin. The Algorithm of Generalization in the Supercompiler. In *Proceedings of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation*, pages 531–549, 1988.
26. D. A. Turner. Total functional programming. *Journal of Universal Computer Science*, 10(7):751–768, 2004.
27. P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Theoretical Computer Science*, 73:231–248, 1990.
28. Philip Wadler. Recursive types for free!, 1990.