Development of the Productive Forces

Gavin E. Mendel-Gleason[1] and Geoff W. Hamilton[2]

[1] School of Computing, Dublin City University ggleason@computing.dcu.ie
[2] School of Computing, Dublin City University hamiltion@computing.dcu.ie

Abstract. Proofs involving infinite structures can use corecursive functions as inhabitants of a corecursive type. Admissibility of such functions in theorem provers such as Coq or Agda, requires that these functions are productive. Typically this is proved by showing satisfaction of a guardedness condition. The guardedness condition however is extremely restrictive and many programs which are in fact productive and therefore will not compromise soundness are nonetheless rejected. Supercompilation is a family of program transformations which retain program equivalence. Using supercompilation we can take programs whose productivity is suspected and transform them into programs for which guardedness is syntactically apparent.

1 Introduction

The Curry-Howard correspondence lets us take advantage of a close correspondence between programs and proofs. The idea is that inhabitation of a type is given by demonstration of a type correct program having that type. In the traditional Curry-Howard regime type inhabitation is given with terminating programs. This requirement avoids the difficulty that non-terminating programs might allow unsound propositions to be proved.

This was extended by Coquand [6] to include potentially infinite structures. The notions of co-induction and co-recursion allow infinite structures and infinite proofs to be introduced without loss of soundness.

In the Agda and Coq theorem provers, the guardedness condition is the condition checked to ensure admissibility of coinductive programs. This condition ensures that the function is *productive* in the sense that it will always produce a new constructor finitely. The guardedness condition has the advantage of only requiring a syntactic check on the form of the program and is a sufficient condition for productivity. However, it suffers from being highly restrictive and rejects many programs which *are* in fact productive. Requiring guards is a rather rigid approach to ensuring productivity.

The purpose of this paper is to suggest that theorem provers can use a notion of pre-proofs[4], proofs for which the side conditions which ensure soundness have not been demonstrated, which can be transformed into a genuine proof using equational reasoning. The main idea is that, if a program is type-correct after program transformation using equational reasoning, it was correct prior to

transformation. This was the main idea behind the paper by Danielsson et al. [7].

The approach has a number of advantages over the traditional syntactic check or ad-hoc methods of extending the guardedness condition to some larger class. The number of programs which might be accepted automatically can be increased simply by using automated program transformers. All but one of the examples given in this paper were found by way of the supercompilation program transformation algorithm. The approach also allows one to keep the chain of equational reasoning present, allowing an audit trail of correctness.

This paper is structured as follows. First we will present a number of motivating examples. We will give a bird's-eye view of supercompilation followed by a description of cyclic proof and transformations in this framework. We will give a demonstration of how one of the motivating examples can be transformed in this framework. Finally we will end with some possibilities for future development.

2 Examples

In order to motivate program transformation as a means for showing type inhabitation, we present a number of examples in the language Agda. First, we would like to work with the natural numbers which include the point at infinity. This we write as the codata declaration with the familiar constructors of the Peano numbers. We also define the analogous codata for lists, allowing infinite streams.

```
module Productive where

codata ℕ∞ : Set where
  czero : ℕ∞
  csucc : ℕ∞ → ℕ∞

codata CoList (A : Set) : Set where
  [] : CoList A
  _::_ : A → CoList A → CoList A

infixr 40 _+_
_+_ : ℕ∞ → ℕ∞ → ℕ∞
czero + y = y
(csucc x) + y = csucc (x + y)

sumlen : CoList ℕ∞ → ℕ∞
sumlen [] = czero
sumlen (x::xs) = csucc (x + (sumlen xs))
```

When attempting to enter this program into a theorem prover, such as Coq or Agda, the type checker will point out that this program does not meet the guardedness condition. This is due to the fact that there is an intermediate application (of +) which might consume any constructors which *sumlen* produces. However, the program is in fact productive and no such problem occurs in practice. This becomes evident after supercompiling the above term, where we arrive at the following transformed term:

```
mutual
  sumlen_sc : CoList ℕ∞ → ℕ∞
  sumlen_sc [ ] = czero
  sumlen_sc (x::xs) = csucc (aux x xs)
  aux : ℕ∞ → CoList ℕ∞ → ℕ∞
  aux czero xs = sumlen_sc xs
  aux (csucc x) xs = csucc (aux x xs)
```

This term will now be accepted by Agda's type checker. We see that we have basically unrolled the intermediate implication eliminations. We will look more closely at this later in Section 4.

In the above example we managed to remove some intermediate applications for two functions which were both productive. We can also find similar results however by unrolling intermediate inductive steps. The following example, given by Komendaskaya and Bertot [3], relies on this *always eventually* type behaviour. That is, behaviour that will *always* be productive, *eventually* after some finite process. Here the finite process is guaranteed by the inductive character of the natural numbers.

```
data Bool : Set where
  true : Bool
  false : Bool
data ℕ : Set where
  zero : ℕ
  succ : ℕ → ℕ
infix 5 if_then_else_
if_then_else_ : {A : Set} → Bool →
  A → A → A
if true then x else y = x
if false then x else y = y
infixr 40 _::_
codata Stream (A : Set) : Set where
  _::_ :A → Stream A → Stream A
le : ℕ → ℕ → Bool
le zero _ = true
le _ zero = false
le (succ x) (succ y) = le x y
pred : ℕ → ℕ
pred zero = zero
pred (succ x) = x
f : Stream ℕ → Stream ℕ
f (x::y::xs) = if (le x y)
  then (x::(f (y::xs)))
  else (f ((pred x)::y::xs))
```

Again we have the problem that the type checker is unable to accept f as being productive as the else clause makes an unguarded call to f though the function is productive as we structurally descend on the argument x which is inductively defined. However, after supercompilation we arrive at the following program.

$$
\begin{array}{l}
\mathit{mutual} \\
\quad f\_sc : \mathit{Stream}\ \mathbb{N} \to \mathit{Stream}\ \mathbb{N} \\
\quad f\_sc\ (\mathit{zero}{::}y \qquad {::}xs) = \mathit{zero}{::}(f\_sc\ (y{::}xs)) \\
\quad f\_sc\ (\mathit{succ}\ x{::}\mathit{zero}{::}xs) = g\ x\ xs \\
\quad f\_sc\ (\mathit{succ}\ x{::}\mathit{succ}\ y{::}xs)\ \mathit{with}\ \mathit{le}\ x\ y \\
\quad \ldots \mid \mathit{true} = (\mathit{succ}\ x){::}(f\_sc\ ((\mathit{succ}\ y){::}xs)) \\
\quad \ldots \mid \mathit{false} = h\ x\ y\ xs \\
\quad g : \mathbb{N} \to \mathit{Stream}\ \mathbb{N} \to \mathit{Stream}\ \mathbb{N} \\
\quad g\ \mathit{zero}\ xs = \mathit{zero}{::}(f\_sc\ (\mathit{zero}{::}xs)) \\
\quad g\ (\mathit{succ}\ x)\ xs = g\ x\ xs \\
\quad h : \mathbb{N} \to \mathbb{N} \to \mathit{Stream}\ \mathbb{N} \to \mathit{Stream}\ \mathbb{N} \\
\quad h\ \mathit{zero}\ y\ xs = \mathit{zero}{::}(f\_sc\ ((\mathit{succ}\ y){::}xs)) \\
\quad h\ (\mathit{succ}\ x)\ y\ xs\ \mathit{with}\ \mathit{le}\ x\ y \\
\quad \ldots \mid \mathit{true} = (\mathit{succ}\ x){::}(f\_sc\ ((\mathit{succ}\ y){::}xs)) \\
\quad \ldots \mid \mathit{false} = h\ x\ y\ xs
\end{array}
$$

Now we have a program which passes Agda's type checker. The intermediate computations have been turned into inductive loops and the coinductive function now exhibits a guarded call.

With these motivating examples in hand, we would like to look a bit at the method of program transformation which is used to produce these final programs which are now acceptable by our type checker.

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 [2].

In Figure 3 we describe the syntax of the language. The set **TyCtx** describes *type variable contexts* which holds the free type variables in a context. The set **Ctx** describes *variable contexts* which holds information about free term variables, and what type they have. The empty context is denoted $\cdot$, and we extend contexts with variables of a type, or a type variable using $\cup$. The contexts are assumed to be sets.

The unit type is denoted $\mathbb{1}$ and is established as the type of the unit term $()$. Functions types, which are the result of a term $\lambda x\colon A.\ t$ where t has type B are given using $A \to B$. The type of a type abstraction $\Lambda X.\ t$ is given with the syntax $\forall X.\ A$ in which the type variable X may be present in the type A

Variables

$\mathbf{Var} \ni x, y, z$	Variables
$\mathbf{TyVar} \ni X, Y, Z$	Type Variables
$\mathbf{Fun} \ni f, g, h$	Function Symbols

Contexts

$$\mathbf{Ctx} \ni \Gamma ::= \cdot \mid \Gamma \cup \{x : A\}$$
$$\mathbf{TyCtx} \ni \Delta ::= \cdot \mid \Delta \cup \{X\}$$

Types

$$\mathbf{Ty} \ni A, B, C ::= 1 \mid X \mid A \to B \mid \forall X.A \mid A + B \mid A \times B \mid \nu\hat{X}.\ A \mid \mu\hat{X}.\ A$$

Terms

$$\begin{aligned}\mathbf{Tr} \ni r, s, t ::= {} & x \mid f \mid () \mid \lambda x\!:\!A.\ t \mid \Lambda X.\ t \mid r\ s \mid r[A] \mid (t, s) \\ \mid {} & \mathrm{left}(t, A + B) \mid \mathrm{right}(t, A + B) \\ \mid {} & \mathrm{in}_\nu(t, A) \mid \mathrm{out}_\nu(t, A) \mid \mathrm{in}_\mu(t, A) \mid \mathrm{out}_\mu(t, A) \\ \mid {} & \mathrm{case}\ r\ \mathrm{of}\ \{x_1 \Rightarrow s \mid x_2 \Rightarrow t\} \mid \mathrm{split}\ r\ \mathrm{as}\ (x_1, x_2)\ \mathrm{in}\ s\end{aligned}$$

Universal Type Variables

$$UV(\Delta \cup \{\hat{X}\}) := UV(\Delta)$$
$$UV(\Delta \cup \{X\}) := \{X\} \cup UV(\Delta)$$

Type Formation

$$\frac{UV(\Delta) \vdash A\ \mathbf{type} \qquad \Delta \vdash B\ \mathbf{type}}{\Delta \vdash A \to B\ \mathbf{type}} \qquad \frac{\Delta \cup \{\hat{X}\} \vdash A\ \mathbf{type} \qquad \alpha \in \{\nu,\ \mu\}}{\Delta \vdash \alpha\hat{X}.\ A\ \mathbf{type}}$$

$$\frac{\Delta \vdash A\ \mathbf{type} \qquad \Delta \vdash B\ \mathbf{type} \qquad \circ \in \{+,\ \times\}}{\Delta \vdash A \circ B\ \mathbf{type}} \qquad \frac{}{\Delta \vdash 1\ \mathbf{type}}$$

$$\frac{\Delta \cup \{\hat{X}\}\ \mathbf{TyCtx}}{\Delta \cup \{\hat{X}\} \vdash \hat{X}\ \mathbf{type}} \qquad \frac{\Delta \cup \{X\}\ \mathbf{TyCtx}}{\Delta \cup \{X\} \vdash X\ \mathbf{type}} \qquad \frac{\Delta \cup \{X\} \vdash A\ \mathbf{type}}{\Delta \vdash \forall X.\ A\ \mathbf{type}}$$

Context Formation

$$\frac{}{\cdot\ \mathbf{TyCtx}} \qquad \frac{\Delta\ \mathbf{TyCtx} \qquad X \notin \Delta}{\Delta \cup \{X\}\ \mathbf{TyCtx}}$$

$$\frac{}{\Delta \vdash \cdot\ \mathbf{Ctx}} \qquad \frac{\Delta \vdash \Gamma\ \mathbf{Ctx} \qquad x \notin dom(\Gamma) \qquad \Delta \vdash T\ \mathbf{type}}{\Delta \vdash \Gamma \cup \{x\!:\!T\}\ \mathbf{Ctx}}$$

Fig. 1. Language

and the term t. Sum types allow a term of one of two types, A and B, to be injected into a third type $A+B$. We can inject into this sum for terms of type A using the left injection: $\mathrm{left}(t, A+B)$ or for terms of type B on the right using $\mathrm{right}(s, A+B)$. The pair type $A \times B$ is introduced with a pairing term (s,t) where s has type A and t has type B. We introduce inductive types $\mu\hat{X}.\ A$ with the term $\mathrm{in}_\mu(t, \mu\hat{X}.\ A)$. Similarly for coinductive types $\nu\hat{X}.\ A$ we introduce the term $\mathrm{in}_\nu(t, \nu\hat{X}.\ A)$. Similarly each type (save for unit) is equiped with an elimination term, whose meaning will be clear from the dynamic semantics.

Types are introduced by way of type formation rules such that a type A is well formed if we can derive $\Delta \vdash A$ **`type`**. These rules ensure that only types which are strictly positive in μ and ν type variables are allowed, while universally quanitifed variables are unrestricted. This is achieved by segregating the type variables introduced using ν and μ using a hat above the type variable, $\hat{X}$.

Term Substitution

$x[x := t]$	$\equiv t$
$x[y := t]$	$\equiv x$ if $x \neq y$
$\mathtt{f}[x := t]$	$\equiv \mathtt{f}$
$(r\ s)[x := t]$	$\equiv (r[x := t])\ (s[x := t])$
$(\lambda y\colon A.\ r)[x := t]$	$\equiv \lambda y\colon A.\ r[x := t]$ Provided that $\lambda y\colon A.\ r$ is α-converted to use a fresh variable if $y \in \{x\} \cup FV(t)$.
$(\Lambda X.\ r)[x := t]$	$\equiv \Lambda X.\ r[x := t]$
$\mathrm{in}_\alpha(s, A)[x := t]$	$\equiv \mathrm{in}_\alpha(s[x := t], A)$
$\mathrm{out}_\alpha(s, A)[x := t]$	$\equiv \mathrm{out}_\alpha(s[x := t], A)$
$()[x := t]$	$\equiv ()$
$\mathrm{right}(s, A)[x := t]$	$\equiv \mathrm{right}(s[x := t], A)$
$\mathrm{left}(s, A)[x := t]$	$\equiv \mathrm{left}(s[x := t], A)$
$(s, u)[x := t]$	$\equiv (s[x := t], u[x := t])$
$\mathrm{case}\ r\ \mathrm{of}\ \{y \Rightarrow s \mid z \Rightarrow u\}[x := t]$	$\equiv \mathrm{case}\ r[x := t]\ \mathrm{of}\ \{y \Rightarrow s[x := t] \mid z \Rightarrow u[x := t]\}$ Provided that $\lambda y\colon A.\ s$ or $\lambda z\colon A.\ u$ are α-converted to use a fresh variable if $y \in \{x\} \cup FV(t)$ or $z \in \{x\} \cup FV(t)$ respectively.
$\mathrm{split}\ r\ \mathrm{as}\ (y, z)\ \mathrm{in}\ u[x := t]$	$\equiv \mathrm{split}\ r[x := t]\ \mathrm{as}\ (y, z)\ \mathrm{in}\ u[x := t]$ Provided that $\lambda y\colon A.\ \lambda z\colon A.\ u$ is α-converted to use a fresh variable for y or z if $y \in \{x\} \cup FV(t)$ or $z \in \{x\} \cup FV(t)$ respectively.

Fig. 2. Term Substitution

We describe *substitutions* which use a function $FV(t)$ to obtain the free type and term variables from a term. We also choose free variables to be *fresh*, meaning they are chosen from some denumerable set such that they are not present in a given set of variables. A variable chosen in this way we will write as $x = \mathit{fresh}(S)$

Type Substitution on Terms

$$
\begin{array}{ll}
x[X := A] & \equiv x \\
\mathtt{f}[X := A] & \equiv \mathtt{f} \\
()[X := A] & \equiv () \\
(r\ s)[X := A] & \equiv (r[X := A])\ (s[X := A]) \\
(r[A])[X := A] & \equiv (r[X := A])\ (A[X := A]) \\
(\lambda x\colon A.\ r[X := A] & \equiv \lambda x\colon A[X := A].\ r[X := A] \\
\mathrm{in}_\alpha(s, B)[X := A] & \equiv \mathrm{in}_\alpha(s[X := A], B[X := A]) \\
\mathrm{out}_\alpha(s, B)[X := A] & \equiv \mathrm{out}_\alpha(s[X := A], B[X := A]) \\
\mathrm{right}(s, B)[X := A] & \equiv \mathrm{right}(s[X := A], B[X := A]) \\
\mathrm{left}(s, B)[X := A] & \equiv \mathrm{left}(s[X := A], B[X := A]) \\
(s, u)[X := A] & \equiv (s[X := A], u[X := A]) \\
(\mathrm{case}\ r\ \mathrm{of}\ \{y \Rightarrow s \mid z \Rightarrow u\})[X := A] & \equiv \mathrm{case}\ r[X := A]\ \mathrm{of} \\
 & \qquad \{\ y \Rightarrow s[X := A] \\
 & \qquad \mid z \Rightarrow u[X := A] \\
(\mathrm{split}\ r\ \mathrm{as}\ (y, z)\ \mathrm{in}\ u)[X := A] & \equiv \mathrm{split}\ r[X := A]\ \mathrm{as}\ (y, z)\ \mathrm{in}\ u[X := A]
\end{array}
$$

Type Substitution on Types

$$
\begin{array}{ll}
X[X := A] & \equiv A \\
X[Y := A] & \equiv X \text{ if } X \neq Y \\
\mathbb{1}[X := A] & \equiv \mathbb{1} \\
B + C[X := A] & \equiv B[X := A] + C[X := A] \\
B \times C[X := A] & \equiv B[X := A] \times C[X := A] \\
(B \to C)[X := A] & \equiv B[X := A] \to C[X := A] \\
(\forall Y.\ B)[X := A] & \equiv \forall Y.\ B[X := A] \\
 & \quad \text{Provided that } (\forall Y.\ B) \text{ is } \alpha\text{-converted to use} \\
 & \quad \text{a fresh type-variable if } Y \in \{X\} \cup FV(A). \\
(\Lambda Y.\ r)[X := A] & \equiv \Lambda Y.\ r[X := A] \\
 & \quad \text{Provided that } (\Lambda Y.\ r) \text{ is } \alpha\text{-converted to use} \\
 & \quad \text{a fresh type-variable if } Y \in \{X\} \cup FV(A). \\
(\alpha Y.\ r)[X := A] & \equiv \alpha Y.\ r[X := A],\ \alpha \in \{\nu,\ \mu\} \\
 & \quad \text{Provided that } (\alpha Y.\ r) \text{ is } \alpha\text{-converted to use} \\
 & \quad \text{a fresh type-variable if } Y \in \{X\} \cup FV(A).
\end{array}
$$

Fig. 3. Type Substitution

if it is *fresh* with respect to the set S. Substitutions of a single variable will be written $[X := A]$ or $[x := t]$ for type and term variables respectively. The full operational meaning of substitutions is given in Figure 2 and Figure 3.

Reduction Rules

$$(\lambda x : A.r)\ s \rightsquigarrow_1 r[x := s] \quad (\Lambda X.r)\ A \rightsquigarrow_1 r[X := A]$$

$$\mathrm{out}_\alpha(\mathrm{in}_\alpha(r, U), U) \rightsquigarrow_1 r \qquad \mathtt{f} \rightsquigarrow_\delta \Omega(\mathtt{f})$$

$$\text{case left}(r, A + B) \text{ of } \{x \Rightarrow s \mid y \Rightarrow t\} \rightsquigarrow_1 s[x := r]$$

$$\text{case right}(r, A + B) \text{ of } \{x \Rightarrow s \mid y \Rightarrow t\} \rightsquigarrow_1 t[y := r]$$

$$\text{split } (r, s) \text{ as } (x, y) \text{ in } t \rightsquigarrow_1 t[x := r][y := s]$$

Structural Rules

$$\frac{r\ \mathcal{R}\ r'}{r\ \mathcal{R}^s\ r'} \qquad \frac{r\ \mathcal{R}^s\ r'}{r\ s\ \mathcal{R}^s\ r'\ s} \qquad \frac{r\ \mathcal{R}^s\ r'}{r\ A\ \mathcal{R}^s\ r'\ A} \qquad \frac{r\ \mathcal{R}^s\ r'}{\mathrm{out}_\alpha(r, U)\ \mathcal{R}^s\ \mathrm{out}_\alpha(r', U)}$$

$$\frac{r\ \mathcal{R}^s\ r'}{\text{case } r \text{ of } \{x \Rightarrow s \mid y \Rightarrow t\}\ \mathcal{R}^s\ \text{case } r' \text{ of } \{x \Rightarrow s \mid y \Rightarrow t\}}$$

$$\frac{r\ \mathcal{R}^s\ r'}{\text{split } r \text{ as } (x, y) \text{ in } t\ \mathcal{R}^s\ \text{split } r' \text{ as } (x, y) \text{ in } t}$$

Evaluation Relations

$$\begin{aligned}
&r \rightsquigarrow_n s ::= r\ \rightsquigarrow_1^s\ s \\
&r \rightsquigarrow s\ \ ::= r\ \rightsquigarrow_\delta^s\ s \vee r \rightsquigarrow_n s \\
&r\ R^+\ s\ ::= r\ R\ s \vee (\exists r'.r\ R\ r' \wedge r'\ R^+\ s) \\
&r\ R^*\ s\ ::= r = s \vee r\ R^+\ s
\end{aligned}$$

Fig. 4. Evaluation

We define the evaluation relation $\rightsquigarrow$ in Figure 4. This relation is the usual normal order evaluation relation. It is deterministic and so there is always a unique redex. We take the transitive closure of the relation to be $\rightsquigarrow^*$.

We introduce recursive terms by way of function constants. Although it is possible to encoded these directly in System F, it simplifies the presentation to provide them explicitly. Function constants are drawn from a set **Fun**. We couple our terms with a function Ω which associates a function constant f with a term t, $\Omega(f) = t$, where t may itself contain any function constants in the domain of Ω. We make use of this function in the $\rightsquigarrow$ relation which allows us to unfold recursively defined functions.

For a term t with type A in a context $\Delta; \Gamma$ we write $\Delta\ ; \Gamma \vdash t : A$. A type derivation must be given using the rules given in Figure 5.

$$\frac{\Delta \vdash \Gamma \cup \{x\!:\!A\}\ \ \mathbf{Ctx}}{\Delta\,;\Gamma \cup \{x\!:\!A\} \vdash x\,:\,A}\ I^{Var} \qquad \frac{}{\Delta\,;\Gamma \vdash ()\,:\,\mathbb{1}}\ I^{1}$$

$$\frac{\Delta \cup \{X\}\,;\Gamma \vdash t\,:\,A}{\Delta\,;\Gamma \vdash (\Lambda X.\,t)\,:\,\forall X.\,A}\ I^{\forall} \qquad \frac{\Delta\,;\Gamma \vdash t\,:\,\forall X.\,A \qquad \Delta \vdash B\ \ \mathbf{type}}{\Delta\,;\Gamma \vdash t[B]\,:\,A[X := B]}\ E^{\forall}$$

$$\frac{\Delta\,;\Gamma \cup \{x\!:\!A\} \vdash t\,:\,B}{\Delta\,;\Gamma \vdash (\lambda x\!:\!A.\,t)\,:\,A \to B}\ I^{\to} \qquad \frac{\Delta\,;\Gamma \vdash r\,:\,A \to B \qquad \Delta\,;\Gamma \vdash s\,:\,A}{\Delta\,;\Gamma \vdash (r\ s)\,:\,B}\ E^{\to}$$

$$\frac{\Gamma \vdash \Omega(f) : A}{\Delta\,;\Gamma \vdash f\,:\,A}\ I^{\delta} \qquad \frac{\Delta\,;\Gamma \vdash r\,:\,A \qquad \Delta\,;\Gamma \vdash s\,:\,B}{\Delta\,;\Gamma \vdash (r,s)\,:\,(A \times B)}\ I^{\times}$$

$$\frac{\Delta\,;\Gamma \vdash t\,:\,A \qquad \Delta \vdash B\ \ \mathbf{type}}{\Delta\,;\Gamma \vdash \mathrm{left}(t, A+B)\,:\,(A+B)}\ \mathrm{I}^{+}_{L} \qquad \frac{\Delta\,;\Gamma \vdash t\,:\,B \qquad \Delta \vdash A\ \ \mathbf{type}}{\Delta\,;\Gamma \vdash \mathrm{right}(t, A+B)\,:\,(A+B)}\ \mathrm{I}^{+}_{R}$$

$$\frac{\Delta\,;\Gamma \vdash t\,:\,\alpha\hat{X}.\,A \qquad \alpha \in \{\mu,\nu\}}{\Delta\,;\Gamma \vdash \mathrm{out}_{\alpha}(t, \alpha\hat{X}.\,A)\,:\,A[\hat{X} := \alpha\hat{X}.\,A]}\ \mathrm{E}^{\alpha}$$

$$\frac{\Delta\,;\Gamma \vdash t\,:\,A[\hat{X} := \alpha\hat{X}.\,A] \qquad \alpha \in \{\mu,\nu\}}{\Delta\,;\Gamma \vdash \mathrm{in}_{\alpha}(t, \alpha\hat{X}.\,A)\,:\,\alpha\hat{X}.\,A}\ \mathrm{I}^{\alpha}$$

$$\frac{\Delta\,;\Gamma \vdash r\,:\,A+B \qquad \Delta\,;\Gamma \cup \{x\!:\!A\} \vdash t\,:\,C \qquad \Delta\,;\Gamma \cup \{y\!:\!B\} \vdash s\,:\,C}{\Delta\,;\Gamma \vdash (\mathrm{case}\ r\ \mathrm{of}\ \{x \Rightarrow t \mid y \Rightarrow s\})\,:\,C}\ E^{+}$$

$$\frac{\Delta\,;\Gamma \vdash s\,:\,A \times B \qquad \Delta\,;\Gamma \cup \{x\!:\!A\} \cup \{y\!:\!B\} \vdash t\,:\,C}{\Delta\,;\Gamma \vdash (\mathrm{split}\ s\ \mathrm{as}\ (x,y)\ \mathrm{in}\ t)\,:\,C}\ E^{\times}$$

Fig. 5. System F Proof Rules

Without further restrictions, this type system is unsound. First, the Delta rule for function constants clearly allows non-termination given $\Omega(f) = f : T$. We will deal with this potentiality later when we describe cyclic proof in Section 4.

In addition, we will need a concept of a one-hole context. This allows us to describe terms which are embedded in a surrounding term. We write this as $C[t]$ when we wish to say that the term t has a surrounding context.

4 Cyclic Proof

Typically, in functional programming languages, type checking for defined functions is done by use of a typing rule that assumes the type of the function and proceeds to check the body. This is the familiar rule from programming languages such as Haskell [10] [15] [13]. An example of such a typing rule is as follows:

$$\frac{\Delta\,;\Gamma \cup \{f : A \to B\} \vdash \Omega(f) \,:\, A \to B}{\Delta\,;\Gamma \vdash f \,:\, A \to B}\; FunRec$$

Coupled with guardedness or structural recursion and positivity restrictions on the form of recursive types to ensure (co)termination, this rule will be sound. However, it is also *opaque*. Any transformation of this proof tree will be rigidly expressed in terms of the original function declarations.

In order to allow more fluidity in the structure of our proof trees we introduce a notion of a cyclic proof. Cyclicity can be introduced simply by allowing the type rules to be a coinductive type (in the meta-logic) rather than an inductive one. However, for us to produce the cycles we are interested in, we need to add an additional term and typing rule which allows explicit substitutions, and one derived rule which makes use of the fact that we identify all proofs under the reduction relation $\rightsquigarrow$ as being equivalent. The explicit substitutions will also require an additional evaluation rule which transforms them into computed substitutions. Explicit substitutions can also be introduced at the level of type substitutions, but these are not necessary for our examples.

The Conv, E^{Ω} and the I^{θ} follow from theorems about the calculus which can be established in the meta-logic and in fact we have a formal proof of these theorems for the given calculus in Coq.

We will not prove general soundness conditions, but rely on prior work showing that structural induction and the guardedness are sufficient conditions [9]. Once these conditions have been satisfied, we can assume the correctness of the proof.

Definition 1 (Structural Ordering). *A term t is said to be* less in the structural ordering *than a term s, or* $t <^s s$ *using the relation* $<^s$ *given by the inductive definition in Figure 6.*

Definition 2 (Structural Recursion). *A derivation is said to be structurally recursive if for every sequent used in a* I^{θ} *rule, there exists a privileged variable*

Explicit Substitutions

$t\ \langle x := s \rangle$

Typing Rules

$$\frac{\Delta\,;\Gamma \cup \Gamma' \vdash u\ :\ B \qquad \Delta\,;\Gamma \cup \{x\!:\!B\} \vdash t\ :\ A}{\Delta\,;\Gamma \cup \Gamma' \vdash t\ \langle x := u \rangle\ :\ A}\ I^{\theta}$$

$$\frac{\Delta\,;\Gamma \vdash t\ :\ A \qquad t \rightsquigarrow^{*} s}{\Delta\,;\Gamma \vdash s\ :\ A}\ \text{Conv} \qquad \frac{\Delta\,;\Gamma \vdash C[\Omega(f)]\ :\ A}{\Delta\,;\Gamma \vdash C[f]\ :\ A}\ E^{\Omega}$$

Extended Evaluation

$$t\ \langle x := u \rangle \rightsquigarrow t[x := u] \qquad \frac{t \rightsquigarrow t'}{t\ \langle x := u \rangle \rightsquigarrow t'\ \langle x := u \rangle}$$

Structural Ordering

$$\frac{\text{case } r \text{ of } \{x \Rightarrow s \mid y \Rightarrow t\}}{x\ \mathcal{S}\ r} \qquad \frac{\text{case } r \text{ of } \{x \Rightarrow s \mid y \Rightarrow t\}}{y\ \mathcal{S}\ r}$$

$$\frac{\text{split } r \text{ as } (x, y) \text{ in } t}{x\ \mathcal{S}\ r} \qquad \frac{\text{split } r \text{ as } (x, y) \text{ in } t}{y\ \mathcal{S}\ r}$$

$$\frac{\text{out}_{\alpha}(t, \alpha\hat{X}.\ A)}{\text{out}_{\alpha}(t, \alpha\hat{X}.\ A)\ \mathcal{S}\ t} \qquad <^{s} := S^{*} \quad \text{Transitive closure of } S$$

Fig. 6. Explicit Substitution and Structural Ordering

x such that for all I^θ rules, with substitution σ_i, using that sequent we have that $x \in dom(\sigma_i)$ and $\sigma(x) <^s x$.

It should be mentioned that there is nothing in particular needed for this definition aside from some guarantee that we are well founded. As such this represents a particular implementation *strategy* and we could very well have used a more liberal approach. One such approach is *size-change termination* as described by Neil Jones et al. in [11].

Similarly, we must produce a rule for coinductive types which ensures that all terms of coinductive type are *productive*. We here develop a *guardedness* condition specific to our type theory of cyclic proofs. Essentially this condition ensures we encounter an introduction of a constructor which can not be eliminated on all coinductive cyclic paths. The only intermediate terms must reduce finitely through eliminations of finite or inductively defined terms, ensuring that we will not compute indefinitely prior to producing a constructor.

While structural recursion is focused on determining whether the arguments of a recursive term are subterms of some previously destructured term, the dual problem is of determining if a recursive term's context ensures that the term grows. This means we need ways of describing the surrounding context of a term. However, the contexts we have developed thus far are structured in terms of *experiments*. With coinductive terms we need exactly the opposite variety of contexts, those surrounding terms which are *not* experiments.

The key important features of the contexts we are interested in turns out to be whether or not they introduce constructors, and whether they are guaranteed not to remove them. These properties are necessary in the construction of our proof that guardedness leads to *productivity*.

We can describe the relevant features of the context by describing a *path*. This *path* is a series of constructors that allows us to demonstrate which directions to take down a proof tree to arive at a recurrence.

Definition 3 (Path). *A* path *is a finite sequence of pairs of a proof rule from Figure 5 and an index denoting which antecedent it decends from. This pair is described as a* rule-index-pair*.*

An example of such a path would be the following:

$$\text{OrIntroL}^1,\text{AndIntro}^2,\text{ImpIntro}^1$$

This denotes the context:

$$\text{left}((\lambda x : B.-, s), A)$$

With some unknown (and for the purpose of proving productivity, inconsequential) variable x, term s and types A and B.

With this in hand we can establish conditions for guardedness with recursive definitions based on constraints on paths.

Definition 4 (Admissible). *A path is called admissible if the first element c of the path $p = c, p'$ is one of the rule-index-pairs $OrIntroL^1$, $OrIntroR^1$, $AndIntro^1$, $AndIntro^2$, $AllIntro^1$, $\alpha Intro^1$, $ImpIntro^1$, $OrElim^2$, $OrElim^3$, $AndElim^2$, $AllElim^1$, $Delta^1$ and p' is an admissible path.*

Definition 5 (Guardedness). *A path is called guarded if it terminates at a Pointer rule, with the sequent having a coinductive type and the path can be partitioned such that $p = p', [c], p''$ where c is one of the rule-index-pairs $OrIntroL^1$, $OrIntroR^1$, $AndIntro^1$, $AndIntro^2$, $\nu Intro^1$, $ImpIntro^1$ which we will call* guards *and p' and p'' are admissible paths.*

The idea behind the guardedness condition is that we have to be assured that as we take a cyclic path we produce an Intro rule which will never be removed by the reduction relation. The left hand-side of an elimination rule will never cause the elimination of such an introduction and so is *safe*. However, the right hand side of an elimination rule may in fact cause the removal of the introduction rule when we use the evaluation relation.

5 Program Transformation

Supercompilation is a family of program transformation techniques. It essentially consists of *driving*, *information propagation*, *generalisation* and *folding*.

Driving is simply the unfolding and elimination of cuts. Cut-elimination involves the removal of all intermediate datastructures. This includes anything that would be normalised by evaluation in a language like Coq or Agda, including beta-elimination, case elimination or pair selection. Driving, as it removes cuts from infinite proof objects, generates potentially infinite computations.

Information propagation is the use of meta-logical information about eliminations such as case branches. For example, when a meta-variable is destructed in a case branch, the particular de-structuring may be propagated into sub-terms. This is achieved by an inversion on the typing derivation.

Folding is the search for recurrences in the driven program. A recurrence will be an expression which is a variable renaming of a former expression. Essentially, if a recurrence is found we can create a new recursive function having the same body as the driven expression with a recursive call at the recurrence.

Generalisation may be used in order to find opportunities for folding. We can abstract away arguments which would cause further driving and would not allow us to fold.

Our notion of equivalence of proofs must be quite strict if it is to preserve the operational behaviour of the program. The notion of equivalence we use here is *contextual equivalence*.

Definition 6 (Contextual Equivalence). *For all terms s, t and types A and type derivations $\cdot \vdash s : A$ and $\cdot \vdash t : A$, and given any experiment e such that $x : A \vdash e : B$ then we have that $e[x := s] \Downarrow$ and $e[x := t] \Downarrow$ then $s \sqsubseteq t$. If $s \sqsubseteq t$ and $t \sqsubseteq s$ then s is contextually equivalent to t or $s \cong t$.*

In the examples, the relation between the original and transformed proofs is simply either a compatible relation with the formation rules, or the term is related up to beta-equivalence. In the case of unfolding, it's clear that no real change has taken place since the unfolded pre-proof just extends the prefix of the potentially infinite pre-proof, and is identical by definition. The finite prefix is merely a short hand for the infinite unfolding of the pre-proof.

Dealing with reduction under the evaluation relation is more subtle. In order to establish equivalence here we need to show that if the term reduces, it reduces to an outer-most term which will itself reduce when an experiment is applied. This is essentially a *head normal form*, that is, a term whose outermost step will not reduce in any context. This idea is essential to defining productivity, since it is precisely the fact that we have *done something* irrevocable which gives us productivity.

Folding is also somewhat complex as, in our case, we will require the use of generalisation, which is essentially running the evaluation relation backwards in order to find terms which will be equivalent under reduction, and cycles in the proof which can lead to potential unsoundess. The key insight of this paper is that in fact, unsoundess can not be introduced if the cycles themselves are productive or inductive for coinduction and induction respectively.

Generally the program transformation technique itself is controlled by using some additional termination method such as a depth bound or more popularly the homeomorphic embedding. This however does not influence the correctness of the outcome. If an algorithm in the supercompilation family terminates, the final program is a faithful bisimulation of the original.

Since all of the examples given in this paper clearly follow the fold/unfold generalise paradigm, and all examples are inductive/productive, the correctness can be assumed. In a future work we hope to present the algorithm that was used to find these examples in more detail, and to show that it will in general produce contextually equivalent programs. We will see how these elements are applied in practice by using these techniques to work with cyclic proofs.

5.1 Reduction

Previously we gave a bird's-eye view of supercompilation as being a family of program transformations composed of driving, generalisation and folding. Cyclic proofs give us the tools necessary to justify folding in the context of types and driving is simply the unfolding of a cyclic pre-proof.

In order to perform folding however, we need to be able to arrive at nodes which are α-renamings of former nodes. In order to do this in general we need to be able to generalise terms. We can, using generalisation, regenerate proofs which simply make reference to recursive functions, by generalising to reproduce α-renamings of the function bodies and folding. This ensures that we can produce *at least* the proofs possible already using the original term.

For higher order functional languages, there are a potentially infinite number of generalisations of two terms, and the least general generalisation may itself consist of many incomparable terms [14]. For this reason, some heuristic

approach needs to be applied in order to find appropriate generalisations. We will not be concerned about the particular heuristic approach used to determine generalisations as this is quite a complex subject, but only that it meet the condition that the generalisation can be represented as an elimination rule in the proof tree and that will regenerate the original proof tree under evaluation.

6 Example Revisited

With the notion of cyclic proof, we now have at our disposal the tools necessary to transform pre-proofs into proofs. We will revisit the *sumlen* example given as motivation for the present work and see how we can represent the transformations.

We take again an example using the *co-natural numbers* $\mathbb{N}^\infty \equiv \nu X.1 + X$ and potentially infinite lists $[A] \equiv \Lambda A.\nu X.1 + (A \times X)$. Here we take Ω to be defined as:

```
Ω(zero) := in(left( (),N̄))
Ω(plus) := λ x y : N̄ .
  case (out(x,N̄)) of
    | z ⇒ ys
    | n ⇒
     fold(right(plus n y),N̄)
Ω(sumlen) := λ xs : [N̄] .
  case (out(xs,[N̄])) of
    | nil ⇒ zero
    | p ⇒
     split p as (n,xs′)
      in in(right(plus n (sumlen xs′)),N̄)
```

We can now produce the type derivation by performing the successive steps given explicitly in Figure . Here in the final step we have driven the proof tree to the point that we can now reference two previous nodes. One of those is labelled with a †, the other with a ∗. This final pre-proof is now a proof because it satisfies the guardedness condition. We have taken the liberty of introducing a an additional derived rule $\text{Cons}^{[\![\overline{\mathbb{N}}]\!]}$. It is merely a shorthand for the use of I_R^+ and I^ν, together with a proof that these types admissible under the formation rules.

We can then produce a *residual* program from the cyclic proof. This is simply a mutually recursive (or letrec) function definition which makes any cycle into a recursive call. The residual term will be essentially the one given in agda above.

7 Related Work

The present work uses a program transformation in the supercompilation family. This was first described by Turchin [17] and later popularised by Sørensen, Glück

$$\dfrac{\cdot\,;\{xs\colon [\![\overline{\mathbb{N}}]\!]\} \vdash \mathtt{case}\ xs\ \mathtt{of}\ \{[] \Rightarrow \mathit{czero} \mid x :: xs' \Rightarrow \mathit{csucc}\ (x + (\mathtt{sumlen}\ xs'))\} : \overline{\mathbb{N}}}{\cdot\,;\{xs\colon [\![\overline{\mathbb{N}}]\!]\} \vdash \mathtt{sumlen}\ xs : \overline{\mathbb{N}}}\ I^{\Omega}$$

$$\Downarrow$$

$$\dfrac{\dfrac{\cdot\,;xs : [\![\overline{\mathbb{N}}]\!] \vdash xs : [\![\overline{\mathbb{N}}]\!] \qquad \cdot\,;\ldots \vdash \mathit{czero} : \overline{\mathbb{N}} \qquad \cdot\,;\{x\colon\overline{\mathbb{N}}, xs\colon[\![\overline{\mathbb{N}}]\!]\} \vdash \mathit{csucc}\ (x + (\mathtt{sumlen}\ xs')) : \overline{\mathbb{N}}}{\cdot\,;\{xs\colon [\![\overline{\mathbb{N}}]\!]\} \vdash \mathtt{case}\ xs\ \mathtt{of}\ \{[] \Rightarrow \mathit{czero} \mid x :: xs' \Rightarrow \mathit{csucc}\ (x + (\mathtt{sumlen}\ xs'))\} : \overline{\mathbb{N}}}\ E^{+}}{\cdot\,;\{xs\colon [\![\overline{\mathbb{N}}]\!]\} \vdash \mathtt{sumlen}\ xs : \overline{\mathbb{N}}}\ I^{\Omega}$$

$$\Downarrow$$

$$\dfrac{\dfrac{\cdot\,;xs : [\![\overline{\mathbb{N}}]\!] \vdash xs : [\![\overline{\mathbb{N}}]\!] \qquad \cdot\,;\ldots \vdash \mathit{czero} : \overline{\mathbb{N}} \qquad \dfrac{\cdot\,;\{x\colon\overline{\mathbb{N}}, xs\colon[\![\overline{\mathbb{N}}]\!]\} \vdash x + (\mathtt{sumlen}\ xs') : \overline{\mathbb{N}}}{\cdot\,;\{x\colon\overline{\mathbb{N}}, xs'\colon[\![\overline{\mathbb{N}}]\!]\} \vdash \mathit{csucc}\ (x + (\mathtt{sumlen}\ xs')) : \overline{\mathbb{N}}}\ \mathrm{Cons}^{[\![\overline{\mathbb{N}}]\!]}}{\cdot\,;\{xs\colon [\![\overline{\mathbb{N}}]\!]\} \vdash \mathtt{case}\ xs\ \mathtt{of}\ \{[] \Rightarrow \mathit{czero} \mid x :: xs' \Rightarrow \mathit{csucc}\ (x + (\mathtt{sumlen}\ xs'))\} : \overline{\mathbb{N}}}\ E^{+}}{\cdot\,;\{xs\colon [\![\overline{\mathbb{N}}]\!]\} \vdash \mathtt{sumlen}\ xs : \overline{\mathbb{N}}}\ I^{\Omega}$$

$$\Downarrow$$

$$\dfrac{\dfrac{\cdot\,;xs : [\![\overline{\mathbb{N}}]\!] \vdash xs : [\![\overline{\mathbb{N}}]\!] \qquad \cdot\,;\ldots \vdash \mathit{czero} : \overline{\mathbb{N}} \qquad \dfrac{\dfrac{\cdot\,;\{x\colon\overline{\mathbb{N}}, xs'\colon[\![\overline{\mathbb{N}}]\!]\} \vdash \mathtt{case}\ x\ \mathtt{of}\ \{\mathit{czero} \Rightarrow \mathtt{sumlen}\ xs' \mid \mathit{csucc}\ x \Rightarrow \mathit{csucc}\ (x + (\mathtt{sumlen}\ xs'))\} : \overline{\mathbb{N}}}{\cdot\,;\{x\colon\overline{\mathbb{N}}, xs'\colon[\![\overline{\mathbb{N}}]\!]\} \vdash x + (\mathtt{sumlen}\ xs') : \overline{\mathbb{N}}}\ I^{\Omega} \qquad \cdot\,;\{x\colon\overline{\mathbb{N}}, xs'\colon[\![\overline{\mathbb{N}}]\!]\} \vdash \mathit{csucc}\ (x + (\mathtt{sumlen}\ xs')) : \overline{\mathbb{N}}}\ \mathrm{Cons}^{[\![\overline{\mathbb{N}}]\!]}}{\cdot\,;\{xs\colon [\![\overline{\mathbb{N}}]\!]\} \vdash \mathtt{case}\ xs\ \mathtt{of}\ \{[] \Rightarrow \mathit{czero} \mid x :: xs' \Rightarrow \mathit{csucc}\ (x + (\mathtt{sumlen}\ xs'))\} : \overline{\mathbb{N}}}\ E^{+}}{\cdot\,;\{xs\colon [\![\overline{\mathbb{N}}]\!]\} \vdash \mathtt{sumlen}\ xs : \overline{\mathbb{N}}}\ I^{\Omega}$$

$$\Downarrow$$

$$\mathcal{D} := \dfrac{\dfrac{\cdot\,;\{xs'\colon[\![\overline{\mathbb{N}}]\!]\} \vdash xs' : [\![\overline{\mathbb{N}}]\!] \qquad \overline{\cdot\,;\{xs\colon[\![\overline{\mathbb{N}}]\!]\} \vdash \mathtt{sumlen}\ xs : \overline{\mathbb{N}}}^{\dagger}}{\cdot\,;\{xs'\colon[\![\overline{\mathbb{N}}]\!]\} \vdash \mathtt{sumlen}\ xs\ \langle xs := xs'\rangle : \overline{\mathbb{N}}}\ I^{\theta} \qquad \mathtt{sumlen}\ xs\ \langle xs := xs'\rangle \rightsquigarrow^{*} \mathtt{sumlen}\ xs'}{\cdot\,;\{xs'\colon[\![\overline{\mathbb{N}}]\!]\} \vdash \mathtt{sumlen}\ xs' : \overline{\mathbb{N}}}\ \mathrm{Conv}$$

$$\mathcal{E} := \dfrac{\dfrac{\cdot\,;\{xs'\colon[\![\overline{\mathbb{N}}]\!]\} \vdash xs' : \overline{\mathbb{N}} \qquad \overline{\cdot\,;\{x\colon\overline{\mathbb{N}}, xs\colon[\![\overline{\mathbb{N}}]\!]\} \vdash \mathit{csucc}\ (x + (\mathtt{sumlen}\ xs)) : \overline{\mathbb{N}}}^{*}}{\cdot\,;\{x\colon\overline{\mathbb{N}}, xs'\colon[\![\overline{\mathbb{N}}]\!]\} \vdash (\mathit{csucc}\ (x + (\mathtt{sumlen}\ xs)))\ \langle xs := xs'\rangle : \overline{\mathbb{N}}}\ I^{\theta} \qquad (\mathit{csucc}\ (x + (\mathtt{sumlen}\ xs)))\ \langle xs := xs'\rangle \rightsquigarrow^{*} \mathit{csucc}\ (x + (\mathtt{sumlen}\ xs'))}{\cdot\,;\{x\colon\overline{\mathbb{N}}, xs'\colon[\![\overline{\mathbb{N}}]\!]\} \vdash \mathit{csucc}\ (x + (\mathtt{sumlen}\ xs')) : \overline{\mathbb{N}}}\ \mathrm{Conv}$$

$$\dfrac{\dfrac{\dfrac{\cdot\,;xs : [\![\overline{\mathbb{N}}]\!] \vdash xs : [\![\overline{\mathbb{N}}]\!] \qquad \cdot\,;\ldots \vdash \mathit{czero} : \overline{\mathbb{N}} \qquad \dfrac{\dfrac{\cdot\,;\{x\colon\overline{\mathbb{N}}\} \vdash x : \overline{\mathbb{N}} \qquad \mathcal{D} \qquad \mathcal{E}}{\cdot\,;\{x\colon\overline{\mathbb{N}}, xs'\colon[\![\overline{\mathbb{N}}]\!]\} \vdash \begin{array}{l}\mathtt{case}\ x\ \mathtt{of}\\ \quad \{\ \mathit{czero} \Rightarrow \mathtt{sumlen}\ xs'\\ \quad \mid \mathit{csucc}\ x \Rightarrow \mathit{csucc}\ (x + (\mathtt{sumlen}\ xs'))\end{array} : \overline{\mathbb{N}}}\ E^{+}}{\cdot\,;\{x\colon\overline{\mathbb{N}}, xs'\colon[\![\overline{\mathbb{N}}]\!]\} \vdash x + (\mathtt{sumlen}\ xs') : \overline{\mathbb{N}}}\ I^{\Omega} \qquad \cdot\,;\{x\colon\overline{\mathbb{N}}, xs'\colon[\![\overline{\mathbb{N}}]\!]\} \vdash \mathit{csucc}\ (x + (\mathtt{sumlen}\ xs')) : \overline{\mathbb{N}}}\ \mathrm{Cons}^{[\![\overline{\mathbb{N}}]\!]}\ {}^{*}}{\cdot\,;xs : [\![\overline{\mathbb{N}}]\!] \vdash xs : [\![\overline{\mathbb{N}}]\!] \qquad \cdot\,;\ldots \vdash \mathit{czero} : \overline{\mathbb{N}}}\ E^{+}}{\cdot\,;\{xs\colon [\![\overline{\mathbb{N}}]\!]\} \vdash \begin{array}{l}\mathtt{case}\ xs\ \mathtt{of}\\ \quad \{\ [] \Rightarrow \mathit{czero}\\ \quad \mid x :: xs' \Rightarrow \mathit{csucc}\ (x + (\mathtt{sumlen}\ xs'))\end{array} : \overline{\mathbb{N}}}\ I^{\Omega}\ {}^{\dagger}$$

Fig. 7. Sumlen Cyclic Proof

and Jones [16]. We essentially use the same algorithms with the addition of the use of type information to guide folding.

The use of cylic proofs was developed by Brotherston [4]. We extend this work by dealing also with coinductive types and make use of it in a Curry-Howard settings.

The correspondence between cyclic proof and functional programs has previously been described by Robin Cockett [5]. His work also makes a distinction between inductive and coinductive types. Our work differs in using super compilation as a means of proving type inhabitation.

Various approaches to proving type inhabitation for coinductive types have appeared in the literature. Bertot and Komendanskaya give a method in [3]. A method is also given using sized types is given by Abel in [1]. The approach in this paper differs in that we use transformation of the program rather than reasoning about the side conditions.

8 Conclusion and Future Work

The use of program transformation techniques for proofs of type inhabitation is attractive for a number of reasons. It gives us the ability to mix programs which may or may not be type correct to arrive at programs which are provably terminating. We can keep an audit trail of the reasoning by which the programs were transformed. And we can admit a larger number of programs by transformation to a form which is syntactically correct, obviating the need for complex arguments about termination behaviour. For these reasons we feel that this work could be of value to theorem provers in the future.

To the best of the authors knowledge, no examples of a supercompilation algorithm have yet been given for a dependently typed language. The authors hope to extend the theory to dependent types in the future such that the algorithm might be of assistance to theorem provers.

Currently work is being done on a complete automated proof of correctness of a supercompilation algorithm for the term language described in this paper in the proof assistant Coq. The cyclic proofs are represented using a coinductive data-type, rather than the usual inductive description.

The technique as presented works well for many examples, however there are some examples in which direct supercompilation is insufficient. The following program tries to capture the notion of a semi-decidable existential functional which takes a semi-decidable predicate over the type A. The usual way to write this in a functional language is to use the Sierpinski type [8], the type of one constructor. Here truth is represented with termination, and non-termination gives rise to falsehood.

```
data S : Set where
  T : S
```

However since languages such as Coq and Agda will not allow us to directly represent non-termination, we will embed the Sierpinski type in the delay monad.

```
codata Delay (A : Set) : Set where
  now : A → Delay A
  later : Delay A → Delay A
```

The clever reader might notice that this is in fact isomorphic to the co-natural numbers and that join is simply the minimum of two potentially infinite numbers.

```
join : Delay S → Delay S → Delay S
join (now T) x = now T
join x (now T) = now T
join (later x) (later y) = later (join x y)
ex : {A : Set} → (A → Delay S) →
  Stream A → Delay S
ex p (x::xs) = join (p x) (ex p xs)
```

By unfolding *join* and *ex* we eventually arrive at a term:

```
-- join x' (join (p x) (ex p xs))
```

This term is a repetition of the original body of *ex*, with $p\ x$ abstracted, provided that join is associative. Unfortunately, using direct supercompilation, we are unable to derive a type correct term automatically. However, using ideas presented by Klyutchnikov and Romanenko [12], the technique might be extended in such a way to provide an automated solution for this example as well. Using the fact that the recurrence is contextually equivalent, we can fold the proof to obtain the following program, which is productive, and admissible into Agda.

```
mutual
  ex_trans : {A : Set} →
    (A → Delay S) → Stream A →
    Delay S
  ex_trans p (x::xs) = later (j (p x) p xs)
  j : {A : Set} →
    Delay S → (A → Delay S) →
    Stream A → Delay S
  j (now T) p _ = now T
  j (later n) p (x::xs) = later (j (join n (p x)) p xs)
```

References

1. Andreas Abel. Termination checking with types. Technical report, Institut fr Informatik, Ludwigs-Maximilians-Universitt Mnchen, 2002.
2. Andreas Abel. Typed Applicative Structures and Normalization by Evaluation for System F^{ω}. In Erich Grdel and Reinhard Kahle, editors, *CSL*, volume 5771 of *Lecture Notes in Computer Science*, pages 40–54. Springer, 2009.

3. Yves Bertot and Ekaterina Komendantskaya. Inductive and Coinductive Components of Corecursive Functions in Coq. *Electron. Notes Theor. Comput. Sci.*, 203(5):25–47, 2008.
4. James Brotherston. Cyclic Proofs for First-Order Logic with Inductive Definitions. In B. Beckert, editor, *Automated Reasoning with Analytic Tableaux and Related Methods: Proceedings of TABLEAUX 2005*, volume 3702 of *LNAI*, pages 78–92. Springer-Verlag, 2005.
5. J. Robin B. Cockett. Deforestation, program transformation, and cut-elimination. *Electr. Notes Theor. Comput. Sci.*, 44(1), 2001.
6. Thierry Coquand. Infinite objects in type theory. In *TYPES 93: Proceedings of the international workshop on Types for proofs and programs*, pages 62–78, Secaucus, NJ, USA, 1994. Springer-Verlag New York, Inc.
7. Nils A. Danielsson, John Hughes, Patrik Jansson, and Jeremy Gibbons. Fast and loose reasoning is morally correct. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, volume 41, pages 206–217, New York, NY, USA, January 2006. ACM.
8. M. H. Escardo. Synthetic topology of data types and classical spaces. *ENTCS, Elsevier*, 87:21–156, 2004.
9. Eduardo Gimnez. 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.
10. Andrew D. Gordon. Bisimilarity as a theory of functional programming. *Theor. Comput. Sci.*, 228(1-2):5–47, 1999.
11. Neil D. Jones. Program termination analysis by size-change graphs (abstract). In *IJCAR*, pages 1–4, 2001.
12. 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.
13. Simon L. Peyton Jones and David R. Lester. *Implementing functional languages.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.
14. 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.
15. Benjamin C. Pierce. *Types and programming languages.* MIT Press, Cambridge, MA, USA, 2002.
16. Morten Heine Sørensen, Robert Glück, and Neil D. Jones. A Positive Supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
17. Valentin F. Turchin. The concept of a supercompiler. *ACM Trans. Program. Lang. Syst.*, 8(3):292–325, 1986.