Development of the Productive Forces

Gavin Mendel-Gleason
Geoff Hamilton

Dublin City University, School of Computing

July, 2012

1. Introduction
2. Type Theory
3. Program Transformation

The Problem

- ```
  CoInductive Nat : Type :=
  | Zero : Nat
  | Succ : Nat -> Nat.

  CoFixpoint plus (x : Nat) (y : Nat) : Nat :=
    match x with
      | Zero => y
      | Succ x' => Succ (plus x' y)
    end.

  Require Import List.

  CoFixpoint sumlen (xs : list Nat) : Nat :=
    match xs with
      | nil => Zero
      | cons x xs' => Succ (plus x (Succ (sumlen xs')))
    end.
  ```
  ```

The Problem

- ```
  CoInductive Nat : Type :=
  | Zero : Nat
  | Succ : Nat -> Nat.

  CoFixpoint plus (x : Nat) (y : Nat) : Nat :=
    match x with
      | Zero => y
      | Succ x' => Succ (plus x' y)
    end.

  Require Import List.

  CoFixpoint sumlen (xs : list Nat) : Nat :=
    match xs with
      | nil => Zero
      | cons x xs' => Succ (plus x (Succ (sumlen xs')))
    end.
  ```
- FAIL!
  ```

The Problem

-
```
CoInductive Sierp : Set :=
| T : Sierp
| DS : Sierp -> Sierp.

CoInductive CoEq : Sierp -> Sierp -> Prop :=
| coeq_base : CoEq T T
| coeq_next : forall x y, CoEq x y -> CoEq (DS x) (DS y).

CoFixpoint join (x : Sierp) (y : Sierp) : Sierp :=
  match x with
    | T => T
    | DS x' => match y with
                 | T => T
                 | DS y' => DS (join x' y')
               end
  end.

Definition exist (xs : Stream A) (P : A -> Sierp) : Sierp :=
  match xs with
    | Cons x xs' => join (P x) (exist xs' P)
  end.
```

The Problem

- ```
  CoInductive Sierp : Set :=
  | T : Sierp
  | DS : Sierp -> Sierp.

  CoInductive CoEq : Sierp -> Sierp -> Prop :=
  | coeq_base : CoEq T T
  | coeq_next : forall x y, CoEq x y -> CoEq (DS x) (DS y).

  CoFixpoint join (x : Sierp) (y : Sierp) : Sierp :=
    match x with
      | T => T
      | DS x' => match y with
                   | T => T
                   | DS y' => DS (join x' y')
                 end
    end.

  Definition exist (xs : Stream A) (P : A -> Sierp) : Sierp :=
    match xs with
      | Cons x xs' => join (P x) (exist xs' P)
    end.
  ```
- FAIL!
  ```

The Problem

```
data Bool : Set where
  true : Bool
  false : Bool

data Nat : Set where
  zero : Nat
  succ : Nat -> Nat

codata Stream (A : Set) : Set where
  _::_ : A -> Stream A -> Stream A

le : Nat -> Nat -> Bool
le zero _ = true
le _ zero = false
le (succ x) (succ y) = le x y

pred : Nat -> Nat
pred zero = zero
pred (succ x) = x

f : Stream Nat -> Stream Nat
f (x :: y :: xs) = if (le x y)
                   then (x :: (f (y :: xs)))
                   else (f ((pred x) :: y :: xs))
```

The Problem

- ```
  data Bool : Set where
    true : Bool
    false : Bool

  data Nat : Set where
    zero : Nat
    succ : Nat -> Nat

  codata Stream (A : Set) : Set where
    _::_ : A -> Stream A -> Stream A

  le : Nat -> Nat -> Bool
  le zero _ = true
  le _ zero = false
  le (succ x) (succ y) = le x y

  pred : Nat -> Nat
  pred zero = zero
  pred (succ x) = x

  f : Stream Nat -> Stream Nat
  f (x :: y :: xs) = if (le x y)
                     then (x :: (f (y :: xs)))
                     else (f ((pred x) :: y :: xs))
  ```
- FAIL!
  ```

The Problem

- What's going on here?

The Problem

- What's going on here?
- The problem is that infinite datatypes are not ok unless you can show that they are actually always going to do something and not sit around computing nothing forever.

The Problem

- What's going on here?
- The problem is that infinite datatypes are not ok unless you can show that they are actually always going to do something and not sit around computing nothing forever.
- Agda, Coq etc. have amazing type systems that let you prove virtually anything.

The Problem

- What's going on here?
- The problem is that infinite datatypes are not ok unless you can show that they are actually always going to do something and not sit around computing nothing forever.
- Agda, Coq etc. have amazing type systems that let you prove virtually anything.
- This means we have to be *very* careful to avoid proofs which are vacuous.

The Problem

- What's going on here?
- The problem is that infinite datatypes are not ok unless you can show that they are actually always going to do something and not sit around computing nothing forever.
- Agda, Coq etc. have amazing type systems that let you prove virtually anything.
- This means we have to be *very* careful to avoid proofs which are vacuous.
- The program

$$f :: forall\ A, A$$
$$f = f$$

is a problem.

Types

- Types are an approach to demonstrating properties by giving evidence.

Types

- Types are an approach to demonstrating properties by giving evidence.
- The Curry-Howard Correspondence keeps the specification and programs tightly coupled.

Types

- Types are an approach to demonstrating properties by giving evidence.
- The Curry-Howard Correspondence keeps the specification and programs tightly coupled.
  - Curry-Howard says:

Types

- Types are an approach to demonstrating properties by giving evidence.
- The Curry-Howard Correspondence keeps the specification and programs tightly coupled.
    - Curry-Howard says: Proofs

Types

- Types are an approach to demonstrating properties by giving evidence.
- The Curry-Howard Correspondence keeps the specification and programs tightly coupled.
  - Curry-Howard says: Proofs $\Leftrightarrow$ Programs

Types

- Types are an approach to demonstrating properties by giving evidence.
- The Curry-Howard Correspondence keeps the specification and programs tightly coupled.
  - Curry-Howard says:

Proofs	$\Leftrightarrow$	Programs
Propositions		

Types

- Types are an approach to demonstrating properties by giving evidence.
- The Curry-Howard Correspondence keeps the specification and programs tightly coupled.
  - Curry-Howard says:

Proofs	$\Leftrightarrow$	Programs
Propositions	$\Leftrightarrow$	Types

Types

- Types are an approach to demonstrating properties by giving evidence.
- The Curry-Howard Correspondence keeps the specification and programs tightly coupled.
  - Curry-Howard says:

Proofs	$\Leftrightarrow$	Programs
Propositions	$\Leftrightarrow$	Types

- Type checking is much easier* than theorem proving and you only need to trust your type checker which reduces the size of the kernel of trust. We don't have to trust the method which generates the proofs.

Types

- Types are an approach to demonstrating properties by giving evidence.
- The Curry-Howard Correspondence keeps the specification and programs tightly coupled.
  - Curry-Howard says:

Proofs	$\Leftrightarrow$	Programs
Propositions	$\Leftrightarrow$	Types

- Type checking is much easier* than theorem proving and you only need to trust your type checker which reduces the size of the kernel of trust. We don't have to trust the method which generates the proofs.
- * Except when it isn't - undecidable type checking etc...

Use supercompilation!

- Program transformers can help type theories know when things are equivalent.

Use supercompilation!

- Program transformers can help type theories know when things are equivalent.
- They can also transform programs which are not type correct into ones that are.

Use supercompilation!

- Program transformers can help type theories know when things are equivalent.
- They can also transform programs which are not type correct into ones that are.
- If we find a systematic way to justify our transformations we can mix type theory and program transformation to correctly type more programs.

Use supercompilation!

- Program transformers can help type theories know when things are equivalent.
- They can also transform programs which are not type correct into ones that are.
- If we find a systematic way to justify our transformations we can mix type theory and program transformation to correctly type more programs.
- *Bisimulation relations* can give us a uniform justification of proof equivalence which leave the program transformation technique up to the implementer as long as they supply the relation.

Types

- Contexts of free variables

$$\Gamma \vdash t : A$$

Types

- Contexts of free variables

$$\boxed{\Gamma} \vdash t : \boxed{A}$$

- A program term

Types

- Contexts of free variables

$$\boxed{\Gamma} \vdash t : \boxed{A}$$

- A program term
- A type which the program satisfies

Types

- Antecedents

$$\dfrac{\begin{array}{c}\vdots \\ \boxed{\Gamma_1 \vdash t_1 : A_1}\end{array} \quad \cdots \quad \begin{array}{c}\vdots \\ \boxed{\Gamma_n \vdash t_n : A_n}\end{array}}{\Gamma \vdash t : A}$$

Types

- Antecedents

$$\frac{\begin{array}{c}\vdots \\ \Gamma_1 \vdash t_1 : A_1\end{array} \quad \cdots \quad \begin{array}{c}\vdots \\ \Gamma_n \vdash t_n : A_n\end{array}}{\Gamma \vdash t : A}$$

- Consequent

Modus Ponens

- Implication

$$\frac{\begin{array}{cc}\vdots & \vdots \\ A \to B & A\end{array}}{B}\ \mathrm{ImpElim}$$

Modus Ponens

- Implication
- Proposition

$$\dfrac{\begin{array}{c}\vdots\\ A \to B\end{array} \qquad \begin{array}{c}\vdots\\ A\end{array}}{B}\ \text{ImpElim}$$

Modus Ponens

- Implication
- Proposition

$$\dfrac{\begin{array}{cc}\vdots & \vdots \\ A \to B & A\end{array}}{B}\ \text{ImpElim}$$

- Consequent

Modus Ponens - AKA: Function Application

- Function
- Proposition
- Consequent

$$\dfrac{\begin{matrix}\vdots \\ \Gamma \vdash f : A \to B\end{matrix} \quad \begin{matrix}\vdots \\ A\end{matrix}}{B}\,\text{ImpElim}$$

Modus Ponens - AKA: Function Application

- Function
- Argument

$$\dfrac{\begin{matrix}\vdots \\ \boxed{\Gamma \vdash f : A \to B}\end{matrix} \quad \begin{matrix}\vdots \\ \boxed{\Gamma \vdash t : A}\end{matrix}}{B}\,\text{ImpElim}$$

- Consequent

Modus Ponens - AKA: Function Application

- Function
- Argument

$$\frac{\begin{array}{c}\vdots\\ \Gamma \vdash f : A \to B\end{array} \qquad \begin{array}{c}\vdots\\ \Gamma \vdash t : A\end{array}}{\Gamma \vdash f\ t : B}\ \text{ImpElim}$$

- Application

Proof

- We arrange proof as a tree with every step justified by rules.

$$\dfrac{\dfrac{\cdots \quad \boxed{\Gamma_{1,m} \vdash t_{1,m} : A_{1,m}}}{\Gamma_1 \vdash t_1 : A_1}\text{Rule}^i \quad \cdots \quad \dfrac{\cdots \quad \Gamma_{1,l} \vdash t_{1,l} : A_{1,l}}{\Gamma_n \vdash t_n : A_n}\text{Rule}^j}{\Gamma \vdash t : A}\text{Rule}^k$$

Proof

- We arrange proof as a tree with every step justified by rules.
- Some rules can be terminal

$$\dfrac{\dfrac{\cdots \quad \boxed{\dfrac{}{\Gamma_{1,m} \vdash t_{1,m} : A_{1,m}}}}{\Gamma_1 \vdash t_1 : A_1}\text{Rule}^i \quad \cdots \quad \dfrac{\cdots \quad \Gamma_{1,l} \vdash t_{1,l} : A_{1,l}}{\Gamma_n \vdash t_n : A_n}\text{Rule}^j}{\Gamma \vdash t : A}\text{Rule}^k$$

Infinite Proof

- Might we want infinite proofs?

Infinite Proof

- Might we want infinite proofs? Yes! If we want infinite objects.

Infinite Proof

- Might we want infinite proofs? Yes! If we want infinite objects.
- Infinite streams for instance...

$$\dfrac{\dfrac{}{\vdash 1 : \mathbb{N}} \quad \dfrac{\dfrac{}{\vdash 2 : \mathbb{N}} \quad \vdots}{\vdash \mathit{map}\ (1+)\ \mathit{nats} : [\mathbb{N}]}\mathit{Scons}}{\vdash \mathit{nats} : [\mathbb{N}]}\mathit{Scons}$$

Infinite Proof with a Finite Presentation

- Infinite, or potentially infinite proofs are not unusual.

Infinite Proof with a Finite Presentation

- Infinite, or potentially infinite proofs are not unusual.
- Functional programs with recursion make use of such proofs implicitly.

Infinite Proof with a Finite Presentation

- Infinite, or potentially infinite proofs are not unusual.
- Functional programs with recursion make use of such proofs implicitly.
- The finite presentation is usually achieved by way of a recursive type rule.

$$\frac{f : A, \Gamma \vdash Body(f) : A}{\Gamma \vdash f : A} FunRule$$

Infinite Proof with a Finite Presentation

- Infinite, or potentially infinite proofs are not unusual.
- Functional programs with recursion make use of such proofs implicitly.
- The finite presentation is usually achieved by way of a recursive type rule.

$$\frac{f : A, \Gamma \vdash Body(f) : A}{\Gamma \vdash f : A}\,FunRule$$

Recursive Function Rule

Two big problems here

Recursive Function Rule

Two big problems here

- You can get unsound proofs easily.

Recursive Function Rule

Two big problems here

- You can get unsound proofs easily. e.g. $Body(\omega) = \omega$

$$\dfrac{\dfrac{\omega : A \vdash \omega : A}{\omega : A, \Gamma \vdash Body(\omega) : A}}{\Gamma \vdash \omega : A}\,FunRule$$

Recursive Function Rule

Two big problems here

- You can get unsound proofs easily. e.g. $Body(\omega) = \omega$

$$\dfrac{\dfrac{\omega : A \vdash \omega : A}{\omega : A, \Gamma \vdash Body(\omega) : A}}{\Gamma \vdash \omega : A}\,FunRule$$

- You are stuck with the recursive form given by your recursive functions.

Recursive Function Rule

Two big problems here

- You can get unsound proofs easily. e.g. $Body(\omega) = \omega$

$$\cfrac{\cfrac{\omega : A \vdash \omega : A}{\omega : A, \Gamma \vdash Body(\omega) : A}}{\Gamma \vdash \omega : A} FunRule$$

- You are stuck with the recursive form given by your recursive functions. This will restrict how we can transform proofs (programs!)

Cyclic Proofs

Solution?

Cyclic Proofs

Solution? Cyclic Proof!

Cyclic Proofs

Solution? Cyclic Proof!

- Gives finite presentations of infinite proofs.

Cyclic Proofs

Solution? Cyclic Proof!

- Gives finite presentations of infinite proofs.
- Doesn't privilege a particular recursive structure.

Cyclic Proofs

Solution? Cyclic Proof!

- Gives finite presentations of infinite proofs.
- Doesn't privilege a particular recursive structure.
- By some coincidence it looks just the result of a supercompiler.

Cyclic Proofs

Solution? Cyclic Proof!

- Gives finite presentations of infinite proofs.
- Doesn't privilege a particular recursive structure.
- By some coincidence it looks just the result of a supercompiler.
- We can transform the proof, retaining bisimilarity with our original proof and introducing our own cycles where we like.*

Cyclic Proofs

Solution? Cyclic Proof!

- Gives finite presentations of infinite proofs.
- Doesn't privilege a particular recursive structure.
- By some coincidence it looks just the result of a supercompiler.
- We can transform the proof, retaining bisimilarity with our original proof and introducing our own cycles where we like.*
- *Provided we are careful to put in a condition which ensure soundness.

Cyclic Proofs

$$\dfrac{n : \mathbb{N} \vdash n : \mathbb{N} \quad m : \mathbb{N} \vdash m : \mathbb{N} \quad \dfrac{\dfrac{\begin{array}{l}\text{case } n' \text{ of}\\ \quad 0 \Rightarrow m\\ \quad \mid S\ n'' \Rightarrow S\ (plus\ n''\ m)\end{array} \quad : \mathbb{N}}{n' : \mathbb{N}, m : \mathbb{N} \vdash plus\ n'\ m : \mathbb{N}}}{n' : \mathbb{N}, m : \mathbb{N} \vdash S\ (plus\ n'\ m) : \mathbb{N}}}{n : \mathbb{N}, m : \mathbb{N} \vdash \boxed{\begin{array}{l}\text{case } n \text{ of}\\ \quad 0 \Rightarrow m\\ \quad \mid S\ n' \Rightarrow S\ (plus\ n'\ m)\end{array}} : \mathbb{N}}$$

Cyclic Proofs

$$
\dfrac{n : \mathbb{N} \vdash n : \mathbb{N} \quad m : \mathbb{N} \vdash m : \mathbb{N} \quad \dfrac{\dfrac{\boxed{\begin{array}{l} \text{case } n' \text{ of} \\ \quad 0 \Rightarrow m \\ \quad \mid S\ n'' \Rightarrow S\ (plus\ n''\ m) \end{array}} : \mathbb{N}}{n' : \mathbb{N}, m : \mathbb{N} \vdash plus\ n'\ m : \mathbb{N}}}{n' : \mathbb{N}, m : \mathbb{N} \vdash S\ (plus\ n'\ m) : \mathbb{N}}}{n : \mathbb{N}, m : \mathbb{N} \vdash \boxed{\begin{array}{l} \text{case } n \text{ of} \\ \quad 0 \Rightarrow m \\ \quad \mid S\ n' \Rightarrow S\ (plus\ n'\ m) \end{array}} : \mathbb{N}}
$$

Cyclic Proofs

$$\dfrac{n : \mathbb{N} \vdash n : \mathbb{N} \quad m : \mathbb{N} \vdash m : \mathbb{N} \quad \dfrac{\dfrac{\boxed{\begin{array}{l}\text{case } n' \text{ of}\\ \quad 0 \Rightarrow m \\ \quad \mid S\ n'' \Rightarrow S\ (plus\ n''\ m)\end{array}} : \mathbb{N}}{n' : \mathbb{N}, m : \mathbb{N} \vdash plus\ n'\ m : \mathbb{N}}}{n' : \mathbb{N}, m : \mathbb{N} \vdash S\ (plus\ n'\ m) : \mathbb{N}}}{n : \mathbb{N}, m : \mathbb{N} \vdash \boxed{\begin{array}{l}\text{case } n \text{ of}\\ \quad 0 \Rightarrow m \\ \quad \mid S\ n' \Rightarrow S\ (plus\ n'\ m)\end{array}} : \mathbb{N}}$$

Cyclic Proofs

$$
\dfrac{n : \mathbb{N} \vdash n : \mathbb{N} \quad m : \mathbb{N} \vdash m : \mathbb{N} \quad \dfrac{\dfrac{\boxed{\begin{array}{l} \text{case } n' \text{ of} \\ \quad 0 \Rightarrow m \\ \quad \mid S\ n'' \Rightarrow S\ (plus\ n''\ m) \end{array}} : \mathbb{N}}{n' : \mathbb{N}, m : \mathbb{N} \vdash plus\ n'\ m : \mathbb{N}}}{n' : \mathbb{N}, m : \mathbb{N} \vdash S\ (plus\ n'\ m) : \mathbb{N}}}{n : \mathbb{N}, m : \mathbb{N} \vdash \boxed{\begin{array}{l} \text{case } n \text{ of} \\ \quad 0 \Rightarrow m \\ \quad \mid S\ n' \Rightarrow S\ (plus\ n'\ m) \end{array}} : \mathbb{N}}
$$

$\theta := \{(n', n)\}$

(Co)-Termination Requirements

- Ultimately we want to know that our cyclic proofs do not allow computations with no behaviour. (e.g. ω).

(Co)-Termination Requirements

- Ultimately we want to know that our cyclic proofs do not allow computations with no behaviour. (e.g. ω).
- In order to avoid this with cyclic proof the proof formation rules are not sufficient.

(Co)-Termination Requirements

- Ultimately we want to know that our cyclic proofs do not allow computations with no behaviour. (e.g. ω).
- In order to avoid this with cyclic proof the proof formation rules are not sufficient.
- Something needs to be decreasing for every inductive cycle.

(Co)-Termination Requirements

- Ultimately we want to know that our cyclic proofs do not allow computations with no behaviour. (e.g. ω).
- In order to avoid this with cyclic proof the proof formation rules are not sufficient.
- Something needs to be decreasing for every inductive cycle.
- Some behaviour needs to be ensured for every coinductive cycle.

Inductive Requirements

- Every cycle must have a structurally smaller term.

Inductive Requirements

- Every cycle must have a structurally smaller term.
- Every cycle returning to the same node must have the same structrally smaller term.

$$\dfrac{\dfrac{\Gamma' \vdash \boxed{r : B}}{\vdots} \qquad \dfrac{\Gamma'' \vdash \boxed{s : C}}{\vdots}}{\Gamma \vdash \boxed{t : A}}$$

Inductive Requirements

- Every cycle must have a structurally smaller term.
- Every cycle returning to the same node must have the same structrally smaller term.

$$\dfrac{\dfrac{\Gamma' \vdash r : B}{\vdots} \qquad \dfrac{\Gamma'' \vdash s : C}{\vdots}}{\Gamma \vdash t : A}$$

Inductive Requirements

- Every cycle must have a structurally smaller term.
- Every cycle returning to the same node must have the same structrally smaller term.

$$\dfrac{\dfrac{\Gamma' \vdash \boxed{r : B}}{\vdots} \qquad \dfrac{\Gamma'' \vdash \boxed{s : C}}{\vdots}}{\theta \vdash \boxed{t : A} \qquad \theta'}$$

Coinductive Requirements

- The *path* in every coinductive cycle is constrained.

Coinductive Requirements

- The *path* in every coinductive cycle is constrained.
- By a *path* we mean which antecedents we choose to obtain a sequent starting at some root.

Coinductive Requirements

- The *path* in every coinductive cycle is constrained.
- By a *path* we mean which antecedents we choose to obtain a sequent starting at some root.
- The following path: AndIntro^2, OrIntroL^1, chooses the 2nd, and 1st antecedents respectively.

$$\dfrac{C \qquad \dfrac{A}{(A \vee B)}\,\text{OrIntroL}}{C \wedge (A \vee B)}\,\text{AndIntro}$$

Coinductive Requirements

- The *path* in every coinductive cycle is constrained.
- By a *path* we mean which antecedents we choose to obtain a sequent starting at some root.
- The following path: AndIntro^2, OrIntroL^1, chooses the 2nd, and 1st antecedents respectively.

$$\dfrac{C \qquad \dfrac{A}{(A \vee B)}\text{OrIntroL}}{C \wedge (A \vee B)}\text{AndIntro}$$

- A restriction on the form of paths ensures that we can not have *non-productive* computation. That is, all terms will produce some constructor eventually.

Coinductive Requirements

Coinductive Requirements

- The restriction is made up of two parts, accessible paths, and guarded paths.

Coinductive Requirements

- The restriction is made up of two parts, accessible paths, and guarded paths.
- Definition (Admissible). A path is called admissible if the first element c of the path $p = c, p$ is one of the rule-index-pairs $\mathrm{OrIntroL}^1$, $\mathrm{OrIntroR}^1$, $\mathrm{AndIntro}^1$, $\mathrm{AndIntro}^2$, $\mathrm{AllIntro}^1$, $\alpha\mathrm{Intro}^1$, $\mathrm{ImpIntro}^1$, OrElim^2, OrElim^3, $\mathrm{AndElim}^2$, $\mathrm{AllElim}^1$, Delta^1 and p is an admissible path.

Coinductive Requirements

- The restriction is made up of two parts, accessible paths, and guarded paths.
- Definition (Admissible). A path is called admissible if the first element c of the path $p = c, p$ is one of the rule-index-pairs $\mathrm{OrIntroL}^1$, $\mathrm{OrIntroR}^1$, $\mathrm{AndIntro}^1$, $\mathrm{AndIntro}^2$, $\mathrm{AllIntro}^1$, $\alpha\mathrm{Intro}^1$, $\mathrm{ImpIntro}^1$, OrElim^2, OrElim^3, $\mathrm{AndElim}^2$, $\mathrm{AllElim}^1$, Delta^1 and p is an admissible path.
- Definition (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 $\mathrm{OrIntroL}^1$, $\mathrm{OrIntroR}^1$, $\mathrm{AndIntro}^1$, $\mathrm{AndIntro}^2$, $\nu\mathrm{Intro}^1$, $\mathrm{ImpIntro}^1$ which we will call guards and p and p are admissible paths.

Program Transformation and Cyclic Proof

- Program transformations such as Deforestation and Supercompilation exist naturally in the setting of cyclic proof.

Program Transformation and Cyclic Proof

- Program transformations such as Deforestation and Supercompilation exist naturally in the setting of cyclic proof.
  - Decending into term structure is simply applying transformation techniques to anticedents.

Program Transformation and Cyclic Proof

- Program transformations such as Deforestation and Supercompilation exist naturally in the setting of cyclic proof.
  - Decending into term structure is simply applying transformation techniques to anticedents.
  - Evaluation steps are always justified.

Program Transformation and Cyclic Proof

- Program transformations such as Deforestation and Supercompilation exist naturally in the setting of cyclic proof.
  - Decending into term structure is simply applying transformation techniques to anticedents.
  - Evaluation steps are always justified.
- We should deem any transformation as appropriate if the resulting term is bisimilar to the original proof.

Program Transformation and Cyclic Proof

- Program transformations such as Deforestation and Supercompilation exist naturally in the setting of cyclic proof.
  - Decending into term structure is simply applying transformation techniques to anticedents.
  - Evaluation steps are always justified.
- We should deem any transformation as appropriate if the resulting term is bisimilar to the original proof.
  - Information propagation.

Program Transformation and Cyclic Proof

- Program transformations such as Deforestation and Supercompilation exist naturally in the setting of cyclic proof.
  - Decending into term structure is simply applying transformation techniques to anticedents.
  - Evaluation steps are always justified.
- We should deem any transformation as appropriate if the resulting term is bisimilar to the original proof.
  - Information propagation.
  - Simplification rules:

$$
\begin{array}{l}
\text{case} \;\; \begin{array}{l}\text{case } t \text{ of} \\ \quad\;\; x \Rightarrow r \\ \quad\;\; |\; y \Rightarrow s\end{array} \;\; \text{of} \\
\quad w \Rightarrow t \\
\quad |\; v \Rightarrow u
\end{array}
\;\sim\;
\begin{array}{l}
\text{case } t \text{ of} \\
\quad x \Rightarrow \;\; \begin{array}{l}\text{case } r \text{ of} \\ \quad w \Rightarrow t \\ \quad |\; v \Rightarrow u\end{array} \\
\quad |\; y \Rightarrow \;\; \begin{array}{l}\text{case } s \text{ of} \\ \quad w \Rightarrow t \\ \quad |\; v \Rightarrow u\end{array}
\end{array}
$$

Program Transformation and Cyclic Proof

- Critically, bisimilar program transformation does not care about non-termination, but it respects it!

Program Transformation and Cyclic Proof

- Critically, bisimilar program transformation does not care about non-termination, but it respects it!
- We will neither eliminate nor introduce non-termination.

Program Transformation and Cyclic Proof

- Critically, bisimilar program transformation does not care about non-termination, but it respects it!
- We will neither eliminate nor introduce non-termination.
- This is important because we want to establish that our programs (co)-terminate *later*, after transformation.

Program Transformation

- The program transformer is *multi-result* - we have a stream of transformed programs.

Program Transformation

- The program transformer is *multi-result* - we have a stream of transformed programs.
- The stream is filtered by the constraint on the syntax.

Program Transformation

- The program transformer is *multi-result* - we have a stream of transformed programs.
- The stream is filtered by the constraint on the syntax.
- Since the stream is *lazy* we prune branches which will not meet our syntactic restrictions as they are constructed.

Program Transformation

- The program transformer is *multi-result* - we have a stream of transformed programs.
- The stream is filtered by the constraint on the syntax.
- Since the stream is *lazy* we prune branches which will not meet our syntactic restrictions as they are constructed.
- The streams are implemented with the ω-Monad, a monad which handles the book-keeping of manipulating (potentially) infinite streams.

Success

- *mutual*
  $sumlen\_sc : CoList\ CoNat \rightarrow CoNat$
  $sumlen\_sc\ [\,] = czero$
  $sumlen\_sc\ (x :: xs) = csucc\ (aux\ x\ xs)$

  $aux : CoNat \rightarrow CoList\ CoNat \rightarrow CoNat$
  $aux\ czero\ xs = sumlen\_sc\ xs$
  $aux\ (csucc\ x)\ xs = csucc\ (aux\ x\ xs)$

Success

- *mutual*
  $sumlen\_sc : CoList\ CoNat \to CoNat$
  $sumlen\_sc\ [\,] = czero$
  $sumlen\_sc\ (x :: xs) = csucc\ (aux\ x\ xs)$

  $aux : CoNat \to CoList\ CoNat \to CoNat$
  $aux\ czero\ xs = sumlen\_sc\ xs$
  $aux\ (csucc\ x)\ xs = csucc\ (aux\ x\ xs)$

- Success!

Success?

- ```
  Definition exist (xs : Stream A) (P : A -> Sierp) : Sierp :=
    match xs with
      | Cons x xs' => join (P x) (exist xs' P)
    end.
  ```
  ```

Success?

- ```
  Definition exist (xs : Stream A) (P : A -> Sierp) : Sierp :=
    match xs with
      | Cons x xs' => join (P x) (exist xs' P)
    end.
  ```
- FAIL!
  ```

Success?

- ```
  Definition exist (xs : Stream A) (P : A -> Sierp) : Sierp :=
    match xs with
      | Cons x xs' => join (P x) (exist xs' P)
    end.
  ```
- FAIL!
- Why? Need a result about the associativity of *join*.
  ```

Success?

- ```
  Definition exist (xs : Stream A) (P : A -> Sierp) : Sierp :=
   match xs with
     | Cons x xs' => join (P x) (exist xs' P)
   end.
  ```
- FAIL!
- Why? Need a result about the associativity of *join*.
- But supercompilation can do this! Need to use the *right* generalisation and supercompilation on args.
  ```

Success?

- ```
  Definition exist (xs : Stream A) (P : A -> Sierp) : Sierp :=
    match xs with
      | Cons x xs' => join (P x) (exist xs' P)
    end.
  ```
- FAIL!
- Why? Need a result about the associativity of *join*.
- But supercompilation can do this! Need to use the *right* generalisation and supercompilation on args.
- Justifiable only if the type theory internally supports bisimulation.
  ```

Future Work

- A framework for manipulating a more practical programming language (such as Haskell or Agda).

Future Work

- A framework for manipulating a more practical programming language (such as Haskell or Agda).
- Extension to a type theory with explicit substition for bisimilar terms.

Future Work

- A framework for manipulating a more practical programming language (such as Haskell or Agda).
- Extension to a type theory with explicit substition for bisimilar terms.
- A system for the programmers to interactively transform productive terms into syntactically productive terms by using substitution.

The End

The End

Bisimulation

- To use program transformation with type theory we need justifications that our program transformations are correct.

Bisimulation

- To use program transformation with type theory we need justifications that our program transformations are correct.
- Bisimulations allow us to show equivalences even when we might have infinite behaviours.

Bisimulation

- To use program transformation with type theory we need justifications that our program transformations are correct.
- Bisimulations allow us to show equivalences even when we might have infinite behaviours.
- Relies on a *coinductive* relation between terms.

Bisimulation

- To use program transformation with type theory we need justifications that our program transformations are correct.
- Bisimulations allow us to show equivalences even when we might have infinite behaviours.
- Relies on a *coinductive* relation between terms.
- We can generate them in the process of program transformation.

Bisimulation

- To use program transformation with type theory we need justifications that our program transformations are correct.
- Bisimulations allow us to show equivalences even when we might have infinite behaviours.
- Relies on a *coinductive* relation between terms.
- We can generate them in the process of program transformation.
- If two terms s and t are bisimilar we write $s \sim t$

Coinduction

- With induction we want to describe a property over each constructor assuming the sub-case.

Coinduction

- With induction we want to describe a property over each constructor assuming the sub-case.
  e.g. the integers, prove $P\ 0 \wedge P\ n \rightarrow P(n+1)$, to get $P\ n$

Coinduction

- With induction we want to describe a property over each constructor assuming the sub-case.
  e.g. the integers, prove $P\ 0 \wedge P\ n \rightarrow P(n+1)$, to get $P\ n$
- With coinduction we want to describe a property over each destructor assuming the super-case.

Coinduction

- With induction we want to describe a property over each constructor assuming the sub-case.
  e.g. the integers, prove $P\ 0 \wedge P\ n \rightarrow P(n+1)$, to get $P\ n$
- With coinduction we want to describe a property over each destructor assuming the super-case.
  e.g. streams, prove $P\ l \rightarrow\ P\ (\mathit{head}\ l) \wedge P\ (\mathit{tail}\ l)$ to get $P\ l$

Parks' Principle

- Useful example of a coinductively defined relation

Parks' Principle

- Useful example of a coinductively defined relation
- When are two infinite streams the same?

Parks' Principle

- Useful example of a coinductively defined relation
- When are two infinite streams the same? When every element is the same...

Parks' Principle

- Useful example of a coinductively defined relation
- When are two infinite streams the same? When every element is the same...

$l \; R \; l'$

Parks' Principle

- Useful example of a coinductively defined relation
- When are two infinite streams the same? When every element is the same...

$l \; R \; l'$

$$\text{if} \quad head\ l \quad = \quad head\ l'$$

Parks' Principle

- Useful example of a coinductively defined relation
- When are two infinite streams the same? When every element is the same...

$l \; R \; l'$

$$\begin{array}{rlcl} \text{if} & head\ l & = & head\ l' \\ \text{and} & tail\ l & R & tail\ l' \end{array}$$

Parks' Principle

- Useful example of a coinductively defined relation
- When are two infinite streams the same? When every element is the same...

$l \; R \; l'$

$$
\begin{array}{rlcl}
\text{if} & head\ l & = & head\ l' \\
\text{and} & tail\ l & R & tail\ l' \text{ assuming } l \; R \; l'
\end{array}
$$

Simulation

Bisimulation is formed from two coinductively defined simulation relations:

Simulation

Bisimulation is formed from two coinductively defined simulation relations:

- $s \lesssim t$

Simulation

Bisimulation is formed from two coinductively defined simulation relations:

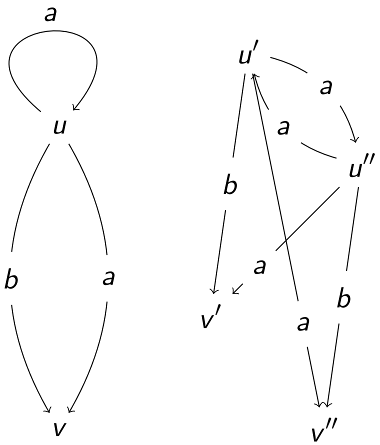
- $s \lesssim t \wedge t \lesssim s$

Simulation

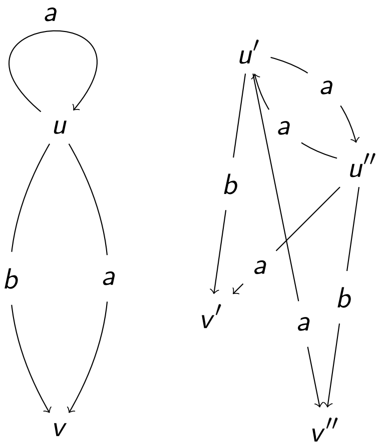
Bisimulation is formed from two coinductively defined simulation relations:

- $s \lesssim t \wedge t \lesssim s$
- $s \lesssim t$ says that whenever $s \xrightarrow{a} s'$ then $t \xrightarrow{a} t'$ and $s' \lesssim t'$

Example Bisimulation

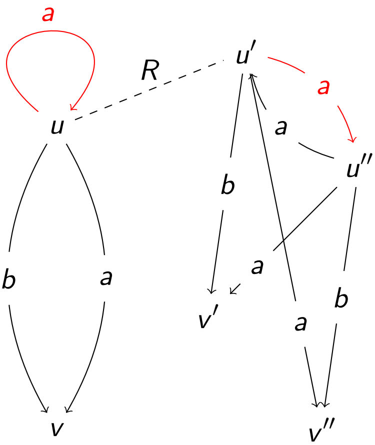


Example Bisimulation



We show one direction of simulation...

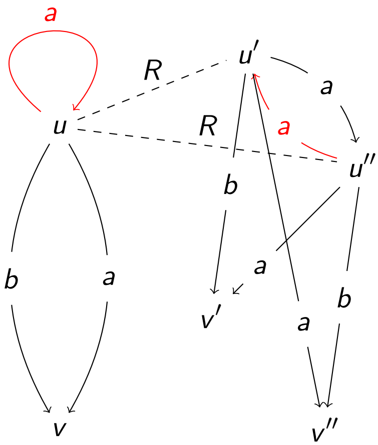
Example Bisimulation



We show one direction of simulation...

- when $u \xrightarrow{a} u$ we choose $u' \xrightarrow{a} u''$ and need to show $u \; R \; u''$ assuming $u \; R \; u'$

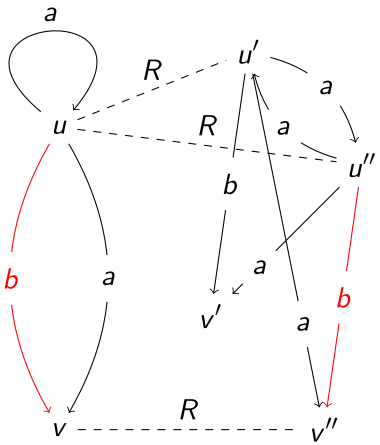
Example Bisimulation



We show one direction of simulation...

- when $u \xrightarrow{a} u$ we choose $u' \xrightarrow{a} u''$ and need to show $u \; R \; u''$ assuming $u \; R \; u'$
- when $u \xrightarrow{a} u$ we choose $u'' \xrightarrow{a} u'$ and need to show $u \; R \; u'$ (done!)

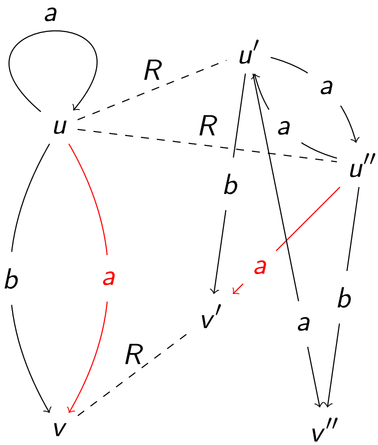
Example Bisimulation



We show one direction of simulation...

- when $u \xrightarrow{a} u$ we choose $u' \xrightarrow{a} u''$ and need to show $u\ R\ u''$ assuming $u\ R\ u'$
- when $u \xrightarrow{b} v$ we choose $u'' \xrightarrow{b} v''$ (done! no further behaviour)

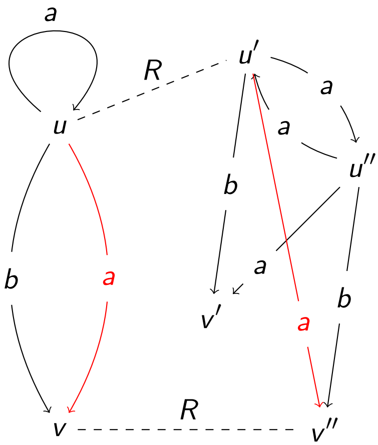
Example Bisimulation



We show one direction of simulation...

- when $u \xrightarrow{a} u$ we choose $u' \xrightarrow{a} u''$ and need to show $u\ R\ u''$ assuming $u\ R\ u'$
- when $u \xrightarrow{a} v$ we choose $u' \xrightarrow{a} v'$ (done! no further behaviour)

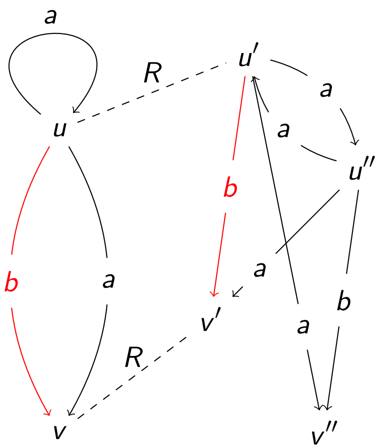
Example Bisimulation



We show one direction of simulation...

- when $u \xrightarrow{a} v$ we choose $u' \xrightarrow{a} v''$ (done! no further behaviour)

Example Bisimulation



We show one direction of simulation...

- when $u \xrightarrow{b} v$ we choose $u' \xrightarrow{b} v'$ (done! no further behaviour)

Term Transition Systems

Can we use this for a transition system for terms?

Term Transition Systems

Can we use this for a transition system for terms?

- Yes!

Term Transition Systems

Can we use this for a transition system for terms?

- Yes! We can now look at (bi)similarity relations over terms.

Term Transition Systems

Can we use this for a transition system for terms?

- Yes! We can now look at (bi)similarity relations over terms.
- We can look at Parks' principle again, with l a stream of As, $l : [A]$

Term Transition Systems

Can we use this for a transition system for terms?

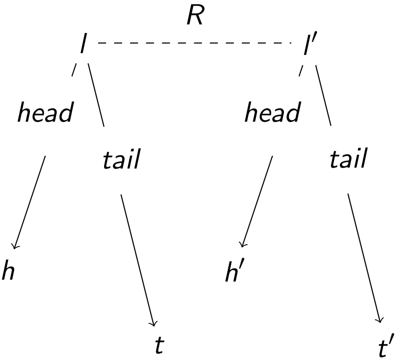
- Yes! We can now look at (bi)similarity relations over terms.
- We can look at Parks' principle again, with l a stream of As, $l : [A]$

$$\begin{tikzcd} & l \arrow[dl, "head"'] \arrow[dr, "tail"] & \\ h & & t \end{tikzcd} \qquad \begin{tikzcd} & l' \arrow[dl, "head"'] \arrow[dr, "tail"] & \\ h' & & t' \end{tikzcd}$$

Term Transition Systems

Can we use this for a transition system for terms?

- Yes! We can now look at (bi)similarity relations over terms.
- We can look at Parks' principle again, with l a stream of As, $l : [A]$



Term Transition Systems

Can we use this for a transition system for terms?

- Yes! We can now look at (bi)similarity relations over terms.
- We can look at Parks' principle again, with l a stream of As, $l : [A]$

$$
\begin{tikzcd}
l \arrow[rr, dash, dashed, "R"] \arrow[d, "head"'] \arrow[ddr, "tail"] & & l' \arrow[d, "head"'] \arrow[ddr, "tail"] & \\
h \arrow[rr, dashed, "="] & & h' & \\
& t & & t'
\end{tikzcd}
$$

Term Transition Systems

Can we use this for a transition system for terms?

- Yes! We can now look at (bi)similarity relations over terms.
- We can look at Parks' principle again, with l a stream of As, $l : [A]$

$$
\begin{tikzcd}
 & l \arrow[rr, dashed, no head, "R"] \arrow[dl, "head"'] \arrow[dd, "tail"] & & l' \arrow[dl, "head"'] \arrow[dd, "tail"] \\
h \arrow[rr, dashed, "="] & & h' & \\
 & t \arrow[rr, dashed, no head, "R"] & & t'
\end{tikzcd}
$$

Term Transition Systems

How do we know which transition we need?

Term Transition Systems

How do we know which transition we need?

- Use a structural operational semantics to define transitions.

Term Transition Systems

How do we know which transition we need?

- Use a structural operational semantics to define transitions.
- Each transition corresponds with an *experiment* which we obtain from the term language and evaluation relation.

Term Transition Systems

How do we know which transition we need?

- Use a structural operational semantics to define transitions.
- Each transition corresponds with an *experiment* which we obtain from the term language and evaluation relation.
- Experiments consist of terms which will lead to a reduction.

Term Transition Systems

How do we know which transition we need?

- Use a structural operational semantics to define transitions.
- Each transition corresponds with an *experiment* which we obtain from the term language and evaluation relation.
- Experiments consist of terms which will lead to a reduction.

  Application $[\,]\ c$

Term Transition Systems

How do we know which transition we need?

- Use a structural operational semantics to define transitions.
- Each transition corresponds with an *experiment* which we obtain from the term language and evaluation relation.
- Experiments consist of terms which will lead to a reduction.

Application	[ ] c
Type Application	[ ] A

Term Transition Systems

How do we know which transition we need?

- Use a structural operational semantics to define transitions.
- Each transition corresponds with an *experiment* which we obtain from the term language and evaluation relation.
- Experiments consist of terms which will lead to a reduction.

Application	$[\,]\ c$
Type Application	$[\,]\ A$
Case Elimination	$\mathrm{case}[\,]\ \mathrm{of}\ \{\ nil\ \Rightarrow\ t \mid (x : xs) \Rightarrow\ s\}$

Term Transition Systems

How do we know which transition we need?

- Use a structural operational semantics to define transitions.
- Each transition corresponds with an *experiment* which we obtain from the term language and evaluation relation.
- Experiments consist of terms which will lead to a reduction.

Application	$[\,]\ c$
Type Application	$[\,]\ A$
Case Elimination	$\text{case}[\,]\ \text{of}\ \{\ nil \ \Rightarrow\ t \mid (x : xs) \Rightarrow\ s\}$
Pair Elimination	$\text{split}[\,]\ \text{as}\ (x, y)\ \text{in}\ s$

Term Transition Systems

What about function terms?

Term Transition Systems

What about function terms?

- Yes!

Term Transition Systems

What about function terms?

- Yes! Use a test value of the appropriate type to look at behaviour

Term Transition Systems

What about function terms?

- Yes! Use a test value of the appropriate type to look at behaviour
- If $f : A \to B$

Term Transition Systems

What about function terms?

- Yes! Use a test value of the appropriate type to look at behaviour
- If $f : A \rightarrow B$ then $f \xrightarrow{@c:A} f\ c$

Term Transition Systems

What about function terms?

- Yes! Use a test value of the appropriate type to look at behaviour
- If $f : A \rightarrow B$ then $f \xrightarrow{@c:A} f\ c$
- $f \sim g$?

Term Transition Systems

What about function terms?

- Yes! Use a test value of the appropriate type to look at behaviour
- If $f : A \rightarrow B$ then $f \xrightarrow{@c:A} f\ c$
- $f \sim g$? if whenever $f \xrightarrow{@c:A} f\ c$ and $g \xrightarrow{@c:A} g\ c$ then $f\ c \sim g\ c$
- This approach retains *extensionality*. That is two functions are the same if they are the same when called with the same arguments.