Automated Theorem Proving for Type Inference, Constructively

Ekaterina Komentdantskaya[1], joint work with Peng Fu[1] and Tom Schrijvers[2]

[1]Heriot-Watt University, Edinburgh; [2]Leuven University

Programming Languages and Compilers, April 2017

Outline

(Motivation) Verification methods: the good, the bad and the ugly

Outline

(Motivation) Verification methods: the good, the bad and the ugly

(Background) Proof-carrying code, revisited

Outline

(Motivation) Verification methods: the good, the bad and the ugly

(Background) Proof-carrying code, revisited

(Technical Contribution) Going beyond state-of-the art: corecursion in type inference

Outline

(Motivation) Verification methods: the good, the bad and the ugly

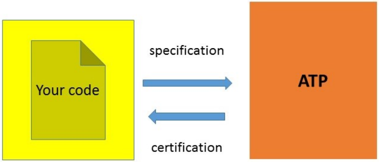
(Background) Proof-carrying code, revisited

(Technical Contribution) Going beyond state-of-the art: corecursion in type inference

(Conclusion) New type inference recipe: tastes good, does good

Two styles of verification

Algorithmic



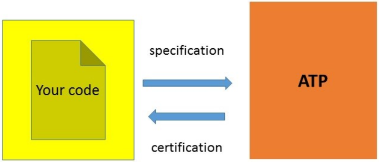
Problems:

- do we trust the specification?
- do we trust the ATP? (itself not verified)

(E.g. SMT solvers – 100K lines of C++ code)

Two styles of verification

Algorithmic

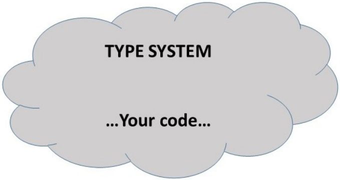


Problems:

- do we trust the specification?
- do we trust the ATP? (itself not verified)

(E.g. SMT solvers – 100K lines of C++ code)

Typeful



- Curry-Howard style of verification
- Proofs are functions that we can re-run and check independently

Type Computation

In typed functional languages and constructive theorem provers, to prove that a context Γ entails theorem A, we need to construct proof p as an inhabitant of type A.

$$\Gamma \vdash p : A$$

Type Computation

In typed functional languages and constructive theorem provers, to prove that a context Γ entails theorem A, we need to construct proof p as an inhabitant of type A.

$$\Gamma \vdash p : A$$

Type computation problems:
$\Gamma \vdash p : A?$ – Type Checking;
$\Gamma \vdash p :?$ – Type Inference;
$\Gamma \vdash ? : A$ – Type Inhabitation.

Type Computation

In typed functional languages and constructive theorem provers, to prove that a context Γ entails theorem A, we need to construct proof p as an inhabitant of type A.

$$\Gamma \vdash p : A$$

Type computation problems:
$\Gamma \vdash p : A?$ – Type Checking;
$\Gamma \vdash p :?$ – Type Inference;
$\Gamma \vdash ? : A$ – Type Inhabitation.
The latter is facilitated by tactic languages in ITP. This talk is about type inhabitation, too.

All three are sometimes known under the name of “type inference", I'll use this terminology, too.

When types get rich...

- they can be as expressive as our best ATPs
- e.g. they can encode pre- and post-conditions
- e.g. they can incorporate reasoning on first-order theories

When types get rich...

- they can be as expressive as our best ATPs
- e.g. they can encode pre- and post-conditions
- e.g. they can incorporate reasoning on first-order theories

Examples: refinement types, Liquid Haskell, F* – directly mimic pre- and post-condition specifications and call SMT solvers to do the solving

Detour: Liquid Haskell example: Well typed programs can go wrong

Consider the following well-typed program in Haskell, that defines a function to compute the average of all elements in a list:

```
average :: [Int] -> Int
average xs  = sum xs `div` length xs
```

Detour: Liquid Haskell example: Well typed programs can go wrong

Consider the following well-typed program in Haskell, that defines a function to compute the average of all elements in a list:

```
average :: [Int] -> Int
average xs  = sum xs `div` length xs
```

We get the desired response on any non-empty list of integers, but get an exception calling

```
ghci > average []
*** Exception: divide by zero
```

Solution: a more fine-grained annotations on types

The refinement type $\{\mathtt{x} : \mathtt{int} \mid \mathtt{x} < \mathtt{100}\}\ \mathtt{list}$ describes a list of integers each of which is smaller than 100. Refinements can express sophisticated data structure invariants.

Solution: a more fine-grained annotations on types

The refinement type $\{\mathtt{x} : \mathtt{int} \mid \mathtt{x} < \mathtt{100}\}\ \mathtt{list}$ describes a list of integers each of which is smaller than 100. Refinements can express sophisticated data structure invariants.
To fix the previous example, we need to refine the type $[Int]$, prohibiting input lists of zero length.

Another code example

Suppose we define a function that computes absolute values of integers:

```
abs   :: Int -> Int
abs n
  | 0 < n = n
  | otherwise = 0 - n
```

We can use refinement types to infer post-conditions, e.g. to infer that the function returns non-negative values. For example, Liquid Haskell will be able to refine the type of `abs` as follows:
$\{-@\ \texttt{abs} :: \texttt{Int} \rightarrow \texttt{Nat}\ @-\}$ where `Nat` is defined as
$\{-@\ \texttt{typeNat} = \{\texttt{v} : \texttt{Int} | 0 \Leftarrow \texttt{v}\}\ @-\}$.
This precise inference is possible because SMT solvers have built-in decision procedures for arithmetic.

When types get rich...

- they can be as expressive as our best ATPs
- e.g. they can encode pre- and post-conditions
- e.g. they can incorporate reasoning on first-order theories

When types get rich...

- they can be as expressive as our best ATPs
- e.g. they can encode pre- and post-conditions
- e.g. they can incorporate reasoning on first-order theories

Examples: refinement types, Liquid Haskell, F* – directly mimic pre- and post-condition specifications

- type inference calls SMT solvers to solve, check and refine the specifications given in types

When types get rich...

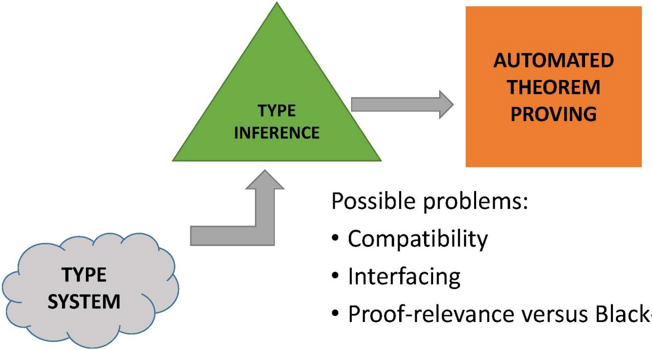
- they can be as expressive as our best ATPs
- e.g. they can encode pre- and post-conditions
- e.g. they can incorporate reasoning on first-order theories

Examples: refinement types, Liquid Haskell, F* – directly mimic pre- and post-condition specifications

- type inference calls SMT solvers to solve, check and refine the specifications given in types

Good cause! where these methods belong in our big picture?

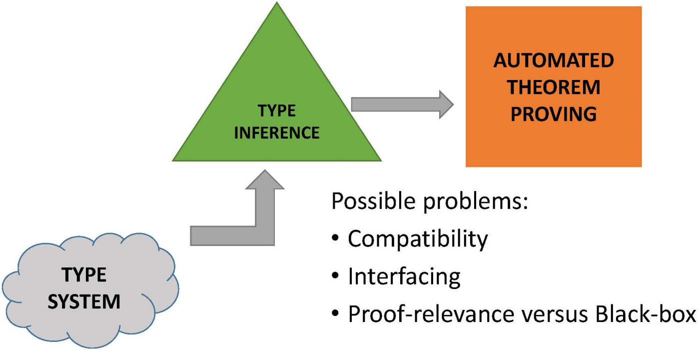
A trend in typed language development



Possible problems:

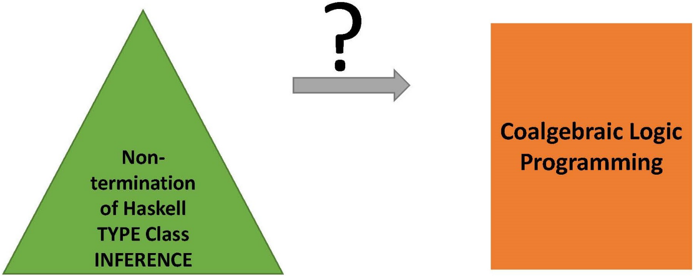
- Compatibility
- Interfacing
- Proof-relevance versus Black-box

A trend in typed language development

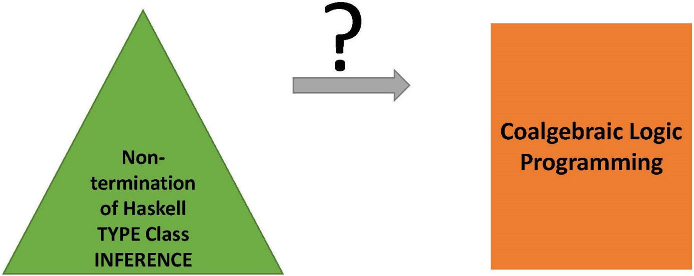


- We lost trust in our typeful verification method
- Do we need a better picture?

Personal Experience, in 2014



Personal Experience, in 2014



Reasons for doubts

- Moral (as discussed)
- Technical – lets see what they are

Outline

(Motivation) Verification methods: the good, the bad and the ugly

(Background) Proof-carrying code, revisited

(Technical Contribution) Going beyond state-of-the art: corecursion in type inference

(Conclusion) New type inference recipe: tastes good, does good

Relation of type classes to Horn Clause logic

```
class Eq x where
  eq :: Eq x => x -> x -> Bool

instance (Eq x, Eq y) => Eq (x, y) where
  eq (x1, y1) (x2, y2) = eq x1 x2 && eq y1 y2

instance Eq Int where
  eq x y = primtiveIntEq x y
```

Relation of type classes to Horn Clause logic

```
class Eq x where
  eq :: Eq x => x -> x -> Bool

instance (Eq x, Eq y) => Eq (x, y) where
  eq (x1, y1) (x2, y2) = eq x1 x2 && eq y1 y2

instance Eq Int where
  eq x y = primtiveIntEq x y
```

This translates into the following logic program:

$$Eq\ (x), Eq\ (y) \Rightarrow Eq(x, y)$$
$$\Rightarrow Eq\ (Int)$$

Relation of type classes to Horn Clause logic

```
class Eq x where
  eq :: Eq x => x -> x -> Bool

instance (Eq x, Eq y) => Eq (x, y) where
  eq (x1, y1) (x2, y2) = eq x1 x2 && eq y1 y2

instance Eq Int where
  eq x y = primtiveIntEq x y
```

This translates into the following logic program:

$Eq\ (x), Eq\ (y) \Rightarrow Eq(x, y)$
$\Rightarrow Eq\ (Int)$

Resolve the query $?\ Eq\ (Int, Int)$.

- We have the following reduction by SLD-resolution:

$$\Phi \vdash Eq\ (Int, Int) \rightarrow Eq\ (Int), Eq\ (Int) \rightarrow Eq\ (Int) \rightarrow \emptyset$$

Problems...

Ok, we have some grounds for interfacing Haskell type class resolution with logic programming. BUT:

Problems...

Ok, we have some grounds for interfacing Haskell type class resolution with logic programming. BUT:

- This syntactic correspondence is too shallow and fragile a ground for a long-term and sustainable methodology

Problems...

Ok, we have some grounds for interfacing Haskell type class resolution with logic programming. BUT:

- This syntactic correspondence is too shallow and fragile a ground for a long-term and sustainable methodology
- Gives a lot of trust to ATP, the latter is used as a black-box oracle, that certifies inference without constructing and passing back a proof evidence

Problems...

Ok, we have some grounds for interfacing Haskell type class resolution with logic programming. BUT:

- This syntactic correspondence is too shallow and fragile a ground for a long-term and sustainable methodology
- Gives a lot of trust to ATP, the latter is used as a black-box oracle, that certifies inference without constructing and passing back a proof evidence
- This approach lacks a conceptual understanding of relation between Types, Computation, and Proof

Problems...

Ok, we have some grounds for interfacing Haskell type class resolution with logic programming. BUT:

- This syntactic correspondence is too shallow and fragile a ground for a long-term and sustainable methodology
- Gives a lot of trust to ATP, the latter is used as a black-box oracle, that certifies inference without constructing and passing back a proof evidence
- This approach lacks a conceptual understanding of relation between Types, Computation, and Proof
- ... it is bound to cause practical and theoretical problems (with runnable proofs, corecursion, soundness, ...)

Problem - 1 (proofs are programs!)

```
class Eq x where
  eq :: Eq x => x -> x -> Bool

instance (Eq x, Eq y) => Eq (x, y) where
  eq (x1, y1) (x2, y2) = eq x1 x2 && eq y1 y2

instance Eq Int where
  eq x y = primtiveIntEq x y

test :: Eq (Int, Int) => Bool
test = eq (1,2) (1,2)

{- eval: test ==> True -}
```

We need to construct a proof evidence d for `Eq (Int, Int)` in `test`. In this example and generally, d needs to be run as a function by Haskell

Problem - 1 (proofs are programs!)

```
class Eq x where
  eq :: Eq x => x -> x -> Bool

instance (Eq x, Eq y) => Eq (x, y) where
  eq (x1, y1) (x2, y2) = eq x1 x2 && eq y1 y2

instance Eq Int where
  eq x y = primtiveIntEq x y

test :: Eq (Int, Int) => Bool
test = eq (1,2) (1,2)

{- eval: test ==> True -}
```

We need to construct a proof evidence d for `Eq (Int, Int)` in `test`. In this example and generally, d needs to be run as a function by Haskell

NB: Type Inhabitation problem!

In Haskell, proofs ARE type inhabitants

```
data Eq x where
  EqD :: (x -> x -> Bool) -> Eq x

eq :: Eq x -> (x -> x -> Bool)
eq (EqD e) = e

k1 :: Eq x -> Eq y -> Eq (x, y)
k1 d1 d2 = EqD q
  where  q (x1, y1) (x2, y2) = eq d1 x1 x2 && eq d2 y1 y2

k2 :: Eq Int
k2 = EqD primtiveIntEq

test :: Eq (Int, Int) -> Bool
test d = eq d (1,2) (1,2)

{- eval: test (k1 k2 k2) ==> True -}
```

How do we obtain `(k1 k2 k2)` for `test`? SLD-resolution alone is not sufficent

Solution - 1: make resolution proof relevant: Horn formulas as types, proofs as terms

Definition (Basic syntax)

Term	t	$::=$	$x \mid K \mid t\ t'$
Atomic Formula	A, B, C, D	$::=$	$P\ t_1\ ...\ t_n$
Horn Formula	H	$::=$	$B_1, ..., B_n \Rightarrow A$
Proof/Evidence	e	$::=$	$\kappa \mid e\ e'$
Axiom Environment	Φ	$::=$	$\cdot \mid \Phi, (\kappa : H)$

Definition (Resolution)

$\boxed{\Phi \vdash e : A}$

$$\frac{\Phi \vdash e_1 : \sigma B_1 \quad \cdots \quad \Phi \vdash e_n : \sigma B_n}{\Phi \vdash \kappa\ e_1 \cdots e_n : \sigma A} \text{ if } (\kappa : B_1, ..., B_n \Rightarrow A) \in \Phi$$

Solution - 1: make resolution proof relevant: Horn formulas as types, proofs as terms

Consider the following logic program Φ (clause names are constant proof terms)

$\kappa_1 : (Eq\ x, Eq\ y) \Rightarrow Eq(x, y)$
$\kappa_2 : \ \Rightarrow Eq\ Int$

Resolve the query $?\ Eq\ (Int, Int)$.

- We have the following resolution reduction:
  $\Phi \vdash Eq\ (Int, Int) \rightarrow_{\kappa_1} Eq\ Int, Eq\ Int \rightarrow_{\kappa_2} Eq\ Int \rightarrow_{\kappa_2} \emptyset$

Solution - 1: make resolution proof relevant: Horn formulas as types, proofs as terms

Consider the following logic program Φ (clause names are constant proof terms)

$$\kappa_1 : (Eq\ x, Eq\ y) \Rightarrow Eq(x, y)$$
$$\kappa_2 : \quad \Rightarrow Eq\ Int$$

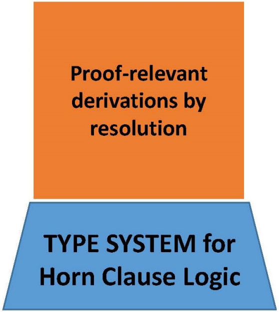
Resolve the query $?\ Eq\ (Int, Int)$.

- We have the following resolution reduction:
  $\Phi \vdash Eq\ (Int, Int) \rightarrow_{\kappa_1} Eq\ Int, Eq\ Int \rightarrow_{\kappa_2} Eq\ Int \rightarrow_{\kappa_2} \emptyset$
- Corresponding derivation:

$$\dfrac{\dfrac{\Phi \vdash \kappa_1 : Eq\ x, Eq\ y \Rightarrow Eq\ (x, y) \quad \Phi \vdash \kappa_2 : Eq\ Int}{\Phi \vdash \kappa_1\ \kappa_2 : Eq\ y \Rightarrow Eq\ (Int, y)} \qquad \Phi \vdash \kappa_2 : Eq\ Int}{\Phi \vdash \kappa_1\ \kappa_2\ \kappa_2 : Eq(Int, Int)}$$

So far...

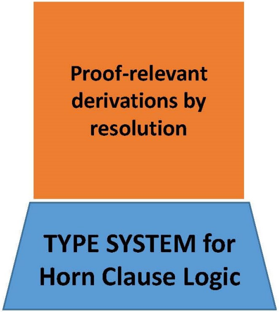
- We have started to build a “house" on solid grounds:



NB: automated proof construction = type inhabitation.

So far...

- We have started to build a “house" on solid grounds:



NB: automated proof construction = type inhabitation.
Is it a suitable home for real-world Haskell type inference?

Problem - 2: non-terminating cases of inference

especially common in “generic programming", cf. also “Scrape your boilerplate with class" papers by Lammel&Jones.

Simple Example of mutually recursive declarations:

```
data EvenList a  =  Nil | ECons a (OddList a)
data OddList a   =  OCons a (EvenList a)

instance (Eq a, Eq (OddList a)) => Eq (EvenList a) where
    eq Nil Nil =  True
    eq (ECons x xs) (ECons y ys)   =  eq x y && eq xs ys
    eq  _ _ =  False

instance (Eq a, Eq (EvenList a)) =>  Eq (OddList a) where
    eq (OCons x xs) (OCons y ys) =  eq x y && eq xs ys
    eq  _ _ =  False

test :: Eq (EvenList Int) => Bool
test = eq Nil Nil

{- eval: test ==> True -}
```

How to obtain evidence for `Eq (EvenList Int)`?

Cycling nontermination

Consider the corresponding logic program Φ

$$\kappa_1 : Eq\ x, Eq\ (EvenList\ x) \Rightarrow Eq\ (OddList\ x)$$

$$\kappa_2 : Eq\ x, Eq\ (OddList\ x) \Rightarrow Eq\ (EvenList\ x)$$

$$\kappa_3 : \quad \Rightarrow Eq\ Int$$

- For Query $Eq\ (EvenList\ Int)$:

$$\Phi \vdash \underline{Eq\ (EvenList\ Int)} \rightarrow_{\kappa_2} Eq\ Int, Eq\ (OddList\ Int) \rightarrow_{\kappa_3} Eq\ (OddList\ Int) \rightarrow_{\kappa_1} Eq\ Int, Eq\ (EvenList\ Int) \rightarrow_{\kappa_3} \underline{Eq\ (EvenList\ Int)}...$$

Cycling nontermination

Consider the corresponding logic program Φ

$$\kappa_1 : Eq\ x, Eq\ (EvenList\ x) \Rightarrow Eq\ (OddList\ x)$$

$$\kappa_2 : Eq\ x, Eq\ (OddList\ x) \Rightarrow Eq\ (EvenList\ x)$$

$$\kappa_3 : \ \Rightarrow Eq\ Int$$

- For Query $Eq\ (EvenList\ Int)$:

  $\Phi \vdash \underline{Eq\ (EvenList\ Int)} \rightarrow_{\kappa_2} Eq\ Int, Eq\ (OddList\ Int) \rightarrow_{\kappa_3}$
  $Eq\ (OddList\ Int) \rightarrow_{\kappa_1} Eq\ Int, Eq\ (EvenList\ Int) \rightarrow_{\kappa_3}$
  $\underline{Eq\ (EvenList\ Int)}...$
- So what is the d such that $\Phi \vdash d : Eq\ (EvenList\ Int)$?

Cycling nontermination

Consider the corresponding logic program Φ

$$\kappa_1 : Eq\ x, Eq\ (EvenList\ x) \Rightarrow Eq\ (OddList\ x)$$

$$\kappa_2 : Eq\ x, Eq\ (OddList\ x) \Rightarrow Eq\ (EvenList\ x)$$

$$\kappa_3 : \quad \Rightarrow Eq\ Int$$

- For Query $Eq\ (EvenList\ Int)$:

  $\Phi \vdash \underline{Eq\ (EvenList\ Int)} \to_{\kappa_2} Eq\ Int, Eq\ (OddList\ Int) \to_{\kappa_3}$
  $Eq\ (OddList\ Int) \to_{\kappa_1} Eq\ Int, Eq\ (EvenList\ Int) \to_{\kappa_3}$
  $\underline{Eq\ (EvenList\ Int)}\ldots$
- So what is the d such that $\Phi \vdash d : Eq\ (EvenList\ Int)$?
  Think of first occurrence as a coinductive hypothesis, and the second – as a coinductive conclusion

Solution-2: Typing Rule for Fixpoint

$$\frac{\Phi, \alpha : T \vdash e : T}{\Phi \vdash \nu\alpha.e : T}$$

- We can view $\nu\alpha.e$ as $\alpha = e$, where $\alpha \in \mathrm{FV}(e)$
- Operational meaning: $\nu\alpha.e \leadsto [\nu\alpha.e/\alpha]e$

Solution-2: Typing Rule for Fixpoint

$$\frac{\Phi, \alpha : T \vdash e : T}{\Phi \vdash \nu\alpha.e : T}$$

- We can view $\nu\alpha.e$ as $\alpha = e$, where $\alpha \in \mathrm{FV}(e)$
- Operational meaning: $\nu\alpha.e \rightsquigarrow [\nu\alpha.e/\alpha]e$
- We can view the type inhabited by such infinite proof as a coinductive type

Solution-2: Typing Rule for Fixpoint

$$\frac{\Phi, \alpha : T \vdash e : T}{\Phi \vdash \nu\alpha.e : T}$$

- We can view $\nu\alpha.e$ as $\alpha = e$, where $\alpha \in \mathrm{FV}(e)$
- Operational meaning: $\nu\alpha.e \rightsquigarrow [\nu\alpha.e/\alpha]e$
- We can view the type inhabited by such infinite proof as a coinductive type
- The typing derivation for $\Phi \vdash d : Eq(EvenList\ Int)$:

$$\frac{\dfrac{\cdots}{\Phi, \alpha : Eq(EvenList\ Int) \vdash \kappa_2\ \kappa_3\ (\kappa_1 \kappa_3\ \alpha) : Eq(EvenList\ Int)}}{\Phi \vdash \nu\alpha.\kappa_2\ \kappa_3\ (\kappa_1 \kappa_3\ \alpha) : Eq(EvenList\ Int)}$$

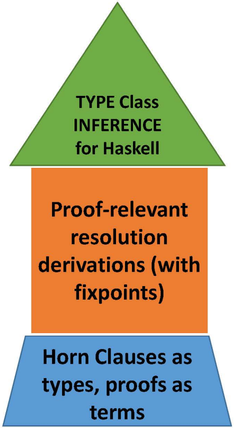
where Φ is the same:

$\kappa_1 : Eq\ x, Eq\ (EvenList\ x) \Rightarrow Eq\ (OddList\ x)$
$\kappa_2 : Eq\ x, Eq\ (OddList\ x) \Rightarrow Eq\ (EvenList\ x)$

$\kappa_3 : \ \Rightarrow Eq\ Int$

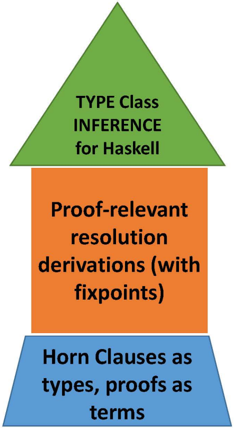
Our method unifies

foundations (type theory), implementation (evidence construction), applications (type class inference)



Our method unifies

foundations (type theory), implementation (evidence construction), applications (type class inference)



The ultimate Problem-3: can this go beyond state-of-the-art?

Outline

(Motivation) Verification methods: the good, the bad and the ugly

(Background) Proof-carrying code, revisited

(Technical Contribution) Going beyond state-of-the art: corecursion in type inference

(Conclusion) New type inference recipe: tastes good, does good

Haskell can handle only cycles, but not loops

[Generally, nontermination may exhibit cycles (formula repeats), loops (formula repeats modulo substitution), or neither.]

```
data Mu h a = In (h (Mu h) a)

data HPTree f a = HPLeaf a | HPNode (f (a, a))

instance Eq (h (Mu h) a) => Eq (Mu h a) where
  eq (In x) (In y) = eq x y

instance (Eq a, Eq (f (a, a))) => Eq (HPTree f a) where
  eq (HPLeaf x) (HPLeaf y) = eq x y
  eq (HPNode xs) (HPNode ys) = eq xs ys
  eq _ _ = False

tree :: Mu HPTree Int
tree = In (HPLeaf 34)

test :: Eq (Mu HPTree Int) => Bool
test = eq tree tree
```

Looping

The corresponding logic program Φ:

$$\begin{aligned}
\kappa_1 &: Eq(h\ (Mu\ h)\ a) \Rightarrow Eq(Mu\ h\ a)\\
\kappa_2 &: (Eq\ a, Eq(f\ (a,a))) \Rightarrow Eq(HPTree\ f\ a)\\
\kappa_3 &: (Eq\ x, Eq\ y) \Rightarrow Eq(x,y)\\
\kappa_4 &: \ \Rightarrow Eq\ Int
\end{aligned}$$

- For query $Eq\ (Mu\ HPTree\ Int)$:

$$\Phi \vdash \underline{Eq(Mu\ HPTree\ Int)} \rightarrow_{\kappa_1} Eq(HPTree\ (Mu\ HPTree)\ Int) \rightarrow_{\kappa_2}$$
$$Eq\ Int, Eq\ (Mu\ HPTree)\ (Int, Int) \rightarrow_{\kappa_4} \underline{Eq\ Mu\ HPTree\ (Int, Int)} \rightarrow_{\kappa_1}$$
$$Eq(HPTree\ (Mu\ HPTree)\ (Int, Int)) \rightarrow_{\kappa_2}$$
$$Eq\ (Int, Int), Eq\ (Mu\ HPTree)\ ((Int, Int), (Int, Int)) \rightarrow_{\kappa_3,\kappa_4,\kappa_4}$$
$$\underline{Eq\ Mu\ HPTree\ ((Int, Int), (Int, Int))}\ldots$$

- Current Haskell: no cycle detected – no answer!

Looping

The corresponding logic program Φ:

$$\kappa_1 : Eq(h\ (Mu\ h)\ a) \Rightarrow Eq(Mu\ h\ a)$$
$$\kappa_2 : (Eq\ a, Eq(f\ (a,a))) \Rightarrow Eq(HPTree\ f\ a)$$
$$\kappa_3 : (Eq\ x, Eq\ y) \Rightarrow Eq(x,y)$$
$$\kappa_4 : \quad \Rightarrow Eq\ Int$$

- For query $Eq\ (Mu\ HPTree\ Int)$:

$$\Phi \vdash \underline{Eq(Mu\ HPTree\ Int)} \rightarrow_{\kappa_1} Eq(HPTree\ (Mu\ HPTree)\ Int) \rightarrow_{\kappa_2}$$
$$Eq\ Int, Eq\ (Mu\ HPTree)\ (Int, Int) \rightarrow_{\kappa_4} \underline{Eq\ Mu\ HPTree\ (Int, Int)} \rightarrow_{\kappa_1}$$
$$Eq(HPTree\ (Mu\ HPTree)\ (Int, Int)) \rightarrow_{\kappa_2}$$
$$Eq\ (Int, Int), Eq\ (Mu\ HPTree)\ ((Int, Int), (Int, Int)) \rightarrow_{\kappa_3, \kappa_4, \kappa_4}$$
$$\underline{Eq\ Mu\ HPTree\ ((Int, Int), (Int, Int))} \ldots$$

- Current Haskell: no cycle detected – no answer!
- In our terms, the question is more subtle: what is the d such that $\Phi \vdash d : Eq\ (Mu\ HPTree\ Int)$?
  It is no longer a question of cycle detection, but a question of proof construction

A Theorem-proving perspective

The logic program Φ:

$$\begin{array}{c}
\kappa_1 : Eq(h\ (Mu\ h)\ a) \Rightarrow Eq(Mu\ h\ a) \\
\kappa_2 : (Eq\ a, Eq(f\ (a,a))) \Rightarrow Eq(HPTree\ f\ a) \\
\kappa_3 : (Eq\ x, Eq\ y) \Rightarrow Eq(x,y) \\
\kappa_4 : \ \Rightarrow Eq\ Int
\end{array}$$

- Directly proving $Eq\ (Mu\ HPTree\ Int)$ seems impossible

A Theorem-proving perspective

The logic program Φ:

$$
\begin{gathered}
\kappa_1 : Eq(h\ (Mu\ h)\ a) \Rightarrow Eq(Mu\ h\ a) \\
\kappa_2 : (Eq\ a, Eq(f\ (a, a))) \Rightarrow Eq(HPTree\ f\ a) \\
\kappa_3 : (Eq\ x, Eq\ y) \Rightarrow Eq(x, y) \\
\kappa_4 : \quad \Rightarrow Eq\ Int
\end{gathered}
$$

- Directly proving $Eq\ (Mu\ HPTree\ Int)$ seems impossible
- Prove a lemma $e : Eq\ x \Rightarrow Eq\ (Mu\ HPTree\ x)$ instead

A Theorem-proving perspective

The logic program Φ:

$$
\begin{gathered}
\kappa_1 : Eq(h\ (Mu\ h)\ a) \Rightarrow Eq(Mu\ h\ a) \\
\kappa_2 : (Eq\ a, Eq(f\ (a,a))) \Rightarrow Eq(HPTree\ f\ a) \\
\kappa_3 : (Eq\ x, Eq\ y) \Rightarrow Eq(x,y) \\
\kappa_4 : \ \Rightarrow Eq\ Int
\end{gathered}
$$

- Directly proving $Eq\ (Mu\ HPTree\ Int)$ seems impossible
- Prove a lemma $e : Eq\ x \Rightarrow Eq\ (Mu\ HPTree\ x)$ instead
- $(e\ \kappa_4) : Eq\ (Mu\ HPTree\ Int)$

A Theorem-proving perspective

The logic program Φ:

$$\begin{array}{c} \kappa_1 : Eq(h\ (Mu\ h)\ a) \Rightarrow Eq(Mu\ h\ a) \\ \kappa_2 : (Eq\ a, Eq(f\ (a,a))) \Rightarrow Eq(HPTree\ f\ a) \\ \kappa_3 : (Eq\ x, Eq\ y) \Rightarrow Eq(x,y) \\ \kappa_4 : \ \ \Rightarrow Eq\ Int \end{array}$$

- Directly proving $Eq\ (Mu\ HPTree\ Int)$ seems impossible
- Prove a lemma $e : Eq\ x \Rightarrow Eq\ (Mu\ HPTree\ x)$ instead
- $(e\ \kappa_4) : Eq\ (Mu\ HPTree\ Int)$
- Seeing our previous discussion of formulas with infinite proof evidence being coinductive types, the proof will need to be constructed by coinduction

A Theorem-proving perspective

$$\begin{gathered}\kappa_1 : Eq(h\ (Mu\ h)\ a) \Rightarrow Eq(Mu\ h\ a)\\ \kappa_2 : (Eq\ a, Eq(f\ (a,a))) \Rightarrow Eq(HPTree\ f\ a)\\ \kappa_3 : (Eq\ x, Eq\ y) \Rightarrow Eq(x,y)\\ \kappa_4 : Eq\ Int\end{gathered}$$

Derive $e : Eq\ x \Rightarrow Eq\ (Mu\ HPTree\ x)$ using fixpoint typing rule

1. Coinductive Assumption $\alpha : Eq\ x \Rightarrow Eq\ (Mu\ HPTree\ x)$

A Theorem-proving perspective

$$\kappa_1 : Eq(h\ (Mu\ h)\ a) \Rightarrow Eq(Mu\ h\ a)$$
$$\kappa_2 : (Eq\ a, Eq(f\ (a,a))) \Rightarrow Eq(HPTree\ f\ a)$$
$$\kappa_3 : (Eq\ x, Eq\ y) \Rightarrow Eq(x,y)$$
$$\kappa_4 : Eq\ Int$$

Derive $e : Eq\ x \Rightarrow Eq\ (Mu\ HPTree\ x)$ using fixpoint typing rule

1. Coinductive Assumption $\alpha : Eq\ x \Rightarrow Eq\ (Mu\ HPTree\ x)$
2. Assume $\alpha_1 : Eq\ x$, to show $Eq\ (Mu\ HPTree\ x)$

A Theorem-proving perspective

$$\kappa_1 : Eq(h\ (Mu\ h)\ a) \Rightarrow Eq(Mu\ h\ a)$$
$$\kappa_2 : (Eq\ a, Eq(f\ (a,a))) \Rightarrow Eq(HPTree\ f\ a)$$
$$\kappa_3 : (Eq\ x, Eq\ y) \Rightarrow Eq(x,y)$$
$$\kappa_4 : Eq\ Int$$

Derive $e : Eq\ x \Rightarrow Eq\ (Mu\ HPTree\ x)$ using fixpoint typing rule

1. Coinductive Assumption $\alpha : Eq\ x \Rightarrow Eq\ (Mu\ HPTree\ x)$
2. Assume $\alpha_1 : Eq\ x$, to show $Eq\ (Mu\ HPTree\ x)$
3. Apply κ_1, we get a new goal $Eq(HPTree\ (Mu\ HPTree)\ x)$

A Theorem-proving perspective

$$\kappa_1 : Eq(h\ (Mu\ h)\ a) \Rightarrow Eq(Mu\ h\ a)$$
$$\kappa_2 : (Eq\ a, Eq(f\ (a,a))) \Rightarrow Eq(HPTree\ f\ a)$$
$$\kappa_3 : (Eq\ x, Eq\ y) \Rightarrow Eq(x,y)$$
$$\kappa_4 : Eq\ Int$$

Derive $e : Eq\ x \Rightarrow Eq\ (Mu\ HPTree\ x)$ using fixpoint typing rule

1. Coinductive Assumption $\alpha : Eq\ x \Rightarrow Eq\ (Mu\ HPTree\ x)$
2. Assume $\alpha_1 : Eq\ x$, to show $Eq\ (Mu\ HPTree\ x)$
3. Apply κ_1, we get a new goal $Eq(HPTree\ (Mu\ HPTree)\ x)$
4. Apply κ_2, we get $Eq\ x, Eq\ (Mu\ HPTree)\ (x,x)$

A Theorem-proving perspective

$$\kappa_1 : Eq(h\ (Mu\ h)\ a) \Rightarrow Eq(Mu\ h\ a)$$
$$\kappa_2 : (Eq\ a, Eq(f\ (a,a))) \Rightarrow Eq(HPTree\ f\ a)$$
$$\kappa_3 : (Eq\ x, Eq\ y) \Rightarrow Eq(x,y)$$
$$\kappa_4 : Eq\ Int$$

Derive $e : Eq\ x \Rightarrow Eq\ (Mu\ HPTree\ x)$ using fixpoint typing rule

1. Coinductive Assumption $\alpha : Eq\ x \Rightarrow Eq\ (Mu\ HPTree\ x)$
2. Assume $\alpha_1 : Eq\ x$, to show $Eq\ (Mu\ HPTree\ x)$
3. Apply κ_1, we get a new goal $Eq(HPTree\ (Mu\ HPTree)\ x)$
4. Apply κ_2, we get $Eq\ x, Eq\ (Mu\ HPTree)\ (x,x)$
5. $Eq\ x$ is proven by α_1

A Theorem-proving perspective

$$\begin{gathered}\kappa_1 : Eq(h\ (Mu\ h)\ a) \Rightarrow Eq(Mu\ h\ a)\\ \kappa_2 : (Eq\ a, Eq(f\ (a,a))) \Rightarrow Eq(HPTree\ f\ a)\\ \kappa_3 : (Eq\ x, Eq\ y) \Rightarrow Eq(x,y)\\ \kappa_4 : Eq\ Int\end{gathered}$$

Derive $e : Eq\ x \Rightarrow Eq\ (Mu\ HPTree\ x)$ using fixpoint typing rule

1. Coinductive Assumption $\alpha : Eq\ x \Rightarrow Eq\ (Mu\ HPTree\ x)$
2. Assume $\alpha_1 : Eq\ x$, to show $Eq\ (Mu\ HPTree\ x)$
3. Apply κ_1, we get a new goal $Eq(HPTree\ (Mu\ HPTree)\ x)$
4. Apply κ_2, we get $Eq\ x, Eq\ (Mu\ HPTree)\ (x,x)$
5. $Eq\ x$ is proven by α_1
6. Apply α on $Eq\ (Mu\ HPTree)\ (x,x)$, get $Eq\ (x,x)$

A Theorem-proving perspective

$$\kappa_1 : Eq(h\ (Mu\ h)\ a) \Rightarrow Eq(Mu\ h\ a)$$
$$\kappa_2 : (Eq\ a, Eq(f\ (a,a))) \Rightarrow Eq(HPTree\ f\ a)$$
$$\kappa_3 : (Eq\ x, Eq\ y) \Rightarrow Eq(x,y)$$
$$\kappa_4 : Eq\ Int$$

Derive $e : Eq\ x \Rightarrow Eq\ (Mu\ HPTree\ x)$ using fixpoint typing rule

1. Coinductive Assumption $\alpha : Eq\ x \Rightarrow Eq\ (Mu\ HPTree\ x)$
2. Assume $\alpha_1 : Eq\ x$, to show $Eq\ (Mu\ HPTree\ x)$
3. Apply κ_1, we get a new goal $Eq(HPTree\ (Mu\ HPTree)\ x)$
4. Apply κ_2, we get $Eq\ x, Eq\ (Mu\ HPTree)\ (x,x)$
5. $Eq\ x$ is proven by α_1
6. Apply α on $Eq\ (Mu\ HPTree)\ (x,x)$, get $Eq\ (x,x)$
7. Apply κ_3, α_1 on $Eq\ (x,x)$, Q.E.D.
   $\nu\alpha.\lambda\alpha_1.\kappa_1\ (\kappa_2\ \alpha_1\ (\alpha\ (\kappa_3\ \alpha_1\ \alpha_1))) : Eq\ x \Rightarrow Eq\ (Mu\ HPTree\ x)$

Proof-relevant Corecursive Resolution at a glance

$$\frac{\Phi \vdash e_1 : \sigma B_1 \quad \cdots \quad \Phi \vdash e_n : \sigma B_n}{\Phi \vdash e\ e_1 \cdots e_n : \sigma A} \text{ if } (e : B_1, ..., B_m \Rightarrow A) \in \Phi$$

$$\frac{\Phi, (\alpha : \underline{A} \Rightarrow B) \vdash e : \underline{A} \Rightarrow B \quad HNF(e)}{\Phi \vdash \nu\alpha.e : \underline{A} \Rightarrow B} \ (\textsc{Nu})$$

$$\frac{\Phi, (\underline{\alpha} : \underline{A}) \vdash e : B}{\Phi \vdash \lambda\underline{\alpha}.e : \underline{A} \Rightarrow B} \ (\textsc{Lam})$$

Proof-relevant Corecursive Resolution at a glance

$$\frac{\Phi \vdash e_1 : \sigma B_1 \quad \cdots \quad \Phi \vdash e_n : \sigma B_n}{\Phi \vdash e\ e_1 \cdots e_n : \sigma A} \text{ if } (e : B_1, ..., B_m \Rightarrow A) \in \Phi$$

$$\frac{\Phi, (\alpha : \underline{A} \Rightarrow B) \vdash e : \underline{A} \Rightarrow B \quad HNF(e)}{\Phi \vdash \nu\alpha.e : \underline{A} \Rightarrow B} \text{ (Nu)}$$

$$\frac{\Phi, (\underline{\alpha} : \underline{A}) \vdash e : B}{\Phi \vdash \lambda\underline{\alpha}.e : \underline{A} \Rightarrow B} \text{ (Lam)}$$

Alternatively, take Howard's system **H** (STLC), prove admissibility of the resolution rule, and extend **H** with rule NU.

Technical Summary

- We extended resolution with coinductive proofs (coinductive hypothesis + corecursive evidence construction), and with implicative goals

Technical Summary

- We extended resolution with coinductive proofs (coinductive hypothesis + corecursive evidence construction), and with implicative goals
- The method is robust and extendable: in the limit, we can go as far as interactive theorem provers go

Technical Summary

- We extended resolution with coinductive proofs (coinductive hypothesis + corecursive evidence construction), and with implicative goals
- The method is robust and extendable: in the limit, we can go as far as interactive theorem provers go
- However, the most intelligent part now becomes to generate the coinductive hypotheses

Technical Summary

- We extended resolution with coinductive proofs (coinductive hypothesis + corecursive evidence construction), and with implicative goals
- The method is robust and extendable: in the limit, we can go as far as interactive theorem provers go
- However, the most intelligent part now becomes to generate the coinductive hypotheses
- See our FLOPS'16 paper for a heuristic automating loop detection and coinductive lemma generation
- So far, it is limited to looping nontermination

Outline

(Motivation) Verification methods: the good, the bad and the ugly

(Background) Proof-carrying code, revisited

(Technical Contribution) Going beyond state-of-the art: corecursion in type inference

(Conclusion) New type inference recipe: tastes good, does good

High-level Summary

Proof-relevant (or Curry-Howard) Resolution:

1. Retains operational semantics of first-order resolution (Operationally, it is still the ATP we started with!)

High-level Summary

Proof-relevant (or Curry-Howard) Resolution:

1. Retains operational semantics of first-order resolution (Operationally, it is still the ATP we started with!)
2. Proof-relevant (in Curry-Howard sense: Horn Formulas as Types, proofs as terms)

High-level Summary

Proof-relevant (or Curry-Howard) Resolution:

1. Retains operational semantics of first-order resolution (Operationally, it is still the ATP we started with!)
2. Proof-relevant (in Curry-Howard sense: Horn Formulas as Types, proofs as terms)
3. (Co)Recursive (with fixpoint terms inhabiting [coinductive] formulas with infinite proofs)

High-level Summary

Proof-relevant (or Curry-Howard) Resolution:

1. Retains operational semantics of first-order resolution (Operationally, it is still the ATP we started with!)
2. Proof-relevant (in Curry-Howard sense: Horn Formulas as Types, proofs as terms)
3. (Co)Recursive (with fixpoint terms inhabiting [coinductive] formulas with infinite proofs)
4. Coherently unifies type theory, automated proving, type inference

High-level Summary

Proof-relevant (or Curry-Howard) Resolution:

1. Retains operational semantics of first-order resolution (Operationally, it is still the ATP we started with!)
2. Proof-relevant (in Curry-Howard sense: Horn Formulas as Types, proofs as terms)
3. (Co)Recursive (with fixpoint terms inhabiting [coinductive] formulas with infinite proofs)
4. Coherently unifies type theory, automated proving, type inference
5. Based on solid principles: Curry-Howard approach to logic, computation, and proof

High-level Summary

Proof-relevant (or Curry-Howard) Resolution:

1. Retains operational semantics of first-order resolution (Operationally, it is still the ATP we started with!)
2. Proof-relevant (in Curry-Howard sense: Horn Formulas as Types, proofs as terms)
3. (Co)Recursive (with fixpoint terms inhabiting [coinductive] formulas with infinite proofs)
4. Coherently unifies type theory, automated proving, type inference
5. Based on solid principles: Curry-Howard approach to logic, computation, and proof
6. ... elegantly bridging the gap between ATP and ITP

Automated inference, program, proof

Bringing three classic themes back together:

Automated inference (type level)	**Corresponding function (term-level)**	**Proof Principle**
terminating resolution		

Automated inference, program, proof

Bringing three classic themes back together:

Automated inference (type level)	**Corresponding function (term-level)**	**Proof Principle**
terminating resolution	proof evidence construction	

Automated inference, program, proof

Bringing three classic themes back together:

Automated inference (type level)	**Corresponding function (term-level)**	**Proof Principle**
terminating resolution	proof evidence construction	inductive proofs

Automated inference, program, proof

Bringing three classic themes back together:

Automated inference (type level)	**Corresponding function (term-level)**	**Proof Principle**
terminating resolution	proof evidence construction	inductive proofs
non-terminating resolution		

Automated inference, program, proof

Bringing three classic themes back together:

Automated inference (type level)	**Corresponding function (term-level)**	**Proof Principle**
terminating resolution	proof evidence construction	inductive proofs
non-terminating resolution	corecursive evidence construction	

Automated inference, program, proof

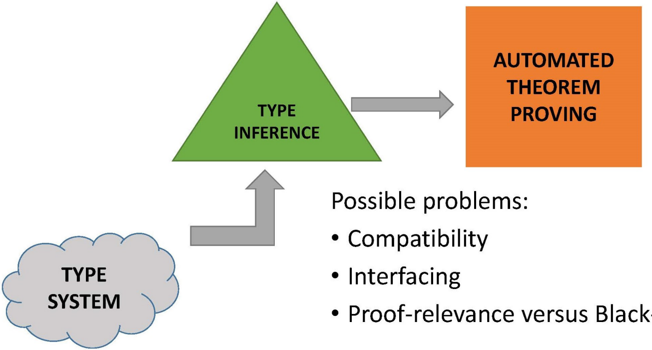
Bringing three classic themes back together:

Automated inference (type level)	**Corresponding function (term-level)**	**Proof Principle**
terminating resolution	proof evidence construction	inductive proofs
non-terminating resolution	corecursive evidence construction	coinductive proofs

Dream for the future

change of methodology for all ATP in Type inference

From:

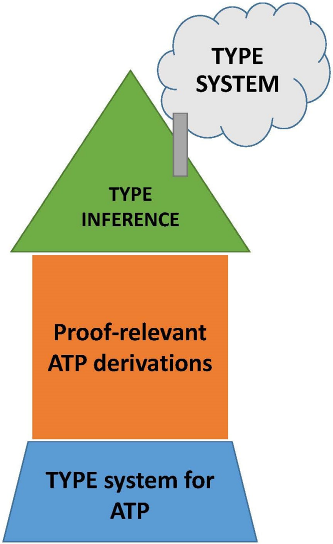


Possible problems:

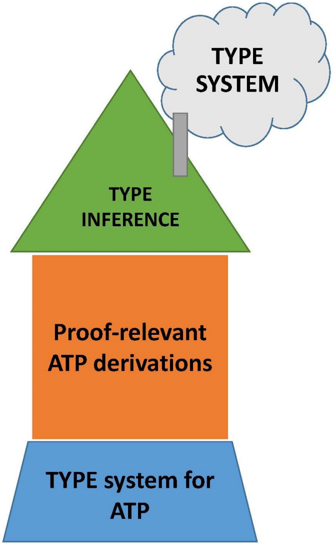
- Compatibility
- Interfacing
- Proof-relevance versus Black-box

To...

Dream for the future



Dream for the future



How far can it take us? – New standards of hygene in type-level computation?

- We have built a similar methodology for TRS (first-order terms as types, reductions as proofs).
- Extend to other type inference problems?
- Extend to ATP richer than Horn clauses (e.g. extended Horn clauses used in SMT-solvers?

Thanks for your attention!