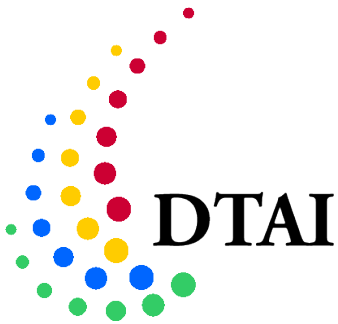
Type Invariants for Haskell

Tom Schrijvers Louis-Julien Guillemette Stefan Monnier

Katholieke Universiteit Leuven, Belgium
TOM.SCHRIJVERS@CS.KULEUVEN.BE

Université de Montréal
{GUILLELJ,MONNIER}@IRO.UMONTREAL.CA

PLPV 2009 – January 20, 2009



Overview

Overview

Haskell's Type Language

Haskell (GHC 6.10) has rich type language:

- **type constants**: algebraic data types

```
data Even
data Odd
```

Haskell's Type Language

Haskell (GHC 6.10) has rich type language:

- **type constants**: algebraic data types

```
data Even
data Odd
```

- **predicates** over types: type classes

```
class Parity p
instance Parity Even
instance Parity Odd
```

Haskell's Type Language

Haskell (GHC 6.10) has rich type language:

- **type constants**: algebraic data types

```
data Even
data Odd
```

- **predicates** over types: type classes

```
class Parity p
instance Parity Even
instance Parity Odd
```

- **functions** over types: type functions

```
type family Flip p
type instance Flip Even = Odd
type instance Flip Odd  = Even
```

Haskell's Type Language Ctd.

- GADTs capture data invariants at type-level, checked at compile time

```
-- list indexed by element type and parity
data List e p
  where Nil  :: List e Even
        Cons :: e -> List e p -> List e (Flip p)
```

Haskell's Type Language Ctd.

- GADTs capture data invariants at type-level, checked at compile time

```
-- list indexed by element type and parity
data List e p
  where Nil  :: List e Even
        Cons :: e -> List e p -> List e (Flip p)

head :: List e p -> e
head (Cons x xs) = x
```

Haskell's Type Language Ctd.

- GADTs capture data invariants at type-level, checked at compile time

```
-- list indexed by element type and parity
data List e p
  where Nil  :: List e Even
        Cons :: e -> List e p -> List e (Flip p)

head :: List e p -> e
head (Cons x xs) = x

tail :: List e p -> List e (Flip p)
tail (Cons x xs) = xs
```

The Problem

The function `tail` does not type-check!

```
-- list indexed by element type and parity
data List e p
  where Nil  :: List e Even
        Cons :: e -> List e p -> List e (Flip p)

tail :: List e p -> List e (Flip p)
tail (Cons x xs) = xs
```

What's happening?

- Assume parity of `xs` is `q`

The Problem

The function `tail` does not type-check!

```
-- list indexed by element type and parity
data List e p
  where Nil  :: List e Even
        Cons :: e -> List e p -> List e (Flip p)

tail :: List e p -> List e (Flip p)
tail (Cons x xs) = xs
```

What's happening?

- Assume parity of `xs` is `q`
- From pattern match, we know `p ~ Flip q`

The Problem

The function `tail` does not type-check!

```
-- list indexed by element type and parity
data List e p
  where Nil  :: List e Even
        Cons :: e -> List e p -> List e (Flip p)

tail :: List e p -> List e (Flip p)
tail (Cons x xs) = xs
```

What's happening?

- Assume parity of `xs` is `q`
- From pattern match, we know `p ~ Flip q`
- From signature, we promise `q ~ Flip p`

The Problem

The function `tail` does not type-check!

```
-- list indexed by element type and parity
data List e p
  where Nil  :: List e Even
        Cons :: e -> List e p -> List e (Flip p)

tail :: List e p -> List e (Flip p)
tail (Cons x xs) = xs
```

What's happening?

- Assume parity of `xs` is `q`
- From pattern match, we know `p ~ Flip q`
- From signature, we promise `q ~ Flip p`
- Type checker cannot show `p ~ Flip (Flip p)`

The Problem

Why can't the type checker show this?

```
p ~ Flip (Flip p)
```

when

```
type family Flip p
type instance Flip Even = Odd
type instance Flip Odd  = Even
```

Open Definitions

Type functions (and type classes) have **open definitions** in Haskell:

- The programmer may extend the definition "at any time", i.e. in another module of the program.
- The type checker has to take the openness into consideration.

Open Definitions

Type functions (and type classes) have **open definitions** in Haskell:

- The programmer may extend the definition "at any time", i.e. in another module of the program.
- The type checker has to take the openness into consideration.

While `p ~ Flip (Flip p)` is valid for

```
type family Flip p
type instance Flip Even = Odd
type instance Flip Odd  = Even
```

Open Definitions

Type functions (and type classes) have **open definitions** in Haskell:

- The programmer may extend the definition "at any time", i.e. in another module of the program.
- The type checker has to take the openness into consideration.

While `p ~ Flip (Flip p)` is valid for

```
type family Flip p
type instance Flip Even = Odd
type instance Flip Odd  = Even
```

it is not, when extended with:

```
type instance Flip Int = Even
```

because `Flip (Flip Int) ~ Odd`

Another Example

```
-- list type indexed by element type and length
data List e l
  where Nil :: List e Zero
        Cons :: e -> List e l -> List e (Succ l)

type family Add x y
type instance Add Zero y = y
type instance Add (Succ x) y = Succ (Add x y)

merge :: List e k -> List e l -> List e (Add k l)
merge Nil         ys = ys
merge (Cons x xs) ys = Cons x (merge ys xs)
```

Another Example

```
-- list type indexed by element type and length
data List e l
  where Nil :: List e Zero
        Cons :: e -> List e l -> List e (Succ l)

type family Add x y
type instance Add Zero y = y
type instance Add (Succ x) y = Succ (Add x y)

merge :: List e k -> List e l -> List e (Add k l)
merge Nil         ys = ys
merge (Cons x xs) ys = Cons x (merge ys xs)
```

Type checker needs commutativity of `Add`:

```
Add x y ~ Add y x
```

Problem Summary

- Properties of type definitions (and type classes) are necessary for type checking in various situations (see paper).

Problem Summary

- Properties of type definitions (and type classes) are necessary for type checking in various situations (see paper).
- Due to their open nature, it is not valid to derive the properties from the definitions of type functions and type classes.

Solution: Type Invariants

Idea:

- Programmer explicitly formulates property as an invariant.

```
type invariant flipflip =
  Flip (Flip x) ~ x
```

Solution: Type Invariants

Idea:

- Programmer explicitly formulates property as an invariant.

```
type invariant flipflip =
  Flip (Flip x) ~ x
```

- All current *and future* instances must satisfy the invariant.

```
type instance Flip Even    = Odd
type instance Flip Odd     = Even
-- type instance Flip Int = Even
```

Invariant Domain?

Are these invariants valid?

```
type family Empty x
-- no instances

type invariant empty1 = Empty x ~ Bool
type invariant empty1 = Empty x ~ Char
```

Invariant Domain?

Are these invariants valid?

```
type family Empty x
-- no instances

type invariant empty1 = Empty x ~ Bool
type invariant empty1 = Empty x ~ Char
```

- Yes, because there are no instances.

Invariant Domain?

Are these invariants valid?

```
type family Empty x
-- no instances

type invariant empty1 = Empty x ~ Bool
type invariant empty1 = Empty x ~ Char
```

- Yes, because there are no instances.
- No, because we can show `Bool ~ Empty Int ~ Char`!

Domain Restriction

- Haskell does not restrict the domain.

Domain Restriction

- Haskell does not restrict the domain.
- We must do so explicitly ourselves:

```
type invariant flipflip =
 Parity p =>  Flip (Flip p) ~ p
```

- type class constraints restrict the domain of the invariant

```
class Parity p
instance Parity Even
instance Parity Odd
```

Class Invariants

- `class Num n` carved in stone (Haskell Report)

Class Invariants

- `class Num n` carved in stone (Haskell Report)
- `class Additive n` afterwards

Class Invariants

- `class Num n` carved in stone (Haskell Report)
- `class Additive n` afterwards
- would have liked to make it a superclass:

```
class Additive n => Num n
```

Class Invariants

- `class Num n` carved in stone (Haskell Report)
- `class Additive n` afterwards
- would have liked to make it a superclass:

```
class Additive n => Num n
```

- instead write invariant:

```
type invariant num_addititve = Num n => Additive n
```

Other Invariants

- Predicates over functions

```
Parity p => Parity (Flip p)
```

Other Invariants

- Predicates over functions

```
Parity p => Parity (Flip p)
```

- Functional Dependencies

```
C x y1, C x y2 => y1 ~ y2
```

Other Invariants

- Predicates over functions

```
Parity p => Parity (Flip p)
```

- Functional Dependencies

```
C x y1, C x y2 => y1 ~ y2
```

- Not limited to one type class or type function

```
Address a => AddressOf (ValueOf a) ~ a
```

- More: see paper.

Proving Invariants

Who proves whether the instances actually obey the invariant?

- in general, too difficult to do automatically

Proving Invariants

Who proves whether the instances actually obey the invariant?

- in general, too difficult to do automatically
- proof is **written manually** by programmer

Proving Invariants

Who proves whether the instances actually obey the invariant?

- in general, too difficult to do automatically
- proof is **written manually** by programmer
- proof is **checked automatically**

Case-based Proofs

What to prove?

- one proof case per type class instance
- add proof cases for new type class instances

```
type invariant flipflip =
 Parity p =>  Flip (Flip p) ~ p

class Parity
instance Parity Even
instance Parity Odd

proofcase flipflip Even = ...
proofcase flipflip Odd  = ...
```

Proof language

Two proof languages:

- **external** proof language
  - high-level
  - human written proofs
  - "big steps", omits details

Proof language

Two proof languages:

- **external** proof language
  - high-level
  - human written proofs
  - “big steps”, omits details
- **internal** proof language
  - low-level, verbose
  - machine checked proofs
  - small steps, all details

External Proof Language

```
proofcase flipflip Even =
    Flip (Flip Even)
  ~ Even
proofcase flipflip Odd =
    Flip (Flip Odd)
  ~ Odd
```

- equational reasoning style proofs
- start from LHS, e.g. `Flip (Flip Even)`
- end with RHS, e.g. `Even`
- using built-in equality

Internal Proof Language

```
proofcase flipflip Even =
  Flip flip_axiom1 `trans` flip_axiom2
proofcase flipflip Odd =
  Flip flip_axiom2 `trans` flip_axiom1
```

- proof combinators (coercions) from System F_C [Sulzmann et al.]
  - `trans :: (t1 ~ t2) -> (t2 ~ t3) -> (t1 ~ t3)`
  - `Flip :: (t1 ~ t2) -> (Flip t1 ~ Flip t2)`
- tree-shaped proofs
- all the details, e.g. which axiom used
  - `flip_axiom1 :: (Flip Even ~ Odd)`
  - `flip_axiom2 :: (Flip Odd ~ Even)`

Translation from External to Internal

Recipe:

- translate each external step $\tau_i \sim \tau_{i+1}$ into internal proof π_i
  - undecidable in general (see paper)
- transitively compose

$$\pi_1 \text{ `trans` } \pi_2 \text{ `trans` } \ldots \text{ `trans` } \pi_{n-1}$$

Inductive Proofs

```
proofcase add_comm Zero (Succ m) =
      Add Zero (Succ m) ~
      Succ (Add Zero m) ~{ind add_comm}
      Succ (Add m Zero) ~
      Add (Succ m) Zero
```

- proof of invariant may rely on (other) invariant

Inductive Proofs

```
proofcase add_comm Zero (Succ m) =
      Add Zero (Succ m) ~
      Succ (Add Zero m) ~{ind add_comm}
      Succ (Add m Zero) ~
      Add (Succ m) Zero
```

- proof of invariant may rely on (other) invariant
- explicitly annotate wich invariant is used

Inductive Proofs

```
proofcase add_comm Zero (Succ m) =
      Add Zero (Succ m) ~
      Succ (Add Zero m) ~{ind add_comm}
      Succ (Add m Zero) ~
      Add (Succ m) Zero
```

- proof of invariant may rely on (other) invariant
- explicitly annotate wich invariant is used
- explicitly annotate whether proof is (mutually) inductive

Inductive Proofs

```
proofcase add_comm Zero (Succ m) =
      Add Zero (Succ m) ~
      Succ (Add Zero m) ~{ind add_comm}
      Succ (Add m Zero) ~
      Add (Succ m) Zero
```

- proof of invariant may rely on (other) invariant
- explicitly annotate wich invariant is used
- explicitly annotate whether proof is (mutually) inductive
- inductive uses must be smaller (see paper)

Inductive Proofs

```
proofcase add_comm Zero (Succ m) =
      Add Zero (Succ m) ~
      Succ (Add Zero m) ~{ind add_comm}
      Succ (Add m Zero) ~
      Add (Succ m) Zero
```

- proof of invariant may rely on (other) invariant
- explicitly annotate wich invariant is used
- explicitly annotate whether proof is (mutually) inductive
- inductive uses must be smaller (see paper)
- annotations are checked automatically

Inductive Proofs

```
proofcase add_comm Zero (Succ m) =
      Add Zero (Succ m) ~
      Succ (Add Zero m) ~{ind add_comm}
      Succ (Add m Zero) ~
      Add (Succ m) Zero
```

- proof of invariant may rely on (other) invariant
- explicitly annotate wich invariant is used
- explicitly annotate whether proof is (mutually) inductive
- inductive uses must be smaller (see paper)
- annotations are checked automatically

Using Invariants

How/when should the type checker use the invariants?

- too general in nature

Using Invariants

How/when should the type checker use the invariants?

- too general in nature
- undecidable when/how to use

Using Invariants

How/when should the type checker use the invariants?

- too general in nature
- undecidable when/how to use
- leads to incompleteness / non-termination

Using Invariants

How/when should the type checker use the invariants?

- too general in nature
- undecidable when/how to use
- leads to incompleteness / non-termination

Manual use!

- function for invariant application

```
flipflip ::  Parity x => (Flip (Flip x) ~ x => a) -> a
```

Using Invariants

How/when should the type checker use the invariants?

- too general in nature
- undecidable when/how to use
- leads to incompleteness / non-termination

Manual use!

- function for invariant application

```
flipflip ::  Parity x => (Flip (Flip x) ~ x => a) -> a
```

- used by programmer

```
tail :: Parity p => List e p -> List e (Flip p)
tail (Cons x xs) = flipflip xs
```

Evaluation

main invariant	aux. invariants	cases	proof size	steps
parity	$0 + 0$	$2 + 0$	10	4
commutativity	$0 + 0$	$4 + 0$	60	19
CPS	$5 + 1$	$21 + 3$	327	101^*

naive proptotype with bounded search for translation

- equational + type class invariants
- number of proof cases
- size of internal proof
- steps of external proof

(*): 18 combinators not covered due to current translation limitations

Summary

- **type invariants**: properties of open type definitions
- type classes **restrict domain** of invariants
- case-based invariant **proofs**
- high-level external proof language
- low-level internal proof language

See paper for formal details!

Future Work

- extend external proof language

Future Work

- extend external proof language
- improve reconstruction of internal proof

Future Work

- extend external proof language
- improve reconstruction of internal proof
- interactive proving tool

Future Work

- extend external proof language
- improve reconstruction of internal proof
- interactive proving tool
- implementation scheme for using type class proofs (current dictionaries unsuitable)

Thank You!

Questions?