Heuristics for type error discovery and recovery

Jurriaan Hage
jur@cs.uu.nl
most work by Bastiaan Heeren (bastiaan@cs.uu.nl)

Center for Software Technology, Department of Computer and Information Sciences
Universiteit Utrecht

September 5, 2006

Overview

1. Motivation and context
2. The heuristics
3. Examples
4. Implementation
5. Summary and validation

Context

- Strongly-typed, higher-order, polymorphic functional languages.
- Typically, type inferencers perform unifications during an AST traversal.
  - Results in a substitution or an error message.
- Unification merges a piece of type information into the substitution.
- Consequence: unifications towards the end of the program get the blame.
  - artifact of inferencing process, not programming
- When an error is discovered, we do not know how we got there.
- Constraint based type systems are certainly an improvement
  - (often) small and mobile pieces of information (reordering)
  - special solvers can be built for them, and reused a lot
- Still, type inferencing is done one constraint at the time (bias)

Contributions

- We consider sets of constraints at the time, not a single constraint
  - approximately a whole binding group at the time
- We map these to a special datastructure, a type graph
  - Essentially, it can also represent inconsistent sets of unifications
- This structure is amenable to the definition of various heuristics
  - general heuristics
  - programming language dependent heuristics
  - programmer dependent heuristics
- We implemented quite a number of these (own, others, folklore)
  - It is easy to define and plug-in new heuristics
- Heuristics encapsulate *expert knowledge* and *remove bias*
- A voting mechanism decides between competing heuristics
- Debugging mechanism: see the Appendix of the paper

Additional benefits

- The usual for using constraints:
  - decoupling of type system specification and inference algorithm
  - many solvers can be defined, each with their own specialty
  - amenable to reuse: many languages use the same kind of constraints
- Global analysis gives more flexibility, and when done cleverly doesn't degrade efficiency much
- Outcome can help us determine which kind of mistakes are made most often
  - Simply consider which heuristic is used most (if used correctly)
  - Helium has a logging facility (60,000 programs thus far)

"Limitations"

- In this talk, I consider only sets of unification (equality) constraints.
- However, we do handle polymorphism (efficiently), but at a different level.
  - Between binding groups versus within binding groups
- I can tell you what we did, but not how.
  - More information in the paper, more in a technical report and even more in Bastiaan Heeren's PhD.

Haskell versus Helium

- A stronger limitation: Helium is a subset of Haskell
- Limited overloading:
  - overloading yes/no is an option
  - no user-definable classes and instances
  - only support for `Enum`,`Eq`,`Num`,`Ord`,`Show`, all instances derived
  - no overloading on numerals... ever.
- No records, $n + k$ patterns, `newtype`, qualified imports, literate programming,...
- Check the website for the current status

Haskell versus Helium

- A stronger limitation: Helium is a subset of Haskell
- Limited overloading:
  - overloading yes/no is an option
  - no user-definable classes and instances
  - only support for `Enum`,`Eq`,`Num`,`Ord`,`Show`, all instances derived
  - no overloading on numerals... ever.
- No records, $n + k$ patterns, `newtype`, qualified imports, literate programming,...
- Check the website for the current status

We have much of Haskell98, but not all (making our life a little easier).

The list of heuristics (a selection)

- high participation rate [28 LOC]
  - enforces that when one int goes against ten booleans, the int loses
- Repair heuristics (selectors) with voting mechanism
  - sibling functions [43] and literals [40]
  - application heuristic [177]
  - tuple heuristic [50]
    - application like heuristic for tuples
  - function binding has too many arguments [32]
    - $f\ x = 0$ although $f :: Int$
  - variable function [40]
    - variable is not an application unless it has some arguments
- Tie breakers (if nothing else helps), mainly to avoid constraints
  - that should not be blamed [10] (type of let and type of body)
  - that are trusted [7] (explicit types, Prelude functions)
  - that give bad error messages [7] (folklore constraints)
- The final tie breaker: first come first blamed [5].

A first example

```
doubleList :: [Int] -> [Int]
doubleList xs = map (*2)
```

Second error message

```
(3,17): Type error in application
 expression        : map (* 2)
 term              : map
   type            : (a   -> b  ) -> [a] -> [b]
   does not match  : (Int -> Int) -> [Int]
 probable fix      : insert a second argument
```

Multi-example

```
elem :: a -> [a] -> Bool
elem = undefined

f :: a -> a
f = if (elem [1,3..] 2) then (\z -> z)
    else (\x -> x) == (\y -> y)
```

First error message

```
(5,9): Type error in application
 expression        : elem [1, 3 .. ] 2
 term              : elem
   type            : a      -> [a] -> Bool
   does not match  : [Int] -> Int -> Bool
 probable fix      : re-order arguments
```

Multi-example

```
elem :: a -> [a] -> Bool
elem = undefined

f :: a -> a
f = if (elem [1,3..] 2) then (\z -> z)
    else (\x -> x) == (\y -> y)
```

Second error message

```
(6,20): Type error in infix application
 expression       : (\x -> x) == (\y -> y)
 operator         : ==
   type           : Int      -> Int      -> Bool
   does not match : (a -> a) -> (b -> b) -> c -> c
```

Multi-example

```
elem :: a -> [a] -> Bool
elem = undefined

f :: a -> a
f = if (elem [1,3..] 2) then (\z -> z)
    else (\x -> x) == (\y -> y)
```

Second error message

```
(6,20): Type error in infix application
 expression       : (\x -> x) == (\y -> y)
 operator         : ==
   type           : Int      -> Int      -> Bool
   does not match : (a -> a) -> (b -> b) -> c -> c
 probable fix     : use . instead        -- with siblings
```

Tie-breaker example

```
f :: Bool
f = (\x -> x) == (\y -> y)
```

Error message

```
(2,15): Type error in infix application
 expression        : (\x -> x) == (\y -> y)
 operator          : ==
   type            : Int      -> Int      -> Bool
   does not match  : (a -> a) -> (b -> b) -> Bool
```

Trusted constraint tie-breaker avoids suggesting to change type of ==

Heuristics

- Heuristics encapsulate expert knowledge on how to discover mistakes
- Many of them are obvious, folklore, or thought up by others,
  - but we implemented them within an infrastructure,
  - and added some of our own.
- What does a heuristic look like?
  - No time. Check out the Helium compiler or ask me to show you
  - I did give some code lengths earlier on
- How do we apply them?
  - basically a list of heuristic functions
  - applied in the given order
  - but some elements of this list can be lists with a voting mechanism.

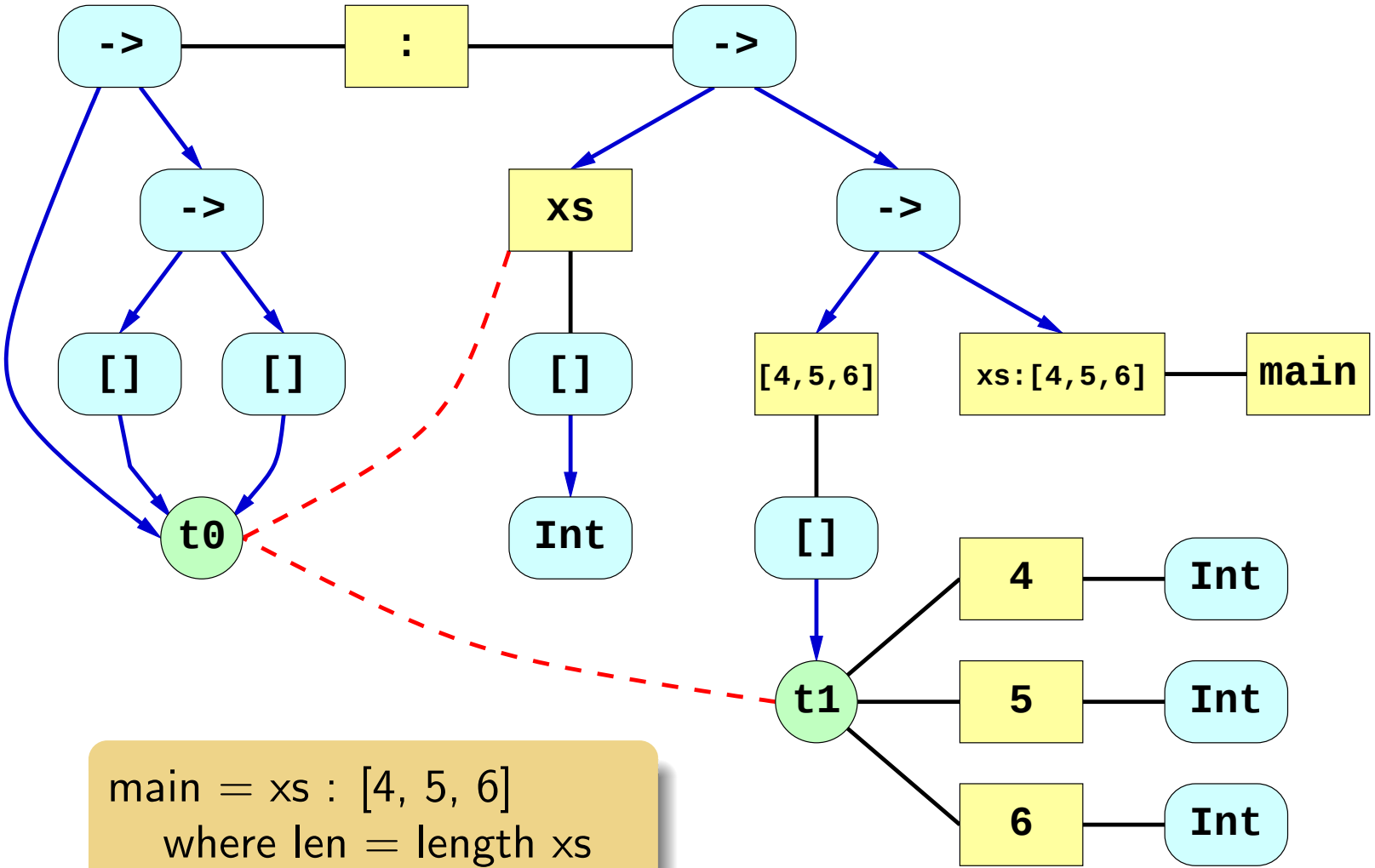
The list in Haskell form

```
listOfHeuristics siblings path =
[ highlyTrustedFilter
, highParticipation 0.95 path
] ++
[ Heuristic (Voting
      [ siblingFunctions siblings
      , similarLiterals
      , applicationEdge
      , tupleEdge
      , fbHasTooManyArguments
      , variableFunction])
] ++
[ applicationResult
, negationResult
, trustFactorOfConstraint
, isTopDownEdge
, positionInList]
```

Type graphs

- Global analysis by considering sets of constraints at the time.
- Polymorphic types have been instantiated.
  - Recall: type graph handles one binding group at the time.
- Indeed, in Helium we usually perform type inferencing greedily, solving one constraint at the time.
- Only when an error occurs in a binding group do we restart type inferencing *only* for that binding group.
- Heuristics are basically graph traversing algorithms
  - And relatively short and simple ones at that

Type graphs



main = xs : [4, 5, 6]
where len = length xs
xs = [1, 2, 3]

How are type graphs used?

- Consider a set of equivalence constraints
- Build the type graph:
  - every type constants, type variable and type application becomes a vertex
  - every constraint becomes an edge
  - but might give rise to derived edges: $a \rightarrow b \equiv b \rightarrow c$
- When two different type constants $(Int, \rightarrow)$ or type application are in the same clique:
  - consider paths between these vertices
  - remove edges (constraints) on these paths
  - until all such paths have disappeared.
- Heuristics help to discover which constraints should be removed.
- When all conflicts are resolved a substitution is easy to compute.

Complications

- Type graphs grow large quickly, and have many derived edges
  - Represent cliques explicitly, merge and split as needed
- Number of error paths grows fast too
  - If error path p contains q then removing a constraint from q also removes p
  - Avoid "detours" (see technical report)
- Infinite types need special treatment (see paper)
- Handling type synonyms

Summary

- We use type graphs for representing sets of equivalence constraints.
- Therefore we can "globally analyze" type constraints.
  - Global $\approx$ per binding group
- We implemented many heuristics, embedded in an infrastructure
  - Adding new heuristics is quite easy, the infrastructure is there
- Heuristics are
  - general ones (weighting),
  - programming language dependent (application), or
  - programmer dependent (siblings)

A small questionnaire

- Validation of our kind of work can only be done experimentally.
- What kind of experiment(ally derived information) would convince you that
  - it works?
  - it is useful?
- Let me know at your convenience.

Any questions?