Constraint Based Type Inferencing in Helium

Jurriaan Hage Bastiaan Heeren
S. Doaitse Swierstra

Institute of Information and Computing Science
Utrecht University
e-mail: {jur,bastiaan,doaitse}@cs.uu.nl

May 14, 2004

The Helium compiler

- Haskell is a higher-order functional programming language similar to Miranda and ML.
- Helium is a maintainable Haskell 98 compiler with a few restrictions.
- A major design criterion is the ability to give good error messages.
- Used in a first year functional programming course.
- Compiled programs are logged.
- Can be used to verify the usefulness of the system.

Types and type systems

- Reasons for type checking
  - discovery of errors before testing,

```
if (very-likely) then
  x+1
else
  x+True
```

  - efficiency (memory, time), and
  - documentation.
- Type systems are practically motivated: type checking should be fast.
- Programmer writes types in his program, compiler checks these.
- Type inferencing allows programmer to drop type annotations.
- Expressiveness of the programming language often leads to cryptic error messages.

A correct program

```
main as bs =
  let
    reverse [] = []
    reverse (x:xs) = (reverse xs) ++ [x]
  in
    (reverse as) : (reverse bs)
```

- In `Haskell`, `:` means cons, `++` is concatenation.
- We can infer that `reverse :: [a] -> [a]`.
- The type variable `'a'` can be any type.

Example 1: avoiding left-to-right bias

```
main as bs =
  let
    reverse [] = True
    reverse (x:xs) = (reverse xs) ++ [x]
  in
    (reverse as) : (reverse bs)
```

ghc (and Hugs)

```
Line 4:
    Couldn't match `[a]' against `Bool'
        Expected type: [a]
        Inferred type: Bool
    In the application `reverse xs'
    In the first argument of `(++)', namely `(reverse xs)'
```

Helium

```
Line (3,10): Type error in right hand side
  term             : True
    type           : Bool
    does not match : [a]
```

Example 2: giving hints

```
main as bs =
  let reverse [] = []
      reverse (x:xs) = (reverse xs) . [x]
  in (reverse as) : (reverse bs)
```

ghc

```
Rev3.hs:3:
   Couldn't match 'b -> c' against '[a]'
       Expected type: b -> c
       Inferred type: [a]
   Probable cause: 'reverse' is applied to too many arguments
       in the call (reverse xs)
   In the first argument of '(.)', namely '(reverse xs)'
```

Helium

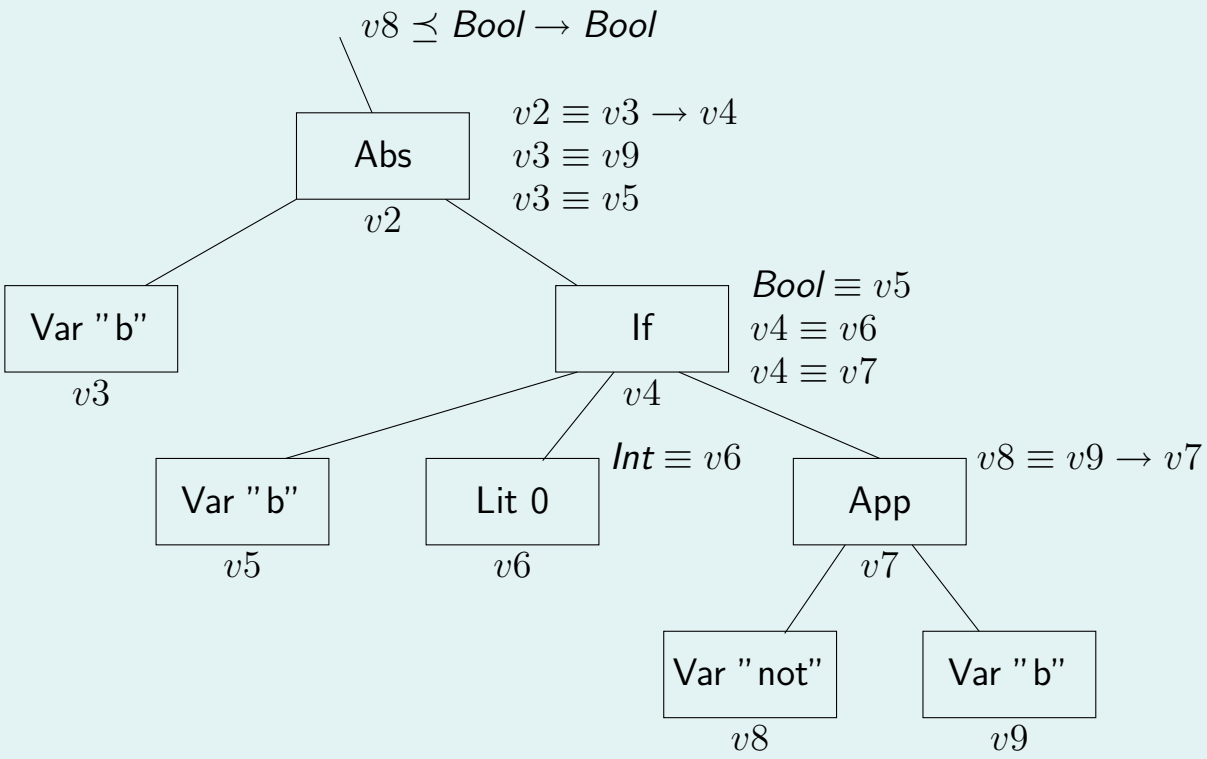
```
(3,23): Type error in variable
 expression        : .
   type            : (a -> b) -> (c -> a) -> c -> b
   expected type   : [d]      -> [e]      -> [d]
 probable fix      : use ++ instead
```

Our contributions

- Separation of type inference process into three different phases:
  1. Generate the constraints in the abstract syntax tree.
  2. Order the constraints into a list.
  3. Solve the constraints.
- There is no single best type inferencer: phase 2 and 3 can be tuned.
- Development of special constraints to deal efficiently and elegantly with let-polymorphism (and explicit types).
- Global approach to solving constraints.
- The use of heuristics for deciding the most likely source of an error.
- Implementing this for a full blown programming language.
- An aside: need not be limited to `Haskell`.

Equality constraints example

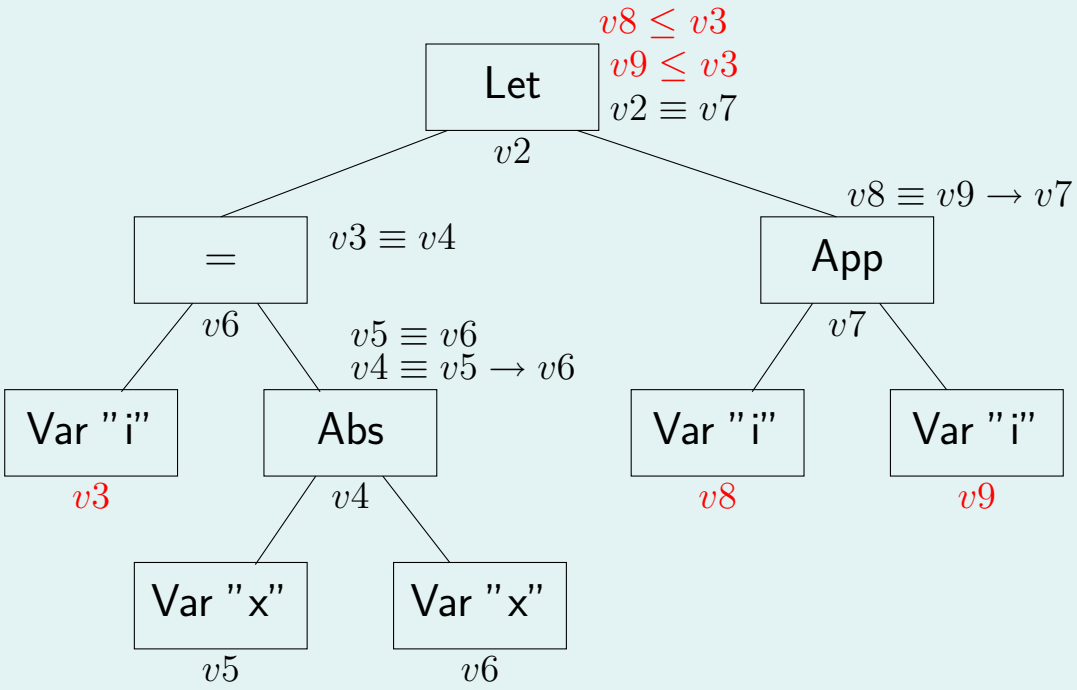
```
\b -> if b then 0 else not b
```



Implicit instance constraints for lets

- `'let i = \x -> x in ..i..i..'` $\Longleftrightarrow$ `'..(\x -> x)..(\x -> x)..'`
- Types of both abstractions derive from the same type $\forall a.a \rightarrow a$, but are generally independent.
- Two choices:
  1. Duplicate constraints for definition of `i`.
  2. Introduce special constraints for this situation.
- Drawbacks first solution:
  - Duplication of effort.
  - Departure from original source makes it more difficult to give good error messages.
- Drawbacks second solution:
  - New sort of constraint is needed (more programming effort).
  - Bias is introduced, because we need to know the (polymorphic) type of `i` before we consider its instances in the body.

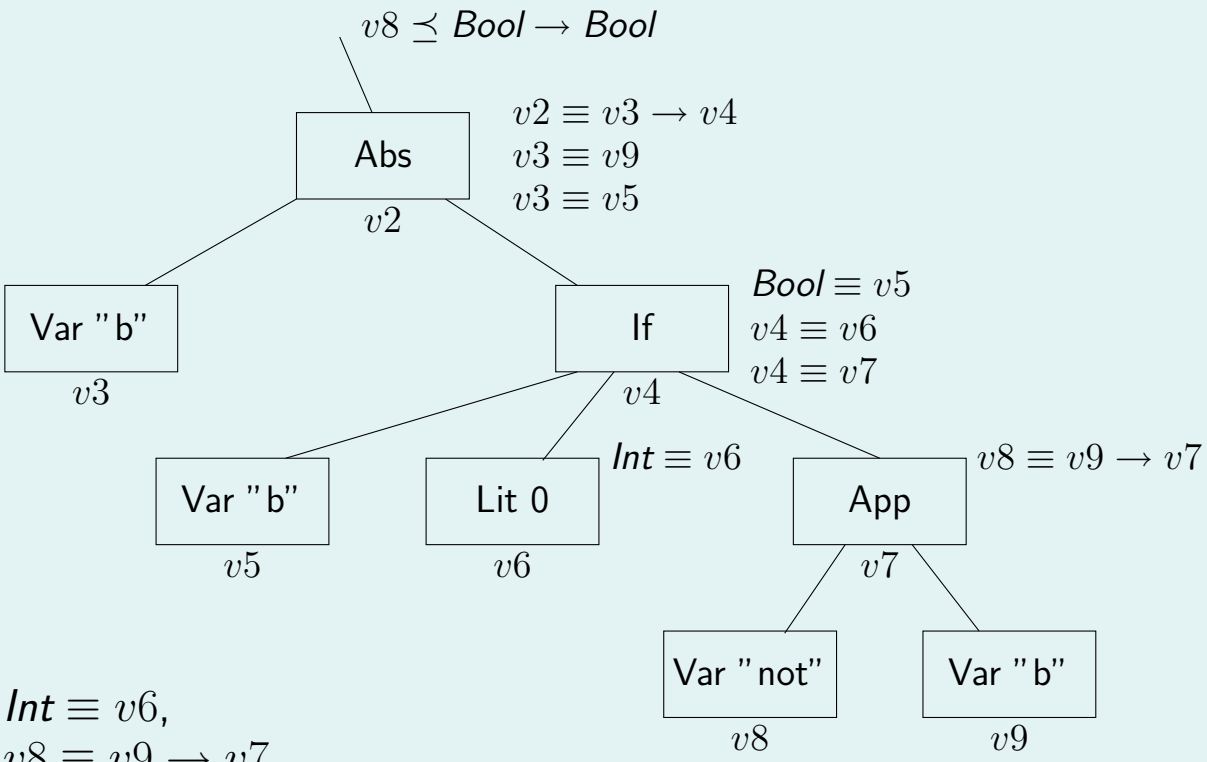
Let example: let i = \ x → x in i i



Ordering the constraints

- Treewalk on constraint tree yields an ordered list of constraints
- Example: post-order, pre-order,...
- Implicit instance constraints like $v3 \leq v8$ are solved by turning them into equality constraints at the right time.
- Doing it after the constraints for the definition of `'i'` are solved works well enough.
- Every treewalk considers constraints of let definitions before those of the body.
- This introduces some bias: we blame the use of a let definition, not the definition itself.

Post-order for \ b → if b then 0 else not b



$Int \equiv v6$,
$v8 \equiv v9 \to v7$,
$Bool \equiv v5$, $v4 \equiv v6$,$v4 \equiv v7$,
$v2 \equiv v3 \to v4$, $v3 \equiv v5$, $v3 \equiv v9$,
$v8 \preceq Bool \to Bool$.

Greedy constraint solving

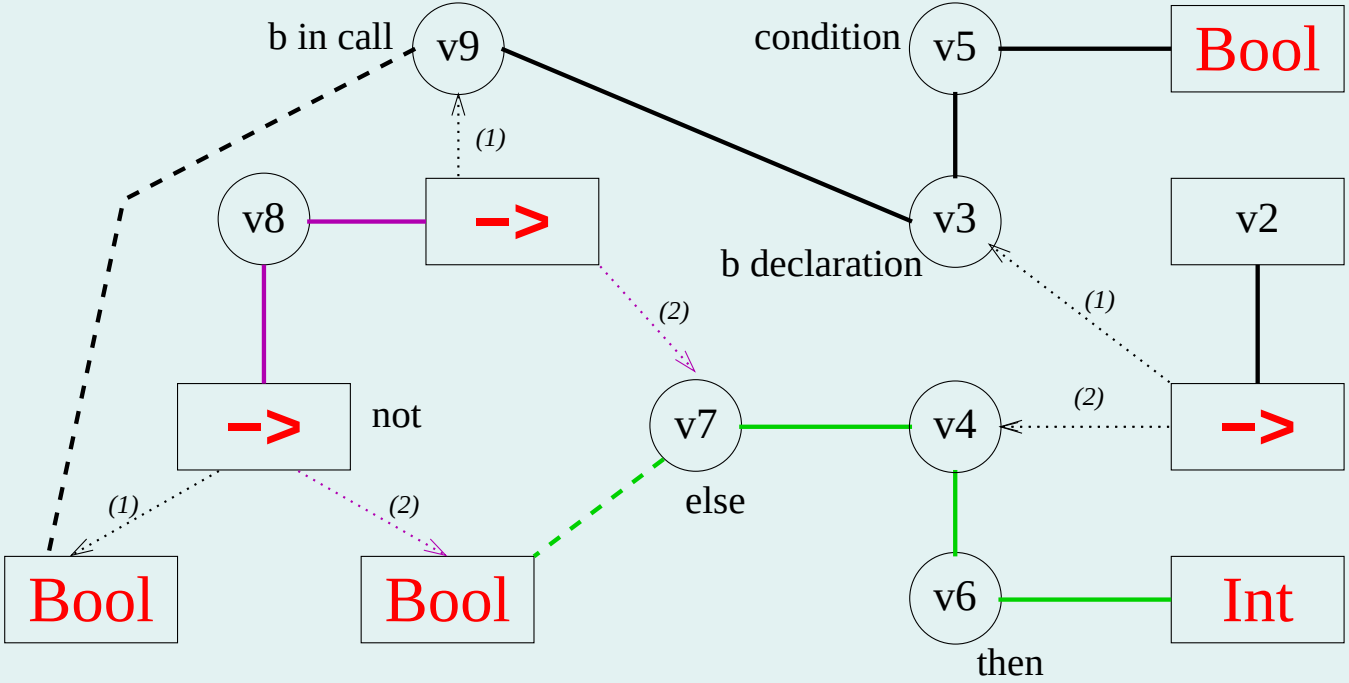
- Take a list of constraints.
- Consider the constraints one by one.
- Update a substitution along the way.
- Inconsistent constraints are ignored, but
- error messages are generated for them.
- $\mathit{Int} \equiv v6$,
  $v8 \equiv v9 \to v7$,
  $\mathit{Bool} \equiv v5$, $v4 \equiv v6$, $v4 \equiv v7$,
  $v2 \equiv v3 \to v4$, $v3 \equiv v5$, $v3 \equiv v9$,
  $v8 \preceq \mathit{Bool} \to \mathit{Bool}$.
- We find fault at $v8 \preceq \mathit{Bool} \to \mathit{Bool}$, because $\mathit{Int} \equiv v6 \equiv v4 \equiv v7 \equiv \mathit{Bool}$.
- A different order may find it sooner: postpone $v2 \equiv v3 \to v4$.

Global constraint solving

- Type graph solver: build a graph from a set of constraints
- Don't forget the order imposed by the lets!
- So we actually build a sequence of such graphs.
- Heuristics work on type graph, yielding a 'most likely source of inconsistencies'
  - Siblings
  - Majority voting
  - Permuting arguments

Type graph example

```
\b -> if b then 0 else not b
```



- Cutting $v4$ from $v7$: the else part is incorrect.
- The constraint $v4 \equiv v7$ generates an error message to this effect.

Future work

- Type inferencing in chunks: combines the speed of the greedy solver with the precision of the type graph solver.
- Adding type inference directives for newly added constructs
- Analyzing logger information
- Adding different kinds of program analysis which may use the same infrastructure/solver.
- Programming environment, possibly including
  - type explanation
  - history sensitive type inferencing
  - easy tuning of the compiler
- Apply our ideas to other programming languages and compilers