Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Strategies for solving constraints in type and effect systems

Jurriaan Hage
jur@cs.uu.nl
joint work with Bastiaan Heeren

Center for Software Technology, Department of Computer and Information Sciences
Universiteit Utrecht
Department of Computer Science, Open University, The Netherlands

September 1, 2008

Overview

Motivation

The basic operators

Non-local reorderings

Properties and implementation

Summary

1. Motivation

Type based program analysis

- Program analysis riding piggy back on type inferencing mechanism.
- Type deduction system: specifies declaratively the (consistent) solutions.
  - Hindley-Milner for polymorphic lambda calculus
- Type inferencer: computes (the best) solution.
  - Folklore Algorithm M, Damas/Milner's algorithm W
- All the algorithms traverse the abstract syntax tree (parse tree) of the program.
- Each algorithm is based on unification.
  - Solving equivalence constraints, like $a \rightarrow Int \equiv Bool \rightarrow b$.
- They differ only in when they perform which unification.
- The same for other validating type based analyses such as Volpano and Smith's Security Analysis.

Main disadvantages

- Turning a deduction system into an algorithm is tedious and error prone.
- Especially if you consider multiple analyses at the same time.
- The order of unification is fixed.
- So why is that a problem?

An example

- Consider $\lambda f \rightarrow (f\ id, f\ True)$
- Algorithm M stops at $True$
  - $True$ does not match the argument type of f which is ...
- Hugs blames the argument id instead.
  - id does not match the argument type of f which is ...
- Algorithm W blames the application $f\ True$.
- A bottom-up algorithm will stop when it considers the different types found for f at the binding site.
- Ad nauseam.

A fixed order of unification

- Different orders of unification should not influence satisfiability.
- If you stay true to the deduction system, this is no problem.
- The ordering of unification does determine which unification is the first to fail.
- Which unification fails determines the error message offered to the programmer.
- In other words, each strategy offers its own view on the problem.
- Disadvantage: committing to a fixed solving strategy also commits you to the corresponding view.
- Moreover, a different algorithm implies different order implies need for a new soundness proof.

So what do we propose?

- Use of special operators in type rules to declaratively specify restrictions and degrees of freedom in performing unification (solving strategy).
- Compiler is not committed to any fixed strategy.
  - Each programmer can select his favourite one or use multiple in parallel.
- Changing strategy can be done without changing or understanding the compiler.
  - The type system, the ordering of unifications, and performing the unifications have been decoupled.
- Supporting various strategies/views in parallel is easy to implement.
- Emulates existing type inference algorithms
  - `helium -X`
  - Also useful for experimentation and comparison

Can we do more to improve error messages? §1

- By implementing heuristics that consider sets of constraints at the time.
- We actually did this for Hindley-Milner (IFL 2006) and Haskell's type classes (PADL 2005).
  - Need to look at O'Sullivan et al, suggested by referee.
- But also here, we use our operators.
- But this approach has drawbacks:
  - Typically quite a bit of effort involved.
  - Language dependent, analysis dependent.
    - Although it does depend on the heuristic
- Our operators live in the world of constraints, and are not tied to any particular language, analysis or language of constraints.

2. The basic operators

The conditional rule with assumption sets

- Associate constraints with nodes in the AST.
- We build a constraint tree, not a constraint set.
- Operator language on top of constraint language.

$$\mathcal{T}_{\mathcal{C}} = [c_1, c_2, c_3] \underline{\Diamond} \{ \mathcal{T}_{\mathcal{C}1}, \mathcal{T}_{\mathcal{C}2}, \mathcal{T}_{\mathcal{C}3} \}$$

$$c_1 = (\tau_1 \equiv \mathit{Bool}) \qquad c_2 = (\tau_2 \equiv \beta) \qquad c_3 = (\tau_3 \equiv \beta)$$

$$\frac{\mathcal{A}_1, \mathcal{T}_{\mathcal{C}1} \vdash e_1 : \tau_1 \qquad \mathcal{A}_2, \mathcal{T}_{\mathcal{C}2} \vdash e_2 : \tau_2 \qquad \mathcal{A}_3, \mathcal{T}_{\mathcal{C}3} \vdash e_3 : \tau_3}{\mathcal{A}_1 +\!\!+ \mathcal{A}_2 +\!\!+ \mathcal{A}_3, \mathcal{T}_{\mathcal{C}} \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \beta}$$

- A strategy turns a constraint tree into a list of constraints.
- Impossible: first c_1, then the subtrees, then $\{c_2, c_3\}$.

Associating constraints with subexpressions. §2

- Some constraints 'belong' to certain subexpressions:

$$\mathcal{T}_{\mathcal{C}} = [c_2, c_3] \underline{\Diamond} \{ c_1 \triangledown \mathcal{T}_{\mathcal{C}1}, \mathcal{T}_{\mathcal{C}2}, \mathcal{T}_{\mathcal{C}3} \}$$

$$c_1 = (\tau_1 \equiv Bool) \quad c_2 = (\tau_2 \equiv \beta) \quad c_3 = (\tau_3 \equiv \beta)$$

$$\frac{\mathcal{A}_1, \mathcal{T}_{\mathcal{C}1} \vdash e_1 : \tau_1 \qquad \mathcal{A}_2, \mathcal{T}_{\mathcal{C}2} \vdash e_2 : \tau_2 \qquad \mathcal{A}_3, \mathcal{T}_{\mathcal{C}3} \vdash e_3 : \tau_3}{\mathcal{A}_1 +\!\!+ \mathcal{A}_2 +\!\!+ \mathcal{A}_3, \mathcal{T}_{\mathcal{C}} \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \beta}$$

- c_1 is generated by the conditional, but associated with the boolean subexpression.
- Example strategy: left-to-right, bottom-up for then and else part, push down $Bool$ (do c_1 before $\mathcal{T}_{\mathcal{C}1}$).

Semantics by tree walk

- Solving strategy is deriveed from the semantics given to the operators.
- Define tree walk (Is code formal enough?)

$$\mathbf{data}\ TreeWalk = TW\ (\forall a.[a] \rightarrow [([a],[a])] \rightarrow [a])$$

- Example strategy of previous slide as a tree walk:

$$\begin{aligned} &treewalk1 = TW\ (\lambda nd\ kids \rightarrow f\ (unzip\ kids) +\!\!+ nd) \\ &\quad \mathbf{where}\ f\ (csets, assocs) = conc\ assocs +\!\!+ conc\ csets \end{aligned}$$

- A function, flatten, uses the strategy to turn a constraint tree to a list of constraints.

$$flatten :: TreeWalk \rightarrow ConstraintTree \rightarrow [Constraint]$$

Extensions: operators parameterised by label §2

- Simple idea: allow different tree walks for different non-terminals
- Haskell interpreter Hugs considers tuples right-to-left, other constructs left-to-right.
- Essentially, *flatten*'s strategy parameter can depend on the AST node:

$$flatten :: (Label \rightarrow TreeWalk) \rightarrow ... \rightarrow [Constraint]$$

The strict operator, $\ll$

- Can we force subexpressions to be done left-to-right?

$$\mathcal{T}_C = [c_2, c_3] \underline{\Diamond} \{ c_1 \triangledown \mathcal{T}_{C1} \ll \mathcal{T}_{C2} \ll \mathcal{T}_{C3} \}$$

$$c_1 = (\tau_1 \equiv Bool) \quad c_2 = (\tau_2 \equiv \beta) \quad c_3 = (\tau_3 \equiv \beta)$$

$$\frac{\mathcal{A}_1, \mathcal{T}_{C1} \vdash e_1 : \tau_1 \qquad \mathcal{A}_2, \mathcal{T}_{C2} \vdash e_2 : \tau_2 \qquad \mathcal{A}_3, \mathcal{T}_{C3} \vdash e_3 : \tau_3}{\mathcal{A}_1 +\!\!+ \mathcal{A}_2 +\!\!+ \mathcal{A}_3, \mathcal{T}_C \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \beta}$$

- Even if we choose a right-to-left treewalk, the conditional will still be inferred left-to-right.
  - *flatten* ignores the treewalk in strict expressions.
- Still, c_1 can be before or after $\mathcal{T}_{C1}$.

Application of $\ll$: let-polymorphism

- When using constraints, basically two solutions for dealing with polymorphism:
  - Duplicate sets of constraints
  - Solve constraints for the definition before it is used.
- Former solution unacceptable: duplication of effort, and worse, of errors.
- The latter solution imposes restrictions on the order of solving constraints.
- Can be handled by making the solver more complicated, but....
- we can also use $\ll$.
- Details are in the paper.

3. Non-local reorderings

Environments versus assumption sets

- Assumption set based type system pass information about identifier upwards.
  - Constraints imposed at binding sites.
- Environment based type systems pass information about declared identifiers downwards.
  - Constraints imposed at identifier uses.
- To mimick environment based systems with ours, we allow to "spread" constraints from binding site to use site.
- $\ll^{\circ}$ allows spreading, but does not enforce it, $\ll$ forbids spreading.
- Similarly for the other operators.
- Before flattening, choose to spread or not.
- Details, again, in the paper.

Phasing

- Use if certain types of constraints always before the others.
- Application: constraints from type signatures before constraints from the definitions themselves.
- Basic idea is simple: associate a phase number with constraints.
- Constraints with low phase number go first.
- Use default phase for constraints, unless stated otherwise.
- Constraints encountered early are blamed less often.
  - Signatures easier to get correct, so first signature constraints.

4. Properties and implementation

Implementation

- Educational compiler Helium in use since 2002.
- Allows (some) experimentation with different orders.
- Efficiency:
  - What kind?
  - Constraints are simpler, but we have more of them.
  - Computing substitution on the fly will be a bit more efficient, but not much.
    - We believe the gain offsets the loss.
  - Comparing algorithms:
    - no difference for correct programs.
    - Lee and Yi: W sees too many constraints, M too few.
    - Seeing constraints takes time, but seeing more might give better message.

Emulation

- Many existing algorithms and implementations can be emulated by choosing the appropriate treewalk.
- For example: algorithm W is emulated by a *bottomUp* strategy combined with *spreading*.
- The type system is always the same.
- Only the interpretation of the operators changes.
- And the choice to spread or not.

Correctness proofs

- Soundness proof (w.r.t. Hindley-Milner)
- A general sketch is given, independent of the analysis and language.
- No actuall proof in the paper. Should it be?
- Full proof in PhD thesis of second author. Give number of pages?
- Proof not essentially more difficult, but it is quite long.
- Some of it can be avoided: mapping assumptions sets back to environments (our "mistake")
- Essentially, we prove correct (most/all) possible algorithms!

Proof ideas/sketch

- Show you get the same solution for every possible treewalk.
- For equivalence constraints solving order irrelevant.
- Correctness essentially depends on our use of $\ll$.
  - Makes sure that generalization and instantiation constraints are not solved before their time has come.
- Proof hinges less on the particular traversal,
- which makes it less arbitrary and more abstract.

5. Summary

Summary

- Bridge the gap between type system specification and type inference implementation.
- High-level operators in the type rules disallow and enable certain unification orders.
- The remaining freedom is for the programmer to fill in.
- Choosing a strategy is done by choosing a semantics for (some of) the proposed operators.
  - As exposed to the programmer through the compiler.
- Solvers may impose certain restrictions on the order in which constraints should be solved.
  - Our operators can be used to assure these restrictions hold.
- Multiple solving "back-ends", multiple strategies (in parallel), multiple independent error messages.
- No need to commit to solving strategy while the compiler is being built.

Questions?

- What should we include to make it more understandable or complete?
  - Actual M and W algorithms?
  - Changing assumption set based to environment based? (Future Work.)
- Give the proofs? How detailed?
- Code for flattening, spreading etc.?
- Reinclude phasing? New feature extension.
- Apply to another validating analysis: Security Analysis? (Future Work.)