Type Processing by Constraint Reasoning

Peter J. Stuckey, Martin Sulzmann, Jeremy Wazny

8th November 2006

Chameleon

Chameleon is Haskell-style language

- treats type problems using constraints
- gives expressive error messages
- has a programmable type system
- Developers: Martin Sulzmann, Jeremy Wazny
- `http://www.comp.nus.edu.sg/~sulzmann/chameleon/`
- Examples taken from Chameleon
  - Caveat
  - Previous versions of Chameleon supported some features
  - No single version shows has all the behaviour we illustrate

What are Types Good for?

- Type systems are important tools in the design, analysis, and verification of programming languages.
- Well-typed programs dont "go wrong" for a reasonable class or errors.
- Type systems need to
  - Check types!
  - (Preferably) Infer types
  - (Definitely) Explain type errors!

Type Systems are Getting More Complex

- "Fortran" types
- Hindley/Milner types: Higher-order, polymorphic
- Type Classes: controlled ad-hoc polymorphism
  - Multiparameter Type Classes: relations among types
- Constructor Classes: classes with higher kinds
- Functional Dependencies: classes with improvement rules
- Existential Types: "objects"
- Lexically Scoped Annotations: more expressive type declarations
- Guarded Abstract Data Types: types differ per constructor
- Extended Abstract Data Types: difference controlled by classes
- ...

Thesis

Understanding types be first mapping them to constraints:

- improves understanding
  - inference = satisfiability
  - checking = implication tests
- eases implementation
- allows more useful error handling
- makes type systems easier to extend

Hindley/Milner Types

- **Types** t ::=
  - variable a
  - function arrow $t \longrightarrow t$
  - base types e.g. $Char$, $Bool$
  - constructor types e.g. $[t]$ (list of t)
- **Type scheme** $\sigma ::= t \mid \forall \bar{a}.t$
- **Primitive Constraint** $p ::= t = t \mid True$
- **Constraint** $C ::= p \mid C \wedge C$

Hindley/Milner Type Inference

```
f u = g u 't' False
g x y z = if x then z else y
```

- Inferring type for g

  $t_x = Bool$ $\quad \{t_x \mapsto Bool\}$

  $t_{ift} = t_z,\ t_{ift} = t_y$ $\quad \{t_x \mapsto Bool, t_{ift} \mapsto t_z, t_y \mapsto t_z\}$

  $t_g = t_x \rightarrow t_y \rightarrow t_z \rightarrow t_{ift}$ $\quad \{t_g \mapsto Bool \rightarrow t_z \rightarrow t_z \rightarrow t_z\}$

- Inferring type for f

  $t_g = Bool \rightarrow t_1 \rightarrow t_1 \rightarrow t_1$ $\quad \{t_g \mapsto Bool \rightarrow t_1 \rightarrow t_1 \rightarrow t_1\}$

  $t_g = t_u \rightarrow t_{gu}$ $\quad \{t_{gu} \mapsto t_1 \rightarrow t_1 \rightarrow t_1, t_u \mapsto Bool\}$

  $t_{gu} = Char \rightarrow t_{gut}$ $\quad \{t_{gut} \mapsto Char \rightarrow Char, t_1 \mapsto Char\}$

  $t_{gut} = Bool \rightarrow t_{body}$ $\quad$ FAILS

Hindley/Milner Type Inference

```
f u = g u 't' False
g x y z = if x then z else y
```

- Error message:

```
Couldn't match `Char' against `Bool'
  Expected type:  Char
  Inferred type:  Bool
In the third argument of `g', namely `False'
In the definition of `f':  f u = g u 't' False
```

- Fixed order of evaluation
- Only substitutions are maintained

Hindley/Milner Type Inference

```
f u = g u 't' False
g x y z = if x then z else y
```

- Potential corrected versions of the program:
  - ```
    f u = g u True False
    g x y z = if x then z else y
    ```
  - ```
    f u = g u 't' False
    g x y z = if z then x else y
    ```
  - ```
    f u = g u 't' 'f'
    g x y z = if x then z else y
    ```
- Only the last correction changes the indicated location!
    ```

Locations

- Collect constraints (not substitutions)
- Remember locations of constraints
  - **Locations** l (integers)
  - **Justification** $J ::= \epsilon \mid l \mid [l, \ldots l]$
  - **Primitive constraint** $p ::= (t = t)_J \mid \mathit{True}$
- Location annotated program

  $\texttt{f u = (((g}_1\ \texttt{u}_2)_3\ \texttt{'t'}_4)_5\ \texttt{False}_6)_7$

  $\texttt{g x y z = (if}_{11}\ \texttt{x}_8\ \texttt{then z}_9\ \texttt{else y}_{10})_{12}$
- Locations Annotated Type Constraints: $(t_1 = t_g)_1$, $(t_2 = t_u)_2$, $(t_1 = t_2 \rightarrow t_3)_3$, $(t_4 = \mathit{Char})_4$, $(t_3 = t_4 \rightarrow t_5)_5$, $(t_6 = \mathit{Bool})_6$, $(t_5 = t_6 \rightarrow t_7)_7$, $t_f = t_u \rightarrow t_7$, $(t_8 = t_x)_8$, $(t_8 = \mathit{Bool})_{11}$, $(t_9 = t_z)_9$, $(t_{10} = t_y)_{10}$, $(t_{12} = t_9)_{12}$, $(t_{12} = t_{10})_{12}$, $t_g = t_x \rightarrow t_y \rightarrow t_z \rightarrow t_{12}$.

Type Error = Unsatisfiable Constraint

- Failure of unification = unsatisfiable constraint
- Minimal unsatisfiable subset: E of constraint D
  - E is unsatisfiable: $\models \neg\tilde{\exists}E$
  - all strict subsets of E are satisfiable: $\forall e \in E. \models \tilde{\exists}(E - \{e\})$
- Minimal unsatisfiable subset = minimal reason for type error!

Displaying Type Errors

- Determine a minimal unsatisfiable subset
- For example: $(t_1 = t_g)_1$, $(t_1 = t_2 \longrightarrow t_3)_3$, $(t_4 = Char)_4$, $(t_3 = t_4 \longrightarrow t_5)_5$, $(t_6 = Bool)_6$, $(t_5 = t_6 \longrightarrow t_7)_7$, $(t_9 = t_z)_9$, $(t_{10} = t_y)_{10}$, $(t_{12} = t_9)_{12}$, $(t_{12} = t_{10})_{12}$, $t_g = t_x \longrightarrow t_y \longrightarrow t_z \longrightarrow t_{12}$.
- Highlight locations of the minimal unsatisfiable subset

```
f u = g u 't' False
g x y z = if x then z else y
```

- Not restricted to a single function definition!
- At least one location highlighted must be changed!

Multiple Minimal Unsatisfiable Subsets

Type information is usually highly redundant.

- `f x = if x then (toUpper x) else (toLower x)`
  where `toUpper,toLower :: Char -> Char`
- `f x = if x then (toUpper x) else (toLower x)`
- `f x = if x then (toUpper x) else (toLower x)`

Locations appearing in all minimal unsatisfiable subsets are more likely to be in error!

- You may be able to remove all errors changing one
  `f x = if x then (toUpper x) else (toLower x)`

Better Error Messages

- Choose a location from the minimal unsatisfiable subset E
- Remove the constraints for that location from E (satisfiable)
- Find conflicting types (via E) at that location
  - Assign a colour to each conflicting type
  - Find the locations causing each conflicting type
- Report the error in terms of locations and conflicting types
- Highlight the causes of each conflicting type

Better Error Messages Example

- Choose a location: 12
- Remove constraints for location: $(t_{12} = t_9)_{12}$, $(t_{12} = t_{10})_{12}$
- Find conflicting types at location: $t_9 = Bool$, $t_{10} = Char$
  - Assign a colour to each conflicting type: t_9, t_{10}
  - Find the locations causing each conflicting type: t_9: [1,3,5,6,9], t_{10}: [1,3,4,10]
- Report the error in terms of locations and conflicting types

```
Problem :  Then and else branch must have same type
Types   :  Bool
           Char
Conflict:  f u = g u 't' False
           g x y z = if x then z else y
```

Find Locations that Cause a Type?

How do we find a minimal set of constraints C that cause another constraint c to hold?

- Implication: $\models C \rightarrow c$
- Analogous to finding minimal unsatisfiable subset.
  - Start with $E := C$
  - Delete a constraint $p \in E$: $E := E - \{p\}$
  - If $\models E \rightarrow c$ continue, otherwise replace
  - Stop when no constraint $p \in E$ can be deleted.

A Type Error may not be just Two Conflicting Types!

```
f 'a' b    True = error "'a'"
f c   True z    = error "'b'"
f x   y    z    = if z then x else y

GHC: Couldn't match 'Char' against 'Bool'
         Expected type: Char
         Inferred type: Bool
     In the definition of 'f': f x y z = if z then x else y

Chameleon:  Problem  : Definition clauses not unifiable
            Types    : Char -> a -> b -> c
                       d -> Bool -> e -> f
                       g -> g -> h -> i
            Conflict: f 'a' b True = error "'a'"
                      f c True z = error "'b'"
                      f x y z = if z then x else y
```

Failure of Type Checking

- Function type declaration as well as definition

```
h :: a -> (a,b)
h x = (x,x)
```

- Inferred type: `h :: c -> (c,c)`
- Find variable bound in the declared type w.r.t. inferred type b
- Find minimal implicant of binding $b = a$
- Highlight it

```
h.hs:2: ERROR: Inferred type does not subsume declared
Declared: forall a,b.  a -> (a,b)
Inferred: forall a.  a -> (a,a)
Problem : The variable ‘b’ makes the declared type too
          h x = (x, x)
```

Type Classes: Controlled Overloading

A class of types that all supply the same interface:

- class declarations defined interface

```
class Num a where
      (+), (-), (*) :: a -> a -> a
      integer constant :: a
```

- instance declarations define members of class

```
instance Num Int where (+) = iplus; (-) = iminus; ...
instance Num Float where (+) = fplus; (-) = fminus; ...
```

- Allows overloading (like coercion) e.g.
- `3.1415 + 2 :: Float`

Type Classes: Controlled Overloading

A class of types that all supply the same interface:

- class declarations defined interface

```
class Eq a where
      (==) :: a -> a -> Bool
```

- instance declarations define members of class

```
instance Eq Int where (==) = eqInt
instance Eq Bool where (==) = eqBool
```

- Overloading `eqList:: Eq a => [a] -> [a] -> Bool`

```
eqList []     []     = True
eqList (x:xs) (y:ys) = (x == y) && eqList xs ys
eqList _      _      = False
```

Superclasses

Classes and instances can require that the members have instances of other classes

- Class heirarchy

```
class Eq a => Ord a where
      (>) :: a -> a -> Bool
```

- Members of `Ord` must be members of Eq
- Instances can also be constrained

```
instance Eq a => Eq [a] where
         (==) = eqList
```

Multi-Parameter Type Classes

- Single parameter type classes implicitly define sets
- Multi-parameter type classes define relationships among types

```
class    Convert a b where
         upcast :: a -> b
         downcast :: b -> Maybe a
instance Convert Int Float where
         upcast = fromInteger
         downcast f = if (ceiling f == floor f)
         then Just (floor f)
         else Nothing
```

- Clearly classes define constraints!

Type Classes

- **Primitive constaint** $p ::= (t = t)_J \;\mid\; U\ \bar{t} \;\mid\; True$
- **Type scheme** $\sigma ::= t \;\mid\; \forall \bar{a}.C \Rightarrow t$
- **Class decl** $cd ::= \texttt{class}\ (C \Rightarrow U\ \bar{a})_l\ \texttt{where}\ [m :: (C \rightarrow t)_l]$
- **Instance decl** $id ::= \texttt{instance}\ (C \Rightarrow U\ \bar{t})_l\ \texttt{where}\ [m = e]$

We need the ability to

- Check satisfiability of constraints
- Check implication of constraints
- Simplify constraints
- Check classes and instances agree!

Constraint Handling Rules (CHRs)

- Lightweight theorem prover for constraints
- Maps a set (conjunction) of constraints to an equivalent set of constraints
- Simplification rules: $c_1, \ldots, c_n \Longleftrightarrow d_1, \ldots, d_m$
  Replace lhs by rhs

`Eq Int <=> True`	`Eq Int` is replaced by True (proved)
`Eq Float <=> True`	`Eq Float` is replaced by True (proved)
`Eq [a] <=> Eq a`	To prove `Eq [a]` prove `Eq a`

- Propagation rules: $c_1, \ldots, c_n \Longrightarrow d_1, \ldots, d_m$
  Add rhs to lhs

`Ord a ==> Eq a`	Proving `Ord a` also requires proving `Eq a`

- Derivation
  $Ord\ [Int] \longrightarrow Ord\ [Int], Eq\ [Int] \longrightarrow Ord\ [Int], Eq\ Int \longrightarrow Ord\ [Int]$

Constraint Handling Rules with Justifications

Constraint operations need to maintain justifications:

- Applying a rule $c_1, \ldots, c_n \Longleftrightarrow d_1, \ldots, d_m$
  - Find matching set $c'_1, \ldots c'_n$
  - Find equations E that force the match (minimal implicant)
  - Collect justifications J for $c'_1, \ldots c'_n$ and E
  - Remove $c'_1, \ldots c'_n$
  - Add $d_1, \ldots, d_m$ extending justifications by J.
- Applying `Eq [a] <=> (Eq a)`$_{15}$ to $(t_1 = Int)_1$, $(t_2 = [t_1])_2$, $(t_3 = t_2 \rightarrow t_4)_4$, $(t_3 = t_5 \rightarrow t_6)_6$, $(Eq\ t_5)_7$.
  - minimal implicant of $\exists a. t_5 = [a]$ is $(t_2 = [t_1])_2$, $(t_3 = t_2 \rightarrow t_4)_4$, $(t_3 = t_5 \rightarrow t_6)_6$
  - $J = [2, 4, 6, 7]$
  - Result $(t_1 = Int)_1$, $(t_2 = [t_1])_2$, $(t_3 = t_2 \rightarrow t_4)_4$, $(t_3 = t_5 \rightarrow t_6)_6$, $(Eq\ t_1)_{[2,4,6,7,15]}$.

CHR Algorithms

Restrictions on CHR programs

- Confluent: all derivations for C lead to the same result
- Terminating: all derivations terminate
- Range-restricted: all vars in $d_1, \ldots, d_m$ appear in $c_1, \ldots, c_n$
- Single-headed simplification: $n = 1$ for simplification rules

Under these restrictions: KEY RESULTS

- The rules define a canonical form: If $P \models C_1 \leftrightarrow C_2$ then $C_1 \longrightarrow^* D$ and $C_2 \longrightarrow^* D$
- Satisfiability is decidable
- Minimal unsatisfiable subsets can be determined
- Implication is decidable
- Minimal implicants can be determined

Representing Typing Problems with CHRs

Use CHRs to represent types

```
f u = (((g₁ u₂)₃ 't'₄)₅ False₆)₇
class (True => Eq a)₁₃ where (==) :: (a -> a -> Bool)₁₄
instance (Eq a => Eq [a])₁₅ where (==) = eqList₁₆
```

is represented as

$$
\begin{array}{l}
F\ t_f \ \texttt{<=>}\ (G\ \ t_1)_1\,,\ (t_2 = t_u)_2\,,\ (t_1 = t_2 \rightarrow t_3)_3\,,\ (t_4 = Char)_4\,,\\
\qquad\qquad (t_3 = t_4 \rightarrow t_5)_5\,,\ (t_6 = Bool)_6\,,\ (t_5 = t_6 \rightarrow t_7)_7\,,\\
\qquad\qquad t_f = t_u \rightarrow t_7\\
Eq\ \ a\ \texttt{==>}\ True_{13}\\
(==)\ \ t_e\ \texttt{<=>}\ (Eq\ \ a)_{14}\,,\ (t = a \rightarrow a \rightarrow Bool)_{14}\\
Eq\ \ [a]\ \texttt{<=>}\ (Eq\ \ a)_{15}
\end{array}
$$

Type Processing using CHRs

- Type inference:
  - $F\ t_f \longrightarrow^* D$
  - Inferred type: $\bar{\exists}_{t_f} D$
  - For example
    - $G\ t_g \longrightarrow^* (t_8 = t_x)_8,\ (t_8 = Bool)_{11},\ (t_9 = t_z)_9,\ (t_{10} = t_y)_{10},\ (t_{12} = t_9)_{12},\ (t_{12} = t_{10})_{12},\ t_g = t_x \rightarrow t_y \rightarrow t_z \rightarrow t_{12}$
    - Inferred type: $\exists t_{12}.t_g = Bool \rightarrow t_{12} \rightarrow t_{12}$
- Type checking: $f :: C \Rightarrow t$
  - $F\ t_f \longrightarrow^* D_1$
  - $t_f = t, C \longrightarrow^* D_2$
  - $\models (\bar{\exists}_{t_f} D_2) \supset (\bar{\exists}_{t_f} D_1)$

Type Errors with Classes

- Failure of inference:
  - only possible through unsatisfiable equational constraints
  - Report as before
- Failure of checking
  - Also possible to have have unmatched type class constraints
  - Extend to report reason (minimal implicant) for these
- New kinds of errors
  - Missing instance errors
  - Ambiguity errors
  - Incompatible `class` and `instance` declarations

Missing Instance Error

- Classic beginners mistake:

```
sum []     = [] -- should be 0
sum (x:xs) = x + sum xs
```

- Inferred type: `sum :: Num [a] => [[a] -> [a]`
- In Haskell 98, each type class constraint appearing in a functions type must be of the form $U\ a$ where a is a type variable.
- For $Num\ [a]$ find a minimal implicant of $Num\ t \wedge \exists t'.t = [t']$
- Highlight the locations of the minimal implicant

```
sum.hs:4: ERROR: Missing instance
Instance:Num [a]: sum []     = [] -- should be 0
                  sum (x:xs) = x + sum xs
```

Ambiguity Error

- `f x y z = show (if x then read y else read z)`
  where `read :: Read a => [Char] -> a`
  and `show :: Show a => a -> [Char]`
- Inferred type: `f :: forall a.(Read a, Show a) => Bool -> [Char] -> [Char] -> [Char]`
- `a` does not appear in type part. Ambiguous
- Highlight the positions where `a` appears in type

  ```
  Ambiguity can be resolved at these locations
  f x y z = show (if x then read y else read z)
  ```
- For example

  ```
  f x y z = show (if x then (read y)::Int else read z)
  ```

Functional Dependencies

```
class Collects ce e | ce -> e where
      empty :: ce
      insert :: e -> ce -> ce
```

- Collection type ce functionally defines element type e
- Without this `empty :: Collects ce e => ce` is ambiguous!

Bibliography

- Theory P. J. Stuckey and M. Sulzmann. A theory of overloading. ACM TOPLAS, 2005.
- EADTs M. Sulzmann, J. Wazny, and P.J. Stuckey. A framework for extended algebraic data types. FLOPS 2006,
- Better error messages P.J. Stuckey, M. Sulzmann, and J. Wazny. Improving type error diagnosis. 2004 ACM Haskell Workshop,
- Functional dependencies G.J. Duck, S. Peyton-Jones, P.J. Stuckey, and M. Sulzmann. Sound and decidable type inference for functional dependencies. ESOP 2004,
- Type errors P.J. Stuckey, M. Sulzmann, and J. Wazny. Interactive type debugging in Haskell. 2003 ACM Haskell Workshop,
- More `http://www.comp.nus.edu.sg/~sulzmann/`