Wildcard Reconstruction

Tom Schrijvers

K.U.Leuven, Belgium
toms@cs.kuleuven.be

February 13, 2007
IBM T.J. Watson Laboratory, Hawthorne, NY, USA

Overview

1. Research Background
2. Motivation
3. Framework
   - Algorithm
   - Properties
   - Extension: Cast Removal
4. Wildcard Inference
   - Algorithm Adaptations
5. Conclusion

Overview

1. Research Background
2. Motivation
3. Framework
   - Algorithm
   - Properties
   - Extension: Cast Removal
4. Wildcard Inference
   - Algorithm Adaptations
5. Conclusion

Research Background

Constraint Logic Programming (CLP)

Ph.D. thesis 2005:

Analyses, Optimizations and Extensions of Constraint Handling Rules

- program analysis (abstract interpretation)
- optimized compilation
- language extensions

CLP applied to type inference:

- Polymorphic ADT Reconstruction (PPDP 2006)
- Inference for Multi-Parameter Type Classes (APLAS 2006)

Constraint Handling Rules

Constraint Handling Rules [Frühwirth'91]

- multi-set rewriting language
- for writing custom constraint solvers (and more)
- executable logic theory

Example

```
:- chr_constraint leq/2.

   reflexive @ leq(X,X) <=> true.
antisymmetric @ leq(X,Y), leq(Y,X) <=> X = Y.
  idempotent @ leq(X,Y) \ leq(X,Y) <=> true.
  transitive @ leq(X,Y), leq(Y,Z) ==> leq(X,Z).
```

Constraint Handling Rules: Execution

Example

```
:- chr_constraint leq/2.

     reflexive @ leq(X,X) <=> true.
antisymmetric @ leq(X,Y), leq(Y,X) <=> X = Y.
   idempotent @ leq(X,Y) \ leq(X,Y) <=> true.
   transitive @ leq(X,Y), leq(Y,Z) ==> leq(X,Z).
```

```
?-  leq(A,B), leq(B,C), leq(C,A)              (trans.)
↣   leq(A,B), leq(B,C), leq(C,A),leq(A,C)     (anti)
↣   leq(A,B), leq(B,A), C = A                 (anti)
↣   A = B, C = A
```

Constraint Handling Rules: Systems

Leuven Systems

- K.U.Leuven CHR (current main system)
  - Prolog: SWI-Prolog, SICStus, Yap, ...
  - JCHR in Java
  - CCHR in C
- language development
  - program analysis, termination
  - optimized compilation
  - language features: aggregates
- complexity results
  - RAM simulator
  - optimal in time & space

Constraint Handling Rules: Applications

Applications

- study of constraint solvers (e.g. lexicographic) [Frühwirth]
- study of declarative algorithms:
  - compexity: union-find, dijkstra [Schrijvers]
  - parallel: union-find, preflow-push [Frühwirth]
- type systems:
  - inference, checking [Sulzmann,Schrijvers]
  - extensible type system [Sulzmann]
- more: agents, semantics web, NLP, verification, . . .
- programming language, e.g. SSS LTD, New Zealand

Overview

1. Research Background
2. Motivation
3. Framework
   - Algorithm
   - Properties
   - Extension: Cast Removal
4. Wildcard Inference
   - Algorithm Adaptations
5. Conclusion

Motivation

Parametric Polymorphism for Java

- $\leq$ Java 1.4:
  - monomorphic libraries,
    e.g. `List list;`
  - many casts,
    e.g. `Integer i = (Integer) list.first();`
  - statically unchecked
- Java 5: **generics**
  - polymorphic libraries,
    e.g. `List<Integer> list;`
  - no/fewer casts
    e.g. `Integer i = list.first();`
  - statically checked

Motivation

Refactoring [Fowler'99]

Improving quality: adding generics to legacy Java code

1. **Library Code**: little
   by hand (e.g. Java 5 libraries),
   automatically [von Dincklage'04]
2. **Application Code**: lots
   automatically [Donovan'04,Fuhrer'05]
   - optimal removal of casts?
   - wildcard types?

Overview

1 Research Background
2 Motivation
3 Framework
- Algorithm
- Properties
- Extension: Cast Removal

4 Wildcard Inference
- Algorithm Adaptations

5 Conclusion

Framework

Start simple...

Insights first

- variant of *Featherweight Java* [Igarashi et al.]
- simple type system: single inheritance, class types only, . . .

...but flexible

Complexity and efficiency later

- extended language, extended type system
- grips on efficiency

Algorithm

The CLP Approach

1. determine unknowns (constraint variables)
2. determine constraints
3. rewrite constraints: propagate and simplify
4. any unknown left? pick one, *label* it and repeat 3

Unknown Type Parameters

Introducing type parameters

- add parameter lists to all types in program
- a fresh unknown type α for every parameter

```
List meth(List l)
{
   return new List();
}
```

$\Longrightarrow$

```
List<α> meth(List<β> l)
{
   return new List<γ>();
}
```

Type Constraints

- subtyping (<:), e.g.

  $$\texttt{Integer} <: \texttt{Number}$$

- equality (=), e.g.

  $$\texttt{Integer} = \texttt{Integer}$$

Type Constraints: Implementation

- `:- chr_constraint subtype/2.`
- `:- chr_constraint equal/2.`

Constraint Derivation

$$\text{(D-M)}\quad \frac{e_0.m(\overline{e}) \qquad mtype(m,[e_0]) = \overline{U} \rightarrow U}{\overline{[e]} <: \overline{U}}$$

$$\text{(D-Rt)}\quad \frac{T\ \ m(\overline{T}\ \overline{x})\{\ \textbf{return}\ e;\}}{[e] <: T}$$

$$\text{(D-Co)}\quad \frac{\textbf{new}\ C\langle\overline{T}\rangle(\overline{e}) \qquad fields(C\langle\overline{T}\rangle) = \overline{U}\ \overline{f};}{\overline{[e]} <: \overline{U}}$$

$$\text{(D-Bc)}\quad \frac{c\langle\overline{T}\rangle \qquad \textbf{class}\ C\langle\overline{X} \triangleleft \overline{B}\rangle \triangleleft N\{\ldots\}}{\overline{T} <: \overline{B}}$$

Constraint Derivation: Implementation

Example

$$(\text{D-CO})\ \frac{\textbf{new}\ C\langle\overline{T}\rangle(\overline{e}) \qquad \mathit{fields}(C\langle\overline{T}\rangle) = \overline{U}\ \overline{f};}{\overline{[e]} <: \overline{U}}$$

Prolog Implementation

```
derive(new(CT,Es),CT) :-
        maplist(derive,Es,TEs),
        field_types(CT,Us),
        maplist(subtype,TEs,Us).
```

Constraint Solving

$$solve = lfp(\rightarrowtail)$$

Rewriting the constraint store: $\sigma \rightarrowtail \sigma'$

- simplify:
  - trivial equations: $\{\alpha = \alpha\} \cup \sigma \rightarrowtail \sigma$
  - ground constraints: $\{c\} \cup \sigma \rightarrowtail c$, if $uv(c) = \emptyset \wedge \vdash c$
- propagate:
  - substitute equations: $\{\alpha = T\} \cup \sigma \rightarrowtail \{\alpha = T\} \cup [T/\alpha]\sigma$
  - add implied constraints: $\sigma \rightarrowtail \sigma \cup \sigma'$, if $\sigma \vdash^P \sigma'$
- detect inconsistency: $\{c\} \cup \sigma \rightarrowtail \mathit{fail}$, if $uv(c) = \emptyset \wedge \nvdash c$

Constraint Solving: Implementation

CHR Rules

```
equal(X,Y) <=> unify_with_occurs_check(X,Y).

subtype(X,Y), subtype(Y,X) ==> equal(X,Y).

subtype(X,Y), subtype(Y,Z) ==> subtype(X,Z).

subtype(X,class(C,P1)), subtype(X,class(C,P2))
        ==> equal(P1,P2).

...
```

Labeling

Labeling = eliminate unknown type

$$\sigma \rightarrowtail \sigma \cup \{\alpha = T\}$$

1. **select variable**: pick a remaining unknown type α where $(\alpha = T) \notin \sigma$
2. **select value**: choose a candidate type to assign $\alpha = T_1$ where $\forall (L <: \alpha), (\alpha <: U) \in \sigma : L <: T_1 <: U$
3. **backtrack**: try different type upon failure $\alpha = T_2, T_3, \ldots$

Properties

Desirable Properties

- **Soundness**: by construction
  type constraints are enforced
- **Completeness**: problematic
- **Termination**: depends on completeness

Completeness

Infinite number of solutions!

Unconstrained unknown parameter α:

- $\alpha =$ `Object`
- $\alpha =$ `List<Object>`
- $\alpha =$ `List<List<Object>>`
- $\alpha =$ `List<List<List<Object>>>`
- ...

Completeness

Solution: Label with upper bound

$$\alpha = \bigsqcap \{T | \alpha <: T\}$$

Advantages:

- it always exists (possibly `Object`)
- it is unique (least upper bound)
- it is sufficient to obtain a solution
- it makes the algorithm terminate

e.g. $\alpha = \bigsqcap \{\texttt{Integer}, \texttt{Number}, \texttt{Object}\} = \texttt{Integer}$

Cast Removal

Goal: choose parameters to render casts redundant

In Java 1.4:

```
(Foo) new List().add(new Foo()).first()
```

becomes in Java 5:

```
(Foo) new List<Foo>().add(new Foo()).first()
```

or simply:

```
new List<Foo>().add(new Foo()).first()
```

Cast Removal

Soft subtyping constraint

- derive soft constraint:

$$(\text{D-CA})\ \frac{(N)_i\ e}{[e] \vec{<:}_i N}$$

- consider soft constraints for labeling:

$$\alpha = (\bigsqcap \{T | \alpha <: T\}) \sqcap (\bigsqcap Sat)$$

where $Sat \subseteq \{T | \alpha \vec{<:} T\}$

- best solution is optimal!

Overview

1. Research Background
2. Motivation
3. Framework
    - Algorithm
    - Properties
    - Extension: Cast Removal
4. Wildcard Inference
    - Algorithm Adaptations
5. Conclusion

Wildcard Types

Problem: No variance for generic parameters

$$C\langle T\rangle <: C\langle S\rangle \Leftrightarrow T = S$$

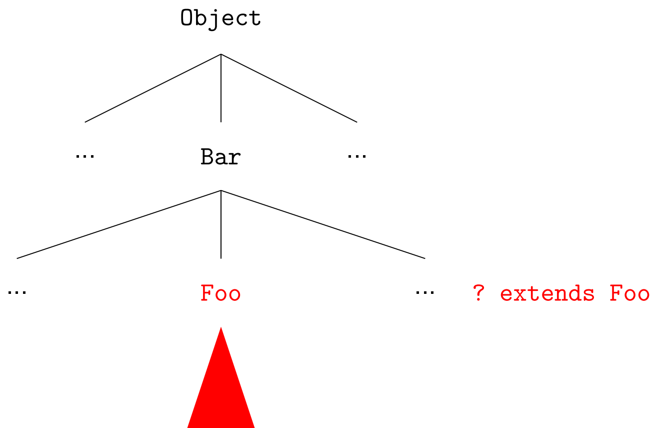
Solution: wildcard types [Torgersen et al. 2004]

$$C\langle T\rangle <: C\langle S\rangle \Leftrightarrow T \subset: S$$

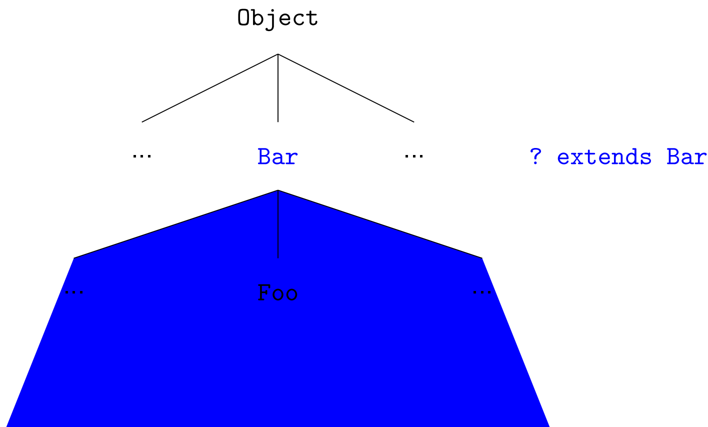
Three forms of wildcard parameters [JLS]:

1. unbounded: `?` (≡ `? extends Object`)
2. bounded from above: `? extends Foo`
3. bounded from below: `? super Foo`

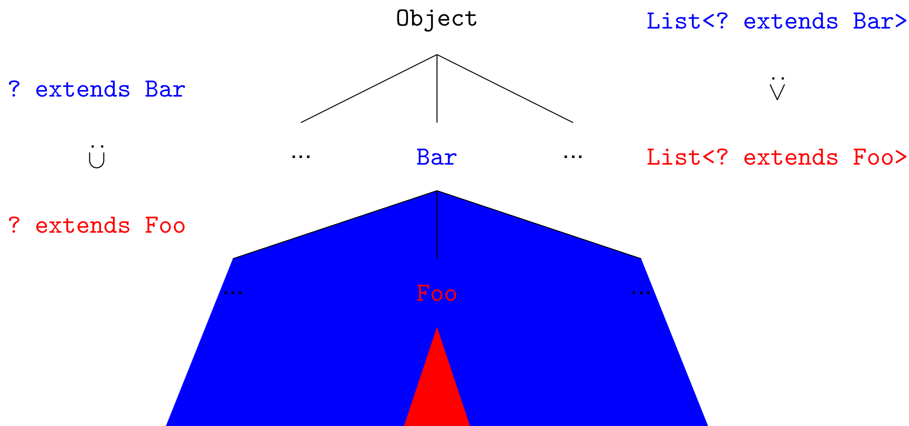
Wildcard Types = Type Ranges



Wildcard Types = Type Ranges



Range Containment ⊂:



Type Constraints

- containment ($\subset:$), e.g.

$$\texttt{? extends Integer} \subset: \texttt{? extends Number}$$

- wildcard captures (*capture*), e.g.

$$\mathit{capture}(\texttt{List<?>}, \texttt{List<X>})$$

- environment subtype ($\prec:$), e.g.

$$\texttt{X} \prec: \texttt{Object} \text{ where } \mathit{capture}(\texttt{List<?>}, \texttt{List<X>})$$

Constraint Derivation

$$e_0.m(\overline{e})$$

$$(\text{D-M})\quad \frac{T_{0,c} \text{ fresh} \qquad \overline{T_c} \text{ fresh} \qquad mtype(m, T_{0,c}) = \overline{U} \rightarrow U}{capture([e_0], T_{0,c}) \qquad capture([\overline{e}], \overline{T_c}) \qquad \overline{T_c} <: \overline{U}}$$

$$(\text{D-RT})\quad \frac{T\ m(\overline{T}\ \overline{x})\{\ \texttt{return}\ e;\} \qquad T_c \text{ fresh}}{capture([e], T_c) \qquad T_c <: T}$$

$$(\text{D-Co})\quad \frac{\begin{array}{c}\texttt{new}\ C\langle\overline{T}\rangle(\overline{e}) \qquad \texttt{class}\ C\langle\overline{X} \triangleleft \overline{B}\rangle \triangleleft N\{\ldots\} \\ fields(C\langle\overline{T}\rangle) = \overline{T_f}\ \overline{f} \qquad \overline{T_e} \text{ fresh}\end{array}}{\begin{array}{c}\overline{T} <: \overline{B} \qquad nowildcard(\overline{T}) \\ capture([\overline{e}], \overline{T_e}) \qquad \overline{T_e} <: \overline{T_f}\end{array}}$$

Constraint Solving

Additional propagation rules, e.g.

$$(\textsc{NoW-WLow})\ \frac{nowildcard(? \rhd T)}{fail}$$

$$(\textsc{NoW-WUp})\ \frac{nowildcard(? \lhd T)}{fail}$$

Labeling

Terminating

Only consider:

1. a class type as before
2. `? extends` $\sqcap\{T|\alpha <: T\}$
3. `? super` $\sqcup\{T|T <: \alpha\}$

No longer unique!

- multiple forms possible: 1-2 and 1-3
- non-unique lub/glb, e.g.
  $\sqcup$ `List<Number>`, `List<Integer>` =
  {`List<? extends Number>`, `List<? super Integer>`}

Overview

1. Research Background
2. Motivation
3. Framework
   - Algorithm
   - Properties
   - Extension: Cast Removal
4. Wildcard Inference
   - Algorithm Adaptations
5. Conclusion

Conclusion

Contributions

1. CLP-based framework for parameter inference
2. optimality of cast removal
3. inference of wildcards types

Future Work

- extend the type language
  - **multiple inheritance**
  - covariant array types, primitive types, ...
- extend the problem scope
  - no distinction between libraries and applications
  - inference of non-parameter types
- optimize the solution (CLP algorithm)
  - add redundant propagation/simplification, faster failing
  - avoid redundant search: pruning
  - search heuristics: value/variable selection

The End

Thank you

- questions?
- comments!