

Subtyping Existential Types

Stefan Wehr Peter Thiemann

University of Freiburg, Germany

FTfJP; July 8, 2008; Paphos, Cyprus

- What are existential types good for?
 - Data abstraction and information hiding
 - Models for OO languages (Bruce, Cardelli, Pierce 1999)
 - Encoding of Java wildcards (Scala; WildFJ; \exists J; TameFJ)
 - Encoding of interface types (LOOM's #-types; JavaGI)

- What are existential types good for?
 - Data abstraction and information hiding
 - Models for OO languages (Bruce, Cardelli, Pierce 1999)
 - Encoding of Java wildcards (Scala; WildFJ; \exists J; TameFJ)
 - Encoding of interface types (LOOM's #-types; JavaGI)
- Why subtyping on existential types?
 - Integration with subtyping in OO languages
 - Avoid explicit pack/unpack operations for existential types

- What are existential types good for?
 - Data abstraction and information hiding
 - Models for OO languages (Bruce, Cardelli, Pierce 1999)
 - Encoding of Java wildcards (Scala; WildFJ; \exists J; TameFJ)
 - Encoding of interface types (LOOM's #-types; JavaGI)
- Why subtyping on existential types?
 - Integration with subtyping in OO languages
 - Avoid explicit pack/unpack operations for existential types

Investigate decidability of subtyping with existential types

Constrained Existential Types

- General form: $\exists \bar{X} \text{ where } \bar{P} . T$
- Choices for constraint P_i :
 - Upper bound constraint: $X \text{ extends } T$
 - Lower bound constraint: $X \text{ super } T$
 - Implementation constraint: $X \text{ implements } \langle \bar{T} \rangle$

Constrained Existential Types

- General form: $\exists \bar{X} \text{ where } \bar{P} . T$
- Choices for constraint P_i :
 - Upper bound constraint: $X \text{ extends } T$
 - Lower bound constraint: $X \text{ super } T$
 - Implementation constraint: $X \text{ implements } \langle \bar{T} \rangle$
- Examples:
 - $\exists Y \text{ where } Y \text{ extends } \text{Shape} . \text{List}\langle Y \rangle$
(encodes the wildcard type `List<? extends Shape>`)
 - $\exists Y \text{ where } Y \text{ super } \text{Integer} . \text{Comparable}\langle Y \rangle$
(encodes the wildcard type `Comparable<? super Integer>`)

Constrained Existential Types

- General form: $\exists \bar{X} \text{ where } \bar{P} . T$
- Choices for constraint P_i :
 - Upper bound constraint: $X \text{ extends } T$
 - Lower bound constraint: $X \text{ super } T$
 - Implementation constraint: $X \text{ implements } \langle \bar{T} \rangle$
- Examples:
 - $\exists Y \text{ where } Y \text{ extends } \text{Shape} . \text{List}\langle Y \rangle$
(encodes the wildcard type `List<? extends Shape>`)
 - $\exists Y \text{ where } Y \text{ super } \text{Integer} . \text{Comparable}\langle Y \rangle$
(encodes the wildcard type `Comparable<? super Integer>`)

Encoding in the reverse direction not always possible:

$$\exists Y . \text{Pair}\langle Y, Y \rangle \not\approx \text{Pair}\langle ?, ? \rangle \approx \exists X, Y . \text{Pair}\langle X, Y \rangle$$

Subtyping on Constrained Existential Types

- Between two existential types

- $\exists Y \text{ where } Y \text{ extends Square} . \text{List}\langle Y \rangle$
 $\leq \exists Y \text{ where } Y \text{ extends Shape} . \text{List}\langle Y \rangle$
- $\exists Y \text{ where } Y \text{ super Number} . \text{Comparable}\langle Y \rangle$
 $\leq \exists Y \text{ where } Y \text{ super Integer} . \text{Comparable}\langle Y \rangle$

Subtyping on Constrained Existential Types

- Between two existential types

- $\exists Y \text{ where } Y \text{ extends Square} . \text{List}\langle Y \rangle$
 $\leq \exists Y \text{ where } Y \text{ extends Shape} . \text{List}\langle Y \rangle$
- $\exists Y \text{ where } Y \text{ super Number} . \text{Comparable}\langle Y \rangle$
 $\leq \exists Y \text{ where } Y \text{ super Integer} . \text{Comparable}\langle Y \rangle$

- Between a non-existential and an existential type

- $\text{List}\langle \text{Square} \rangle$
 $\leq \exists Y \text{ where } Y \text{ extends Shape} . \text{List}\langle Y \rangle$
- $\text{Comparable}\langle \text{Number} \rangle$
 $\leq \exists Y \text{ where } Y \text{ super Integer} . \text{Comparable}\langle Y \rangle$

Subtyping on Constrained Existential Types

- Between two existential types

- $\exists Y$ **where** Y **extends** `Square` . `List<Y>`
 \leq $\exists Y$ **where** Y **extends** `Shape` . `List<Y>`
- $\exists Y$ **where** Y **super** `Number` . `Comparable<Y>`
 \leq $\exists Y$ **where** Y **super** `Integer` . `Comparable<Y>`

- Between a non-existential and an existential type

- `List<Square>`
 \leq $\exists Y$ **where** Y **extends** `Shape` . `List<Y>`
- `Comparable<Number>`
 \leq $\exists Y$ **where** Y **super** `Integer` . `Comparable<Y>`

- Between an existential and a non-existential type

- $\exists Y$ **where** Y **extends** `Shape` . `ArrayList<Y>`
 \leq `Cloneable`
- $\exists Y$ **where** Y **super** `Integer` . `Comparable<Y>`
 \leq `Object`

Subtyping Rules for Constrained Existential Types

$$\frac{\Delta, \bar{P} \vdash U \leq T \quad \bar{X} \cap \text{ftv}(\Delta, T) = \emptyset}{\Delta \vdash \exists \bar{X} \text{ where } \bar{P} . U \leq T} \text{ OPEN}$$

$$\frac{T = [\bar{V}/\bar{X}]U \quad (\forall i) \Delta \Vdash [\bar{V}/\bar{X}]P_i}{\Delta \vdash T \leq \exists \bar{X} \text{ where } \bar{P} . U} \text{ ABSTRACT}$$

- Judgment $\Delta \Vdash P$ denotes **constraint entailment**
- Definition of $\Delta \Vdash P$ depends on the choice of P

- Rule for upper bound constraints:
$$\frac{\Delta \vdash T \leq U}{\Delta \Vdash T \text{ extends } U}$$
- Rule for lower bound constraints:
$$\frac{\Delta \vdash U \leq T}{\Delta \Vdash T \text{ super } U}$$

$\mathcal{E}\mathcal{X}_{uplo}$: Existential Types with Upper and Lower Bounds

- Such existential types occur in practice:
 - Scala
 - Wildcard encodings (WildFJ, TameFJ)
- Syntax:

$$\begin{aligned} T, U, V &::= X \mid N \mid \exists \bar{X} \textbf{where} \bar{P} . N \\ N &::= C \langle \bar{X} \rangle \mid \textit{Object} \\ P &::= X \textbf{extends} T \mid X \textbf{super} T \end{aligned}$$

- Subtyping:

$$\begin{array}{c} \text{REFL} \\ \Delta \vdash T \leq T \end{array} \quad \frac{\text{TRANS} \quad \Delta \vdash T \leq U \quad \Delta \vdash U \leq V}{\Delta \vdash T \leq V} \quad \text{OBJECT} \quad \Delta \vdash T \leq \textit{Object} \quad \frac{\text{EXTENDS} \quad X \textbf{extends} T \in \Delta}{\Delta \vdash X \leq T}$$
$$\frac{\text{SUPER} \quad X \textbf{super} T \in \Delta}{\Delta \vdash T \leq X}$$

OPEN and ABSTRACT as before

A Sample Subtyping Derivation

$\neg T = \exists Y$ **where** Y **super** T . $D\langle Y \rangle$

$U = \exists Y$ **where** Y **extends** $\neg C\langle Y \rangle$. $C\langle Y \rangle$

X **extends** $\neg U \vdash X \leq \neg C\langle X \rangle$

A Sample Subtyping Derivation

$\neg T = \exists Y$ **where** Y **super** T . $D\langle Y \rangle$

$U = \exists Y$ **where** Y **extends** $\neg C\langle Y \rangle$. $C\langle Y \rangle$

X extends $\neg U, Z$ super $U \vdash X \leq \neg C\langle X \rangle$
X extends $\neg U, Z$ super $U \vdash X$ extends $\neg C\langle X \rangle$
X extends $\neg U, Z$ super $U \vdash C\langle X \rangle \leq U$
X extends $\neg U, Z$ super $U \vdash C\langle X \rangle \leq Z$
X extends $\neg U, Z$ super $U \vdash Z$ super $C\langle X \rangle$
X extends $\neg U, Z$ super $U \vdash D\langle Z \rangle \leq \neg C\langle X \rangle$
X extends $\neg U \vdash \neg U \leq \neg C\langle X \rangle$
X extends $\neg U \vdash X \leq \neg C\langle X \rangle$

Subtyping in \mathcal{EX}_{uplo} is undecidable

- Proof by reduction from F_{\leq}^D , an undecidable fragment of F_{\leq} (Pierce 1994)
- Encodes F_{\leq} 's contra-/covariant function type constructor through lower/upper bounds
- See the extended version of the paper for details
- Subtyping in F_{\leq} behaves “well in practice”. Is this also the case for subtyping in \mathcal{EX}_{uplo} ? Experience with Scala?

A Brief Detour: JavaGI

- Conservative extension of Java 1.5
- Generalizes Java's interface mechanism
- Incorporates the essential ideas of Haskell type classes
- Features:
 - Retroactive interface implementations
(decouple interface implementations from class definitions)
 - Implementation constraints
 - Constrained existential types
 - Self-types
 - ...
- See our paper at ECOOP 2007

Retroactive Interface Implementations in JavaGI

- Illegal in Java:

```
for (Character c : someString) { ... }
```

- Reason: String does not implement Iterable
- JavaGI allows the retroactive implementation of Iterable:

```
implementation Iterable<Character> [String] {  
  public Iterator<Character> iterator() {  
    return new Iterator<Character>() {  
      private int index = 0;  
      public boolean hasNext() {  
        return index < length();  
      }  
      public Character next() {  
        return charAt(index++);  
      }  
    };  
  }  
}
```

- Implementation Constraints in JavaGI

- X **implements** $I<\bar{T}>$ states that X implements interface $I<\bar{T}>$
- More powerful than upper bound X **extends** $I<\bar{T}>$ in interaction with self-types

- Existentials in JavaGI

- Encode and generalize interface types
- $\exists Y$ **where** Y **implements** `List<String>` . Y encodes the interface type `List<String>`
- $\exists Y$ **where** Y **implements** `List<String>`,
 Y **implements** `Set<String>` . Y

is the intersection of the interface types `List<String>` and `Set<String>`

$\mathcal{E}\mathcal{X}_{impl}$: Existential Types with Implementation Constraints

- Essentials of constraint entailment and subtyping in JavaGI
- Syntax:

$$\begin{aligned} T, U, V &::= X \mid \exists X \text{ where } \bar{P}. X \\ P &::= X \text{ implements } \langle \bar{T} \rangle \\ def &::= \text{implementation} \langle \bar{X} \rangle \langle \bar{T} \rangle [T] \end{aligned}$$

- Constraint entailment:

$$\begin{array}{c} \text{IMPL} \\ \text{implementation} \langle \bar{X} \rangle \langle \bar{T} \rangle [U] \\ \hline \Delta \Vdash [V/X] (U \text{ implements } \langle \bar{T} \rangle) \end{array} \qquad \begin{array}{c} \text{ENV} \\ P \in \Delta \\ \hline \Delta \Vdash P \end{array}$$

- Subtyping:

$$\begin{array}{c} \text{REFL} \\ \Delta \vdash T \leq T \end{array} \qquad \begin{array}{c} \text{TRANS} \\ \Delta \vdash T \leq U \quad \Delta \vdash U \leq V \\ \hline \Delta \vdash T \leq V \end{array}$$
$$\begin{array}{c} \text{OPEN} \\ \Delta, \bar{P} \vdash X \leq T \quad X \notin \text{ftv}(\Delta, T) \\ \hline \Delta \vdash \exists X \text{ where } \bar{P}. X \leq T \end{array} \qquad \begin{array}{c} \text{ABSTRACT} \\ (\forall i) \Delta \Vdash [T/X] P_i \\ \hline \Delta \vdash T \leq \exists X \text{ where } \bar{P}. X \end{array}$$

Subtyping in \mathcal{EX}_{impl} is undecidable

- Proof by reduction from PCP
- Similar proof technique as used by Kennedy and Pierce (FOOL/WOOD 2007)
- Relies on implementation definitions for existential types

- Subtyping for existentials with lower and upper bounds is undecidable
 - Scala's subtyping relation with existentials probably undecidable
 - Does not imply that subtyping for Java wildcards is undecidable
- Subtyping for existentials with implementation constraints is undecidable
 - JavaGI's subtyping relation is undecidable
 - Revised design of JavaGI without existentials
 - Several other language features make up for the lack of existentials
 - Type system of revised design is decidable and simpler than the system with existentials