Subtyping Existential Types

Stefan Wehr   Peter Thiemann

University of Freiburg, Germany

FTfJP; July 8, 2008; Paphos, Cyprus

Motivation

- 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; ∃J; TameFJ)
  - Encoding of interface types (LOOM's #-types; JavaGI)

Motivation

- 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; ∃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

Motivation

- 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; ∃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: $\exists\, \overline{X}$ **where** $\overline{P}\ .\ T$
- Choices for constraint P_i:
  - Upper bound constraint: X **extends** T
  - Lower bound constraint: X **super** T
  - Implementation constraint: X **implements** $I{<}\overline{T}{>}$

Constrained Existential Types

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

Constrained Existential Types

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

Encoding in the reverse direction not always possible:

$$\exists\, \texttt{Y . Pair<Y,Y>} \not\approx \texttt{Pair<?,?>} \approx \exists\, \texttt{X,Y . Pair<X,Y>}$$

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>`

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>

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

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

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

- Judgment $\Delta \Vdash P$ denotes **constraint entailment**
- Definition of $\Delta \Vdash P$ depends on the choice of P
  - Rule for upper bound constraints: $\dfrac{\Delta \vdash T \leq U}{\Delta \Vdash T\ \texttt{extends}\ U}$
  - Rule for lower bound constraints: $\dfrac{\Delta \vdash U \leq T}{\Delta \Vdash T\ \texttt{super}\ U}$

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

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

$$
\begin{aligned}
T, U, V &::= X \mid N \mid \exists\,\overline{X}\ \mathbf{where}\ \overline{P}\ .\ N \\
N &::= C\texttt{<}\overline{X}\texttt{>} \mid \mathit{Object} \\
P &::= X\ \mathbf{extends}\ T \mid X\ \mathbf{super}\ T
\end{aligned}
$$

- Subtyping:

$$
\text{REFL}\quad \Delta \vdash T \leq T
\qquad
\text{TRANS}\quad \frac{\Delta \vdash T \leq U \qquad \Delta \vdash U \leq V}{\Delta \vdash T \leq V}
\qquad
\text{OBJECT}\quad \Delta \vdash T \leq \mathit{Object}
\qquad
\text{EXTENDS}\quad \frac{X\ \mathbf{extends}\ T \in \Delta}{\Delta \vdash X \leq T}
$$

$$
\text{SUPER}\quad \frac{X\ \mathbf{super}\ T \in \Delta}{\Delta \vdash T \leq X}
$$

OPEN and ABSTRACT as before

A Sample Subtyping Derivation

$\neg\, T = \exists$ `Y` **`where`** `Y` **`super`** T`.` `D<Y>`

$U = \exists$ `Y` **`where`** `Y` **`extends`** $\neg$`C<Y>` `.` `C<Y>`

`X` **`extends`** $\neg\, U \vdash$ `X` $\leq \neg$`C<X>`

A Sample Subtyping Derivation

$$\neg T = \exists\ \texttt{Y}\ \textbf{where}\ \texttt{Y}\ \textbf{super}\ T.\ \texttt{D<Y>}$$

$$U = \exists\ \texttt{Y}\ \textbf{where}\ \texttt{Y}\ \textbf{extends}\ \neg\texttt{C<Y>}\ .\ \texttt{C<Y>}$$

$$\begin{array}{c}
\texttt{X}\ \textbf{extends}\ \neg U, \texttt{Z}\ \textbf{super}\ U \vdash \texttt{X} \leq \neg\texttt{C<X>} \\
\hline
\texttt{X}\ \textbf{extends}\ \neg U, \texttt{Z}\ \textbf{super}\ U \Vdash \texttt{X}\ \textbf{extends}\ \neg\texttt{C<X>} \\
\hline
\texttt{X}\ \textbf{extends}\ \neg U, \texttt{Z}\ \textbf{super}\ U \vdash \texttt{C<X>} \leq U \\
\hline
\texttt{X}\ \textbf{extends}\ \neg U, \texttt{Z}\ \textbf{super}\ U \vdash \texttt{C<X>} \leq \texttt{Z} \\
\hline
\texttt{X}\ \textbf{extends}\ \neg U, \texttt{Z}\ \textbf{super}\ U \Vdash \texttt{Z}\ \textbf{super}\ \texttt{C<X>} \\
\hline
\texttt{X}\ \textbf{extends}\ \neg U, \texttt{Z}\ \textbf{super}\ U \vdash \texttt{D<Z>} \leq \neg\texttt{C<X>} \\
\hline
\texttt{X}\ \textbf{extends}\ \neg U \vdash \neg U \leq \neg\texttt{C<X>} \\
\hline
\texttt{X}\ \textbf{extends}\ \neg U \vdash \texttt{X} \leq \neg\texttt{C<X>}
\end{array}$$

Undecidability of Subtyping in $\mathcal{EX}_{uplo}$

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 & Existentials in JavaGI

- Implementation Constraints in JavaGI
  - X **implements** $I<\overline{T}>$ states that X implements interface $I<\overline{T}>$
  - More powerful then upper bound X **extends** $I<\overline{T}>$ in interaction with self-types
- Existentials in JavaGI
  - Encode and generalize interface types
  - `∃ Y where Y implements List<String> . Y` encodes the interface type `List<String>`
  - ```
    ∃ Y where Y implements List<String>,
              Y implements Set<String> . Y
    ```
    is the intersection of the interface types `List<String>` and `Set<String>`
    ```

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

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

$$\begin{aligned} T, U, V &::= X \mid \exists\, X\ \mathtt{where}\ \overline{P}\ .\ X \\ P &::= X\ \mathtt{implements}\ I\texttt{<}\overline{T}\texttt{>} \\ \mathit{def} &::= \mathtt{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{T}\texttt{>}\ [\,T\,] \end{aligned}$$

- Constraint entailment:

IMPL
$$\frac{\mathtt{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{T}\texttt{>}\ [\,U\,]}{\Delta \Vdash [\overline{V/X}]\,\big(U\ \mathtt{implements}\ I\texttt{<}\overline{T}\texttt{>}\big)}$$

ENV
$$\frac{P \in \Delta}{\Delta \Vdash P}$$

- Subtyping:

REFL
$$\Delta \vdash T \leq T$$

TRANS
$$\frac{\Delta \vdash T \leq U \qquad \Delta \vdash U \leq V}{\Delta \vdash T \leq V}$$

OPEN
$$\frac{\Delta, \overline{P} \vdash X \leq T \qquad X \notin \mathrm{ftv}(\Delta, T)}{\Delta \vdash \exists\, X\ \mathtt{where}\ \overline{P}\ .\ X \leq T}$$

ABSTRACT
$$\frac{(\forall i)\ \Delta \Vdash [T/X]P_i}{\Delta \vdash T \leq \exists\, X\ \mathtt{where}\ \overline{P}\ .\ X}$$

Undecidability of Subtyping in $\mathcal{EX}_{impl}$

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

Summary

- 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