Second-order unification and functional arity

Aleksy Schubert

Faculty of Mathematics, Informatics and Mechanics, University of Warsaw

2nd of May, 2023
Programmers do not want to type a lot.

In STLC:

\[ X \text{ M} = X \text{ N} \rightarrow X \text{ MN} \]

where \( X \text{ M} \), \( X \text{ N} \), and \( X \text{ MN} \) are unification variables.
Programmers do not want to type a lot. Therefore need ways to automatically infer some coding information.
Programmers do not want to type a lot.
Therefore need ways to automatically infer some coding information.
In particular types are mostly optional.
The context of the problem

- Programmers do not want to type a lot.
- Therefore need ways to automatically infer some coding information.
- In particular types are mostly optional.
- Application of a function \( M \) to an argument \( N \)

\[ M \rightarrow N \]

introduces a unification constraint.
Programmers do not want to type a lot. Therefore need ways to automatically infer some coding information. In particular types are mostly optional.

Application of a function $M$ to an argument $N$

$M \ N$

introduces a unification constraint.

In STLC:

$$X_M \doteq X_N \rightarrow X_{MN}$$

where $X_M, X_N, X_{MN}$ are unification variables.
In polymorphic systems

- Types are more complicated

\[ A, B ::= C \mid X \mid A \to B \mid \forall X.A \]
In polymorphic systems

- Types are more complicated

\[ A, B ::= C | X | A \rightarrow B | \forall X.A \]

- Applications of a function \( M \) to an argument \( N \) is more complicated

\[ M\, A_1 \ldots A_n\, N \]
In polymorphic systems

- Types are more complicated

\[ A, B ::= C \mid X \mid A \rightarrow B \mid \forall X.A \]

- Applications of a function \( M \) to an argument \( N \) is more complicated

\[ M A_1 \ldots A_n N \]

- This introduces a unification constraint

\[ F_M A_1 \ldots A_n \vdash X_N \rightarrow X_M A_1 \ldots A_n N \]

where \( F_M \) is a second-order unification variable, \( X_N, X_M A_1 \ldots A_n N \) are unification variables.
If we are (un)lucky...

- Constraints fall within the second-order unification language.
If we are (un)lucky...

- Constraints fall within the second-order unification language.
- If we are (un)lucky – second-order abstract syntax is necessary.
What kind of programming language?

\[ M, N ::= x \mid \lambda x. M \mid \Lambda X. M \mid MA \mid MN \]

Type application critical for unification.
Type application omitted \(\Rightarrow\) seminunification.
Motivation for types:

- Expression of intent with regard to the program.
- Gentler writing of the code.
- Too big types are ineffective.

Example (J.B.Wells):

\[
\begin{align*}
\forall \gamma. (\gamma \rightarrow \gamma) \rightarrow \beta, \\
\forall \mu_1. (\mu_1 \rightarrow \delta_1) \rightarrow (\delta_2 \rightarrow \mu_2) \rightarrow (\tau_2 \rightarrow \tau_2),
\end{align*}
\]

\[\vdash b (\lambda x. cxx)\]

where \(\tau_1 \leq \mu_1\), \(\tau_2 \leq \mu_2\) is an instance of the seminunification problem.
Additional restrictions

Motivation for types:

- Expression of intent with regard to the program.
Motivation for types:
- Expression of intent with regard to the program.
- Gentler writing of the code.
Motivation for types:
- Expression of intent with regard to the program.
- Gentler writing of the code.

Too big types are ineffective.

Example (J.B.Wells):
\[
b : \forall \gamma. (\gamma \rightarrow \gamma) \rightarrow \beta,
\]
\[
c : \forall. (\mu_1 \rightarrow \delta_1) \rightarrow (\delta_2 \rightarrow \mu_2) \rightarrow (\tau_2 \rightarrow \tau_2),
\]
\[\vdash b (\lambda x. cxx)\]
where \(\tau_1 \leq \mu_1\), \(\tau_2 \leq \mu_2\) is an instance of the
seminunification problem.
Additional restrictions

- Motivation for types:
  - Expression of intent with regard to the program.
  - Gentler writing of the code.
- Too big types are ineffective.
- Example (J.B. Wells):

\[
\begin{align*}
  b & : \forall \gamma. (\gamma \rightarrow \gamma) \rightarrow \beta, \\
  c & : \forall. (\mu_1 \rightarrow \delta_1) \rightarrow (\delta_2 \rightarrow \mu_2) \rightarrow (\tau_2 \rightarrow \tau_2), \\
  \vdash b(\lambda x. cxx)
\end{align*}
\]

where \( \tau_1 \leq \mu_1, \quad \tau_2 \leq \mu_2 \) is an instance of the seminunification problem.
Additional restrictions

- All types in inference can be at most of size $n$. 

[Aleksy Schubert]
Additional restrictions

- All types in inference can be at most of size $n$.
- Quantified variables restricted to occur only up to certain depth (Giannini, Ronchi Della Rocca).
All types in inference can be at most of size $n$.
Quantified variables restricted to occur only up to certain depth (Giannini, Ronchi Della Rocca).
What if we bound the arity or functional rank?
Additional restrictions

- All types in inference can be at most of size $n$.
- Quantified variables restricted to occur only up to certain depth (Giannini, Ronchi Della Rocca).

What if we bound the arity or functional rank?

- $\text{arity}(c) = 0$ for a constant $c$ and 
  $\text{arity}(A_1 \to \cdots \to A_n \to c) = \max(\text{arity}(A_1), \ldots, \text{arity}(A_n), n)$,
Additional restrictions

- All types in inference can be at most of size $n$.
- Quantified variables restricted to occur only up to certain depth (Giannini, Ronchi Della Rocca).

What if we bound the arity or functional rank?

- $\text{arity}(c) = 0$ for a constant $c$ and
  $\text{arity}(A_1 \to \cdots \to A_n \to c) = \max(\text{arity}(A_1), \ldots, \text{arity}(A_n), n)$,
- $\text{rank}(c) = 0$ for a constant $c$, and
  $\text{rank}(A \to B) = \max(\text{rank}(A) + 1, \text{rank}(B))$. 
Type-checking and type-inference in domain-free languages is undecidable when arity or rank are restricted.
Type-checking and type-inference in domain-free languages is undecidable when arity or rank are restricted.
Example

Consider SOU instance

\[ A_n \rightarrow Fbc \equiv F(b \rightarrow a)(b \rightarrow c) \]
(Misleading) Machine simulation

\[ C_0 \rightarrow FB_1 \ldots B_k o \equiv FA_1 \ldots A_p (C_{n-1} \rightarrow o) \]
Verification of machine consistency

\[(Fs_1 \ldots sp \ o') \to GD'_1 \ldots D'_l \equiv GD_1 \ldots D_l\]

\[s \to \langle b, o \rangle \rightsquigarrow \langle s, 1 \rangle \text{ in case } \pi_3(s) = b,\]
\[s \to o \rightsquigarrow \langle s, 1 \rangle \text{ in case } \pi_3(s) = \bullet, \text{ or } \pi_2(s) = \bullet,\]
\[s \to \langle s', 1 \rangle \rightsquigarrow \langle s, 2 \rangle \text{ in case } \pi_3(s) = \pi_2(s') \text{ and } \pi_1(s') = \pi_2(s),\]
\[s \to \langle s', 2 \rangle \rightsquigarrow \langle s, 3 \rangle \text{ in case } \pi_3(s) = \pi_2(s') \text{ and } \pi_1(s') = \pi_2(s),\]
\[a \to \langle s, 3 \rangle \rightsquigarrow \langle a, 4 \rangle \text{ in case } \pi_1(s) = a,\]
\[a \to \langle b, 4 \rangle \rightsquigarrow \langle a, 4 \rangle.\]
Conclusions

- Restriction to bounded types leads to decidable type-checking, type reconstruction.
- Restriction to types with bounded arity/rank may lead to undecidable type-checking, type reconstruction.
The End

Aleksy Schubert