Explicit vs. Implicit Polymorphism in OML

Thomas Christensen
MSc. (CS) (Soon)
University of Aarhus, Denmark

Agenda

- Explicit vs. Implicit polymorphism in OML
- Type inference
- Problems
- Generics in OML
- Conclusion
- Questions

Explicit Polymorphism

```
class PolyFunctionTest1

functions
    Identity[@param] : @param -> @param
    Identity (p) == p;


    doTest : () -> int
    doTest() ==
            let a = Identity[int]   -- Explicit function instantiation
            in
                    a(42);

end PolyFunctionTest1
```

Implicit polymorphism

```
doTest : () -> int
doTest() == Identity(42);

doTest : () -> bool
doTest() == Identity(false);

doTest : () -> char
doTest() == Identity(“a”);

…
```

Type Inference

- Omitted type annotations need to be *reconstructed* by the type-checker
- Classical ML-style type inference uses the Hindley-Milner algorithm.
  - 1. Assign type variables to all expressions
  - 2. Generate type constraints using the AST
  - 3. Solve constraints by unification

Type Inference

- Recent (2 days ago) addition to OML
- Untyped Explicit Functions

```
functions
  Foo : int * int * int -> int
  Foo (x, y, z) == x + y + z
```

- Fully implemented (Disclaimer: on the syntactic level only)

Type Inference

- Recent (2 days ago) addition to OML
- Untyped Explicit Functions

```
functions
  Foo : int * int * int -> int
  Foo (x, y, z) == x + y + z
```

- Fully implemented (Disclaimer: on the syntactic level only)

Problems...

- Union types
  - The actual type of the element in the union type cannot be determined statically.
  - In the presence of union types the algorithm may infer too general a type to be actually useful
- Invariants
  - User-defined types may have arbitrarily complex invariants imposed on them. Respecting the invariants would require evaluating them at compile time.

Example 1

```
f[@p] : seq of @p -> @p
f(x) == if len x = 1
        then x(1) + 1
        else x(2) or false;
-------------------------------
let a = [true, 87]
in f(a)
-------------------------------
```

Example 1

```
f[@p] : seq of @p -> @p
f(x) == if len x = 1
          then x(1) + 1
          else x(2) or false;
--------------------------------
let a = [true, 87]
in f(a)
--------------------------------
```

- Constraints generated from inference rules
- Example – sequence length operator

$$\frac{\vdash x : \text{seq of } A}{\vdash \text{len}(x) : \text{nat}}\ \text{SeqLen}$$

- Generates constraints
  - `[x] = seq of A`
  - `[len x] = nat`

Example 1

```
f[@p] : seq of @p -> @p
f(x) == if len x = 1
          then x(1) + 1
          else x(2) or false;
--------------------------------
let a = [true, 87]
in f(a)
--------------------------------
```

- Generate constraints…
- Syntax: [x] type of expression x
- [α] = (nat | bool) = [@p]
- [a] = seq of α = seq of [@p]
- [x] = seq of α = seq of [@p]
- [x] = seq of nat
- [x] = seq of bool

Example 1

```
f[@p] : seq of @p -> @p
f(x) == if len x = 1
          then x(1) + 1
          else x(2) or false;
--------------------------------
let a = [true, 87]
in f(a)
--------------------------------
```

- Solving by unification gives us
  - `[a] = seq of (bool | nat)`
  - `[@p] = bool | nat`
  - `[α] = bool | nat`
  - `[x] = seq of nat`
  - `[x] = seq of bool`

Example 1

```
f[@p] : seq of @p -> @p
f(x) == if len x = 1
          then x(1) + 1
          else x(2) or false;
--------------------------------
let a = [true, 87]
in f(a)
--------------------------------
```

- `[x] = seq of nat`
- `[x] = seq of bool`

Incompatible types !

- Is this a problem ?

Example 1

```
f[@p] : seq of @p -> @p
f(x) == if len x = 1
          then x(1) + 1
          else x(2) or false;
--------------------------------
let a = [true, 87]
in f(a)
--------------------------------
```

- [x] = seq of nat
- [x] = seq of bool

Incompatible types !

- Is this a problem ?
  - Not necessarily, if our specification includes a type declaration that matches (bool | nat)

```
types
   natbool = bool | nat
```

Example 1

```
f[@p] : seq of @p -> @p
f(x) == if len x = 1
          then x(1) + 1
          else x(2) or false;
--------------------------------
let a = [true, 87]
in f(a)
--------------------------------
```

```
types
   natbool = bool | nat
```

- [x] = seq of natbool
- [x] = seq of natbool

- Incompatibility goes away
- This is not unsound, as we infer an existing type.

Example 1

```
f[@p] : seq of @p -> @p
f(x) == if len x = 1
          then x(1) + 1
          else x(2) or false;
--------------------------------
let a = [true, 87]
in f(a)
--------------------------------
```

- However, is it ”ethical” ?
  - We may have defined a type to be used in a specific modelling context.
  - Using this type in another, possibly unrelated context may make our intentions unclear.
  - Access modifiers (`private, public, protected`) can control whether the inference algorithm may use the predefined type in the new context.

Example 1

```
f[@p] : seq of @p -> @p
f(x) == if len x = 1
          then x(1) + 1
          else x(2) or false;
--------------------------------
let a = [true, 87]
in f(a)
--------------------------------
```

- How about if there is no previously defined *natbool* type ?
  - The inference algorithm can create one.
  - **Result:** All specifications are guaranteed to be statically type correct since we can create suitable union types on the fly
    - Do we *really* want this ? (No)
  - Report a type error instead.

Example 2 - Non-disjoint union types

```
class PolyFunctionTest6

types
   natreal = nat | real

functions
   f[@p] : seq of @p -> @p
   f(x) == if len x = 1
           then x(1) mod 2     -- mod :  int  * int  -> int
           else x(2) + 76;     --   + :  real * real -> real

   doTest : () -> int
   doTest() ==
         let a = [42.1 , 87 ]
         in f[natreal](a);

end PolyFunctionTest6
```

Example 2 - Non-disjoint union types

```
class PolyFunctionTest6

types
   natreal = nat | real

functions
   f[@p] : seq of @p -> @p
   f(x) == if len x = 1
           then x(1) mod 2     -- mod :  int  * int  -> int
           else x(2) + 76;     --   + :  real * real -> real

   doTest : () -> int
   doTest() ==
         let a = [42.1 , 87 ]
         in f[natreal](a);

end PolyFunctionTest6
```

```
[x] = seq of nat | real
```

Should we merge disjoint types to the most general type ?

[x] = seq of real

To be completely safe...

- Allow only operations on the union type which are valid for ***all*** member types in the union.
- (This restriction is ignored in the **C** language, leading to ”implementation-dependant ” results [K&R] and potential safety violations)

Proposal for Generics in OML

- Adding generics to OML
- Similar to generics in Java 1.5

```
Linkedlist<String> = new LinkedList<String>()
```

- Class declarations parameterized with a list of type variables

Proposal for Generics in OML

- Syntax:

```
class A <[@S, @T]> is subclass of B

types
    items : seq of @S

instance variables
    currentItem : A<[@T, int]> := new A<[@T, int]>()

end A
```

- Currently at the pre-experimental stage.
- Thanks to Marcel for hacking the grammar into place…

Conclusion

```
types
   natbool = nat | bool

functions
   f[@p] : seq of @p -> @p
   f(x) == if len x = 1
           then x(1) + 1
           else x(2) or false;

   doTest : () -> int
   doTest() ==
         let a = [true, 87]
         in f[natbool](a);
```

- This example typechecks OK in VDMTools POS mode (8 errors in DEF mode), but fails at runtime as it attempts to evaluate: `87 or false`.

Conclusion

- Thus, type inference cannot bring us from POS to DEF correct.
- Proof obligations and dynamic checks are still necessary.

- It **can** provide:
  - Fewer type annotations while retaining the same level of static type correctness.
  - Specifically it can give us implicitly typed polymorphic functions.

Conclusion

- Generics may provide us with additional flexibility in specifying classes.
- Additionally it may move certain type-checks from run-time to compile time.
- The latest version of the OML grammar (created Sunday afternoon) supports the generics syntax. Further work is needed to fully implement this.

end Presentation

- Questions
- Comments
- Objections
- Discussion
- Suggestions