Liberal Typing for Functional Logic Programming

This page contains a web interface to experiment with the type system presented in [1]. The syntax to be used for programs is that of the TOY system.

Error: Macro TOC(None) failed

```
'NoneType' object has no attribute 'endswith'
```

Syntax and Usage

The web interface is composed by two text areas ("Input Program" and "Types / Error Messages") and a button "Typechecking!". The usage is:

1. Write a program in the "Input Program" text area.
2. Press the button "Typechecking!".
3. The system will typecheck the input program and it will write in the "Types / Error Messages" text area the types of all the data and function symbols. Otherwise, it will write in that text area the different errors detected (lexic, syntactic or type errors).

The programs supported in the "Input Program" text area can have data declarations, function type declarations and function rules. In general, the syntax is (almost) similar to current functional and functional logic languages. The main difference is that (as usual in TOY) data and type variables are uppercased and data/type constructors and function symbols are lowercased. Here we show the different categories by examples.

Data declarations

Data declarations define new data types: their constructors and their types. We support the classical data type declarations:

```
data bool = true | false
data nat = z | s nat
data list A = nil | cons A (list A)
```

We also support GADT-like data type declarations, where the type of each constructor is explicitly declared:

```
   data repr A where
      rbool :: repr bool
      rnat  :: repr nat
      rlist :: repr A -> repr (list A)
```

Function type declarations

Function type declarations have following shape:

```
and :: bool -> bool -> bool
id  :: A -> A
eq' :: repr A -> A -> A -> bool
```

Function rules

Function rules follow the definition presented in [1]: `FunctionName Pattern1 ... PatternN = Expression`. A pattern is a constructor symbol partially or totally applied to patterns, or a function symbol partially applied to patterns. An expression is a variable, a symbol, an application or a let binding a variable (`let X = E1 in E2`).

```
and true  X = X
and false X = false

id X = X

f = (let F = id in F true)

eq' rbool X Y = eq'Bool X Y
eq' rnat  X Y = eq'Nat X Y
```

Web interface

Examples

You can copy the code of the examples and paste it in the *Input program* area of the Web interface.

- Size.toy: size as a type-indexed function. This is the pre-loaded example that you find in the Web interface when you start the session.
- Examples.toy: several examples showing well and ill-typed rules and expressions.
- EqualityTypeIndexed.toy: equality as a type-indexed function.
- EqualityGADT.toy: equality as a type-indexed function using GADTs
- Embedded.toy: small embedded language statically typed. Extracted from [2]
- Opacity.toy: some examples showing the support for existential types in constructors and the opacity caused by HO patterns.
- HOFOapply.toy: the apply function appearing in the HO-to-FO translation used in standard FLP implementations is well typed.
- GenericSize.toy: size as a generic function using a universal datatype.

Usual error messages

The error messages follow the same pattern: `ERROR(`**number**`)`**: row : column : message**.

- **Symbol not defined**: When the program uses a symbol not defined as constructor or function. Example:

```
main :: A
main = loop
```

```
ERROR(701):2:8: Symbol loop not defined.
```

- **Could not match expected type**: When the type of an application does not match the expected one. Example:

```
data bool = true | false
data nat = z | s nat

main :: nat
main = s true
```

```
ERROR(702):5:8: Could not match expected type 'nat -> nat' against inferred type 'bool -> A' in the expressi
```

- **The type inferred for the right-hand side is not more general that the one inferred for the left-hand side**: This message is shown when a rule does not hold the third point of the definition of well-typed rule in [1]. Example:

```
snd :: A -> B -> B
snd X Y = Y

co :: (A -> A) -> B
co (snd X) = X
```

```
:1: The types <A, X::A> inferred for the right-hand side do not match <C, X::B> inferred for the left-hand side
```

- **Missing type declaration**: When a function symbol does not have an explicit type declaration. Example:

```
data bool = true | false

main = true
```

```
ERROR(708):3:1: The function main does not have an explicit type declaration.
```

Discussion: the order of the arguments

The simplicity of our type system leads also to simpler behavior when compared to the implementation of GADTs in functional systems like GHC, where well-typedness involving GADTs is subtly sensible to the order of arguments in a function. Consider for instance the program EqualityGADT.toy, which is the equality function as a type-indexed function using GADTs. Adapting the syntax, it is also a legal Haskell program in GHC (eq_GADT.hs). However, for many users it could be an unpleasant surprise that switching the first and third arguments of eq leads to a type error in GHC. The program eq_GADT_reversed.hs is the same as eq_GADT.hs but changing the first and the third arguments of the eq. The following GHCi session shows how this program is rejected in GHC 6.12.3 (12 June 2010) due to a type error in a rule of the eq funcion:

```
GHCi, version 6.12.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Prelude> :l eq_GADT
eq_GADT_reversed.hs  eq_GADT.hs
Prelude> :l eq_GADT_reversed.hs
[1 of 1] Compiling Main             ( eq_GADT_reversed.hs, interpreted )

eq_GADT_reversed.hs:21:3:
    Couldn't match expected type `Bool' against inferred type `Nat'
    In the pattern: Z
    In the definition of `eq': eq Z Z RNat = True
Failed, modules loaded: none.
```

However, our system accept the eq function where the first and the third arguments are swapped (EqualityGADTreversed.toy).

References

1. ^ Francisco J. López-Fraguas and Enrique Martin-Martin and Juan Rodríguez-Hortalá, More Liberal Typing for Functional Logic Programming, 8th Asian Symposium on Programming Languages and Systems (APLAS 2010), 2010, pages 80-96
2. ^ Ralf Hinze, Fun with phantom types, The Fun of Programming, 2003, pages 245-262

Contact

This sofware is developed by Francisco Javier López Fraguas, Enrique Martín Martín and Juan Rodríguez Hortalá, at the Department of Computer Systems and Computing, Universidad Complutense de Madrid, Spain.