HANUS: EMBEDDING JANUS IN HASKELL

Joris ten Tusscher, Joris Burgers, Ivo Gabe de Wolff, Cas van der Rest, Orestis Melkonian





INTRODUCTION

- ▶ Janus is an imperative reversible programming language, meaning that every computation and function can be reversed.
- ► Hanus is an extended implementation of Janus that can be compiled straight to Haskell. Because of this, Hanus contains many awesome Haskell features that are unthinkable in regular Janus!

REVERSE YOUR PROGRAM

► The inverse of a program can be computed automatically.

DIVISION

```
1 [hanus|
2 procedure divide(x :: Int, y :: Int, z :: Int){
3    from x >= y && z == 0 loop
4    z += 1;
5    x -= y;
6   until x < y;
7 } ]</pre>
```

REVERSE OF DIVISION

```
1 [hanus |
2 procedure divide'(x :: Int, y :: Int, z :: Int) {
3     from x < y loop
4          x += y;
5          z -= 1;
6     until x >= y && z == 0;
7 } ]
```

► Procedures are called with either the *call* or the *uncall* keyword. The *main* procedure is called automatically.

EXECUTION

SYNTACTIC CHECKING

► By using *QuasiQuotation*, the programmer gets notified of syntactic errors at compile-time!

CODE

```
[hanus|procedure main() {
   local n : Int = 10;
        n += 10;
   delocal 20;
} ]
```

ERROR

```
Exception when trying to run compile-time code:
   Parsing of Janus code failed in file ....
   First error:
-- Expecting "::" at position LineCol 2 10
```

SEMANTIC CHECKING (JANUS SIDE)

► *Hanus* also reports semantic errors, such as violating Janus-specific constraints for expressions.

CODE

```
count :: BinaryTree a -> Int

[hanus]

a :: BinaryTree Int;

procedure main() {

createNode a;

a.value += (count a.left) + a.value

}

ERROR

Semantic Error (line 6, col 23):
Assigned variable appears on the left-hand side.
```

SEMANTIC CHECKING (HASKELL SIDE)

➤ Since regular Haskell programs are generated, users also get error messages for *anti-quoted* Haskell expressions.

CODE

HASKELL POWER

- ► The programmer can add additional operators by defining functions for forward and backward execution.
- ► We can define an operator that works on all Functors:

DEFINITION

```
1 (=$$) :: Functor f => Operator (f a) (Operator a b, b)
2 (=$$) = Operator forward backward
3    where
4    forward f (Operator fwd _, x) = fmap ('fwd' x) f
5    backward f (Operator _ bwd, x) = fmap ('bwd' x) f

USAGE

1 procedure increase(tree :: BinaryTree Int) {
2    tree =$$ (+=, 42);
3 }
```

► Besides operators, the programmer can also define field and array indexers which allow you to use tree.leftChild and array[x] on the left hand side.

