Janus

Marek Kaput, Radomir Krawczykiewicz

https://github.com/mkaput/janus


JANUSZE
HASKELLA
memegenerator.net

Co to Janus?

- Język programowania
- Interpretowany (interpreter w Haskellu)
- Dynamicznie typowany, ale tylko jawne rzutowanie; wartościowanie zachłanne
- Składnia silnie wzorowana na języku Rust
  - z drobnymi zmianami jak `True`/`False` z Pythona, czy `:=` z Pascala/Go
- Zmienne, instrukcje warunkowe, pętle (z `break`/`continue`), funkcje, referencje (prawie), funkcje wyższego rzędu (technicznie)

Czego nie ma?

- Nie jest (praktycznie) kompletny w sensie Turinga
  - nie ma tablic (ale są stringi)
- Lambd i domknięć
  - ale konstrukcja interpretera pozwala na dość prostą implementację
- Typów złożonych (struktur, klas itp.)
- Modułów
  - cały program musi być w jednym pliku
- Przestrzeni nazw
  - ale są scope'y, więc nie ma spiny
- Używalnej biblioteki standardowej

Typy

Unit	`()`
Bool	`True`, `False`
Int	Bignumy
Double	IEEE double-precision – 64 bity
Char	UTF-8
Str	UTF-8, linked-list
Item	`Func`, `NativeFunc`

Basic Janus

```
/* Comment */
// Comment

let a = 5;
a := square(a);
if a == 25 and True or False {
    println("It works!");
} else {
    while a > 0 { a--; }
    loop { println("It doesn't work :("); break; }
}

fn square(x) { x * x; }

let string = "abrakadabra";
string[2] := 'x';
// string == "abxakadabra"
```

Precedence	Operator	Associativity	Operation
20	`(...)`	n/a	Grouping
19	-	-	-
18	`... ( ... )`	left-to-right	Function call
17	`... ++`	n/a	Postfix increment
	`... --`	n/a	Postfix decrement
16	`! ...`	right-to-left	Logical NOT
	`~ ...`	right-to-left	Bitwise NOT
	`+ ...`	right-to-left	Unary plus
	`- ...`	right-to-left	Unary minus
	`++ ...`	n/a	Prefix increment
	`-- ...`	n/a	Prefix decrement
15	`... ** ...`	right-to-left	Exponentation
14	`... * ...`	left-to-right	Multiplication
	`... / ...`	left-to-right	Division
	`... mod ...`	left-to-right	Remainder
13	`... + ...`	left-to-right	Addition
	`... - ...`	left-to-right	Substraction
12	`... << ...`	left-to-right	Bitwise left shift
	`... >> ...`	left-to-right	Bitwise right shift
11	`... & ...`	left-to-right	Bitwise AND
10	`... ^ ...`	left-to-right	Bitwise XOR
9	`... | ...`	left-to-right	Bitwise OR
8	`... == ...`	left-to-right	Equality
	`... != ...`	left-to-right	Inequality
	`... < ...`	left-to-right	Less than
	`... > ...`	left-to-right	Greater than or equal
	`... <= ...`	left-to-right	Less than or equal
	`... >= ...`	left-to-right	Greater than or equal
7	-	-	-
6	`... and ...`	left-to-right	Logical AND
5	`... or ...`	left-to-right	Logical OR

Intermediate Janus

```
// Everything (almost) is expression
let a = if 2 + 2 == 4 {
    "Math works";
} else {
    42.0;
};

// Including loops
let b = loop { break; };

let c = if False { 123; };

a == "Math works";
b == ();
c == ();

// Block scopes, Let scopes & References
{
    a == "Math works";
    let a = 42;
    a == 42;
    let b = a;
    a := 7;
    a == 7 and b == 7;
}

a == "Math works";
```

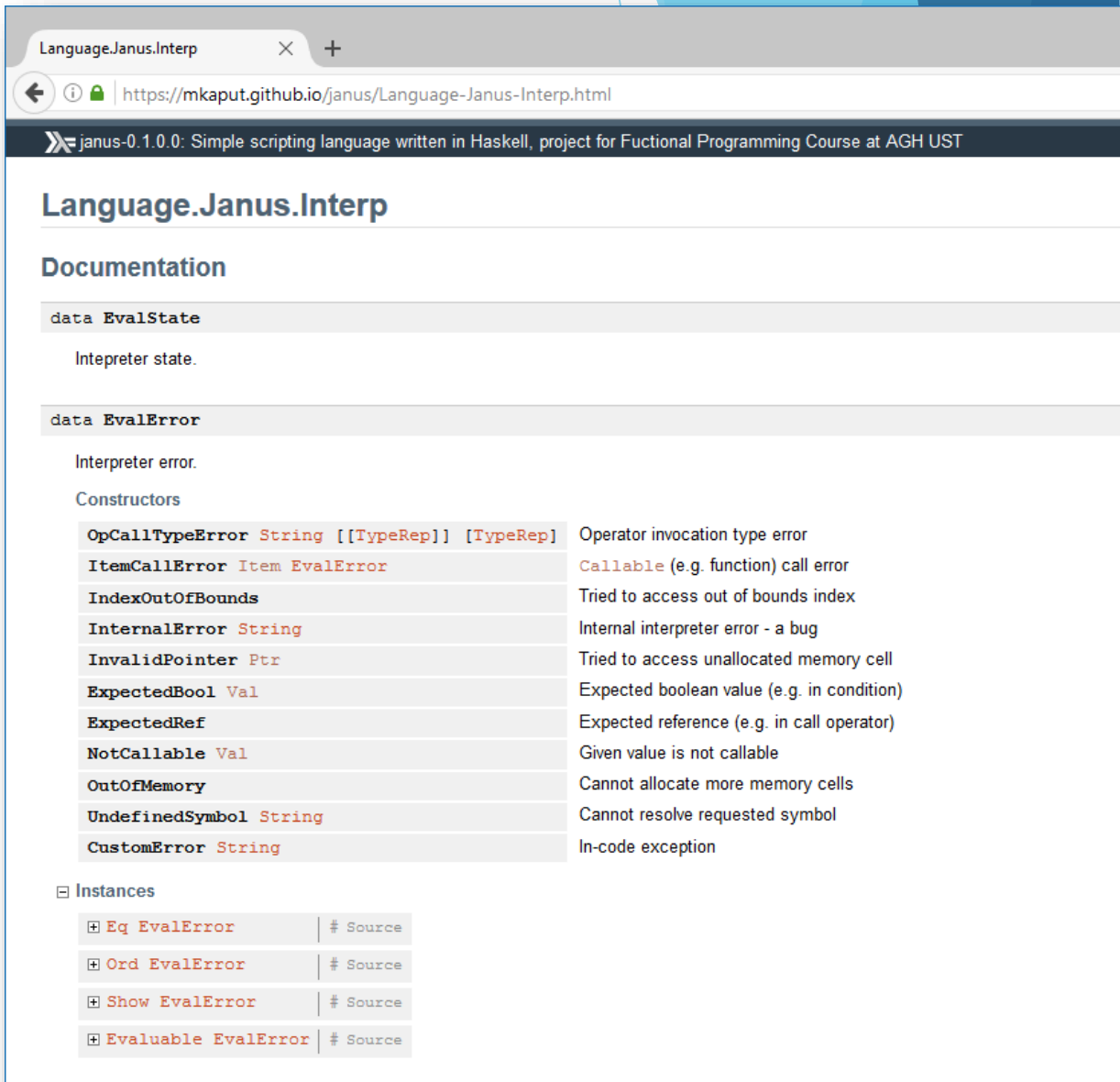
Advanced Janus

```
// No implicit casts
2 + 2.0; // fails
'B' - 1; // fails
1 == 1.0; // fails
"ab" + "bc";
'a' + "bc";

// Higher-order functions
fn call_me(f, a) {
    f(a);
}

// Print with multiple arguments - just like Python
println("Call me: ", call_me(square, 10));

// Input
let a = getline(); // a: Str
let i = int(a);    // i: Int if succeeded
```

Implementacja

Ogółem

- ~2 kLOC
- 96 testów
- Interpreter napisany jako biblioteka:
  - Language.Janus.AST
  - Language.Janus.Interp
  - Language.Janus.Parser
  - Language.Janus.Stdlib
- Publiczne API, w większości udokumentowane
- Monad Transformers, Parsec, Containers, Mutable Hashtables
- Hspec + QuickCheck



InterpM

- `type InterpM = StateT EvalState (ExceptT EvalError IO)`
- Serce interpretera:
  - ```
    class Evaluable a where
      eval :: a -> InterpM Val
    ```
  - ```
    class RefEvaluable a where
      evalRef :: a -> InterpM Ref
      tryEvalRef :: a -> InterpM (Maybe Ref)
      {-# MINIMAL evalRef | tryEvalRef #-}
    ```
  - ```
    class Callable a where
      call :: a -> [Val] -> InterpM Val
    ```
- Dostęp poprzez:
  - `runInterpM :: InterpM a -> IO (Either EvalError a)`
  - `run :: Evaluable a => a -> IO (Either EvalError Val)`
    ```

Ewaluacja

- Ewaluujemy węzły AST; brak IR – reprezentacji pośredniczących
- Większość węzłów AST intepretowana za pomocą mechanizmu *callOpN* + *wrapOpN*
  - *wrapOpN* przyjmuje *N*-argumentową dowolną funkcję Haskella i opakowuje ją w funkcję typu `Val -> Val -> … -> Maybe Val` oraz informacje o typach
  - *callOpN* próbuje wykonać zestaw takich funkcji dla zadanych argumentów. Jeżeli typy nie zgadzają się dla pierwszej funkcji, próbuje następną
  - ```
    eval (ExpExpr a' b') = callOp2 "x ** n" [
      wrapOp2 ((^) :: Integer -> Integer -> Integer),
      wrapOp2 ((**) :: Double -> Double -> Double)
    ] a' b'
    ```
- `RefEvaluable` służy do wyliczania referencji z danego wyrażenia – *Lvalues*
  - Używane w konstrukcji podstawiania: `a[5] := 'c';`
    ```

Stan

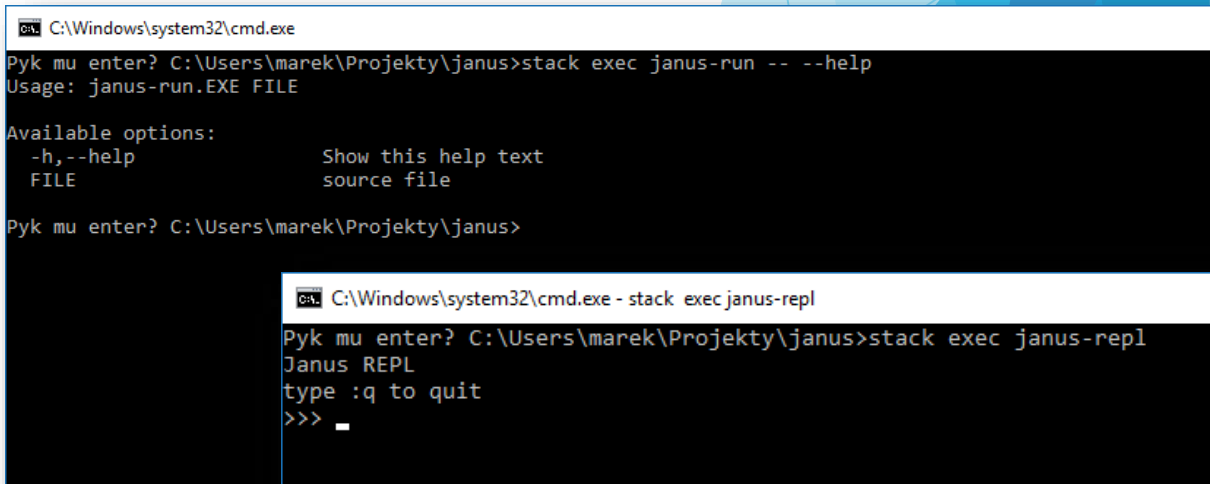
- ```
  data EvalState = EvalState {
       nextMptr :: Ptr,
       mem      :: MHashTable Ptr MemCell,
       stack    :: [StackFrame]
   }
  ```
- Pamięć jako **mutowalna** tablica haszowana – prosta implementacja. `MemCell` zawiera wartość i `refcount` – Garbadge Collector.
- ```
  newtype Ptr = Ptr Word
              deriving (Eq, Ord, Show, Data, Typeable)
  ```
- Ścisła ewaluacja
- Problem: nie używamy ponownie zwolnionych adresów

Stan – stos

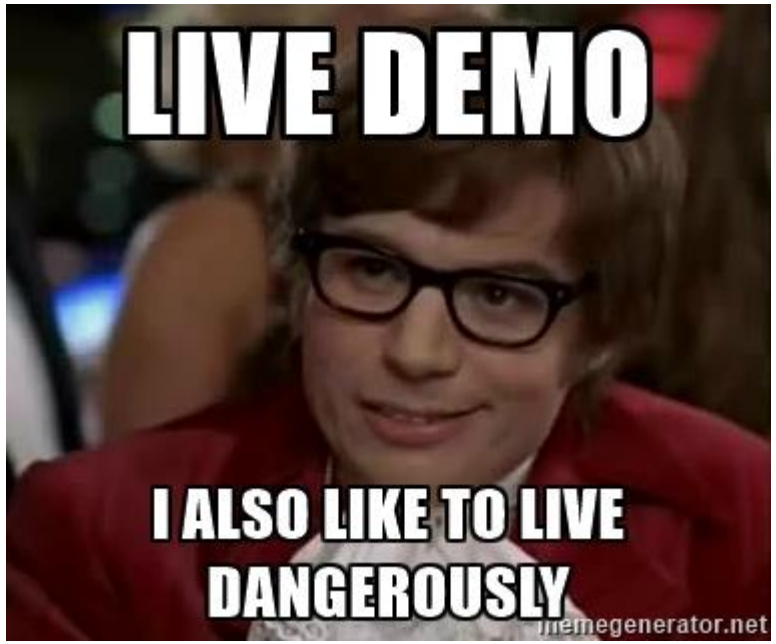
- Idea podobna do odpowiednika z Assemblera
- 4 rodzaje ramek stosu: ScopeFrame, CallFrame, BlockFrame, LoopFrame
- `ScopeFrame { symbols :: MHashTable String Ptr }`
  - *scope* to tylko zbiór nazwanych wskaźników
- Pozostałe rodzaje służą jako markery

Inne aspekty

- Wyjątki obsługiwane za pomocą `ExpectT`
  - Raz rzucony wyjątek przerywa całą monadę InterpM
  - Wada: Mocno komplikuje kwestię zwijania stosu
  - Za pomocą wyjątków obsługujemy skoki: `break, continue, return`
- Cykle pomiędzy modułami *InterpM* i *AST* – plik *hs-boot*
- Tooling: Runner i REPL



Demo


LIVE DEMO
I ALSO LIKE TO LIVE DANGEROUSLY
memegenerator.net