Concepts of programming languages

Janus

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

Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Janus

A reversible programming language.

Not turing complete!

Reversibility

Every statement can be reverted. No history is stored.

```
x += y * 3
```

Reversibility

Every statement can be reverted. No history is stored.

```
x += y * 3
```

```
x -= y * 3
```

Injective functions

Reversible languages can only compute injective functions.

$$\forall x, y : f(x) = f(y) \implies x = y \tag{1}$$

Every output has only a single input.

Injective functions

Reversible languages can only compute injective functions.

$$\forall x, y : f(x) = f(y) \implies x = y \tag{1}$$

Every output has only a single input.

$$h(x) = (x, g(x)) \tag{2}$$

Turing completeness

Turing machines can compute non-injective functions.

Reversible languages are not turing complete.

Reversible Turing complete.

Turing machines

Infinite tape of memory

Finite set of states

Transition function

- Current state
- Current symbol on tape
- Write symbol
- Move tape pointer
- Next state

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

Turing machines

Forward deterministic: given any state and tape, there is at most one transition *from* that state.

Backward deterministic: given any state and tape, there is at most one transition *to* that state.

P is the class of forward deterministic turing machines, NP of non-deterministic turing machines.

Reversible Turing complete: a language that can simulate forward and backward deterministic turing machines.

What do reversible languages compute

Given a forward deterministic turing machine that computes $f(x)$,

There exists a reversible turing machine that computes $x \to (x, f(x))$.

More memory.

Example

`fib`: calculates (n+1)-th and (n+2)-th Fibonacci number.

```
procedure fib
    if n = 0 then
        x1 += 1    ; -- 1st Fib nr is 1.
        x2 += 1    ; -- 2nd Fib nr is 1.
    else
        n -= 1
        call fib
        x1 += x2
        x1 <=> x2
    fi x1 = x2
```

Example

`fib`: calculates (n+1)-th and (n+2)-th Fibonacci number.

```
procedure fib
    if n = 0 then
        x1 += 1    ; -- 1st Fib nr is 1.
        x2 += 1    ; -- 2nd Fib nr is 1.
    else
        n -= 1
        call fib
        x1 += x2
        x1 <=> x2
    fi x1 = x2     ; -- Why do we need this?
```

Example

`fib`: calculates (n+1)-th and (n+2)-th Fibonacci number.

```
procedure fib
    if n = 0 then
        x1 += 1     ; -- 1st Fib nr is 1.
        x2 += 1     ; -- 2nd Fib nr is 1.
    else
        n -= 1
        call fib
        x1 += x2
        x1 <=> x2
    fi x1 = x2      ; -- Why do we need this?
```

- Q: How do we calculate the inverse?

Example

`fib`: calculates (n+1)-th and (n+2)-th Fibonacci number.

```
procedure fib
    if n = 0 then
        x1 += 1     ; -- 1st Fib nr is 1.
        x2 += 1     ; -- 2nd Fib nr is 1.
    else
        n -= 1
        call fib
        x1 += x2
        x1 <=> x2
    fi x1 = x2      ; -- Used for inverting the if-statement.
```

Q: How do we calculate the inverse?

$\mathcal{I}[\![\texttt{if } e_1 \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ fi } e_2]\!] \quad = \texttt{if } e_2 \texttt{ then } \mathcal{I}[\![s_1]\!] \texttt{ else } \mathcal{I}[\![s_2]\!] \texttt{ fi } e_1$

Example

`fib`: calculates (n+1)-th and (n+2)-th Fibonacci number.

```
procedure fibInverse
    if x1 = x2 then
        x2 -= 1          ; -- 2nd Fib nr is 1.
        x1 -= 1          ; -- 1st Fib nr is 1.
    else
        x1 <=> x2
        x1 -= x2
        call fibInverse
        n += 1
    fi n = 0
```

Example

`fib`: calculates (n+1)-th and (n+2)-th Fibonacci number.

- Q: What does the inverse of fib do?

```
procedure fibInverse
    if x1 = x2 then
        x2 -= 1             ; -- 2nd Fib nr is 1.
        x1 -= 1             ; -- 1st Fib nr is 1.
    else
        x1 <=> x2
        x1 -= x2
        call fibInverse
        n += 1
    fi n = 0
```

Relational Programming

Injective Programming	Relational Programming
r-Turing Complete	Turing Complete
backwards deterministic	backwards non-deterministic
restricted language constructs	search procedure (aka *resolution*)

Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Prolog basics

A logic program consists of *facts* and *rules*.

```
parent(alice, joe).
parent(bob, joe).
parent(joe, mary).
parent(gloria, mary).

ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).

descendant(X, Y) :- ancestor(Y, X).
```

The user can then *query* the runtime system, as such:

```
?- ancestor(alice, mary).
true.

?- parent(X, mary).
X = joe;
X = gloria.

?- ancestor(X, mary), not parent(X, mary).
X = alice;
X = bob.
```

Demonstration - Type Predicate

Assume a `type` predicate, relating expressions with types:

```
type(expr, t) :- ... .
```

Demonstration - Type Predicate

Assume a `type` predicate, relating expressions with types:

```
type(expr, t) :- ... .
```

You would normally use it to perform *type-checking*:

```
?- type(1 + 1, int).
true.
?- type(1 + 1, string).
false.
```

But you can also perform *type-inference*:

```
?- type(1 + 1, Type).
Type = int.
?- type("hello world", Type).
Type = string.
?- type(λx:int -> x, Type).
Type = int -> int.
?- type(λx -> x, int -> Type).
Type = int.
```

Going in the reverse direction, you can generate programs:

```
?- type(Expr, int).
Expr = 1;
Expr = 2;
...
Expr = 1 + 1;
Expr = 1 + 2;
...
Expr = if true then 1 else 1;
...
```

Of course, this does not make much sense without a sufficiently expressive type system.

Demonstration - Relational Interpreter

Assume you have implemented a *relational interpreter*:

```
eval(program, result) :- ... .

?- eval(map (+ 1) [1 2 3], Result).
Result = [2 3 4].
```


Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

Demonstration - Relational Interpreter

Assume you have implemented a *relational interpreter*:

```
eval(program, result) :- ... .
```

```
?- eval(map (+ 1) [1 2 3], Result).
Result = [2 3 4].
```

Non-deterministic constructs are also natural:

```
?- eval(amb [a, b, c], Result).
Result = a;
Result = b;
Result = c.
```

But you can also perform *program synthesis* by-example:

```
?- eval(F 1, 2),...,eval(map F [1 2 3], [2 3 4]).
...
F = λx -> x + 1;
...
F = λx -> x - 10 + 10 + 1;
...
```

But you can also perform *program synthesis* by-example:

```
?- eval(F 1, 2),...,eval(map F [1 2 3], [2 3 4]).
...
F = λx -> x + 1;
...
F = λx -> x - 10 + 10 + 1;
...
```

Quine generation is straightforward:

```
?- eval(Quine, Quine).
...
Quine = (λa -> a ++ show a) "(λa -> a ++ show a) ";
...
```

Logic Programming IRL

In practice, bi-directionality breaks with the usage of *extra-logical* features:

- **Variable projection**: inspecting values at runtime
- **Cut (!)**: disables backtracking in certain places
- **Assert/Retract**: dynamically insert/remove facts

Logic Programming IRL

In practice, bi-directionality breaks with the usage of *extra-logical* features:

- **Variable projection**: inspecting values at runtime
- **Cut (!)**: disables backtracking in certain places
- **Assert/Retract**: dynamically insert/remove facts

MiniKanren is a more recent logic programming language, which avoids extra-logical features (as much as possible).

Higher abstraction

- Relational programming, as well as functional programming, both belong to the *declarative* paradigm.
- They both focus on *what* a program does, rather than *how*.

Higher abstraction

- Relational programming, as well as functional programming, both belong to the *declarative* paradigm.
- They both focus on *what* a program does, rather than *how*.

Question

How can we combine them, to get the best of both worlds?

Hanus: Janus embedded in Haskell

In our research project, we use *TemplateHaskell* and *QuasiQuotation* to embed Janus in Haskell:

```
[hanus| procedure encode(im :: Image, ret :: [Byte]) {
    -- Janus commands with antiquotation
}|]
encode :: Image -> [Byte]
encode = call encode
decode :: [Byte] -> Image
decode = uncall encode
```

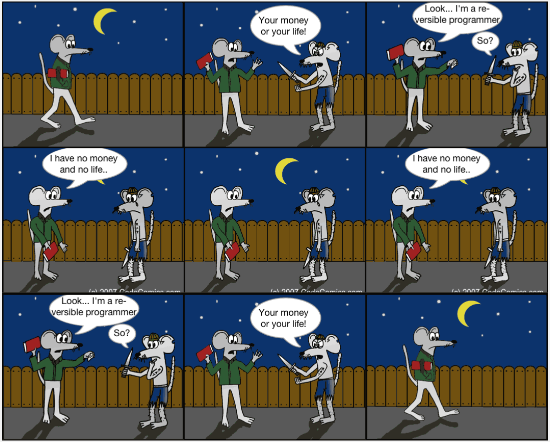
Come and check out our poster in de Vagant!

Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Thanks! Questions?

From CodeComics.com, modified.



Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

References

Axelsen, Holger Bock, and Robert Glück. “What do reversible programs compute?.” FoSSaCS. 2011. Yokoyama, Tetsuo, and Robert Glück. “A reversible programming language and its invertible self-interpreter.” Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation. ACM, 2007.