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]

Motivations

What are reasons to pursue logically reversible computations?

... * More heat efficient circuitry ... * Quantum computing

A small detour

Let's venture into the realm of physics ...

Imagine a set of balls bouncing around in a frictionless world:

Bouncing balls 1

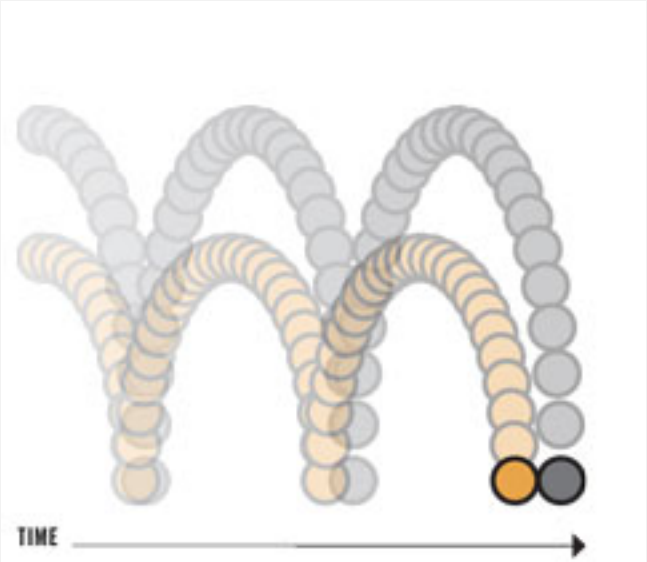

Figure 1: bouncing balls in a world without friction

...

All information about both future and past configurations is preserved.

Bouncing balls 2

bouncing balls in the real world

...

Information about past configurations gets lost as the balls lose velocity.

This information is not truely lost, however.

...

Entropy of the system must increase or remain equal. In this case, **heat** is dissipated into the environment.

Landauer's principle

In computers, information about past states is often lost (or erased) as computations are carried out.

...

However, the second law of thermodynamics still applies.

This means that circuits *must* dissipate some amount of heat as information gets destroyed.

...

Commonly refered to as **Landauer's principle**.

Reversible computing

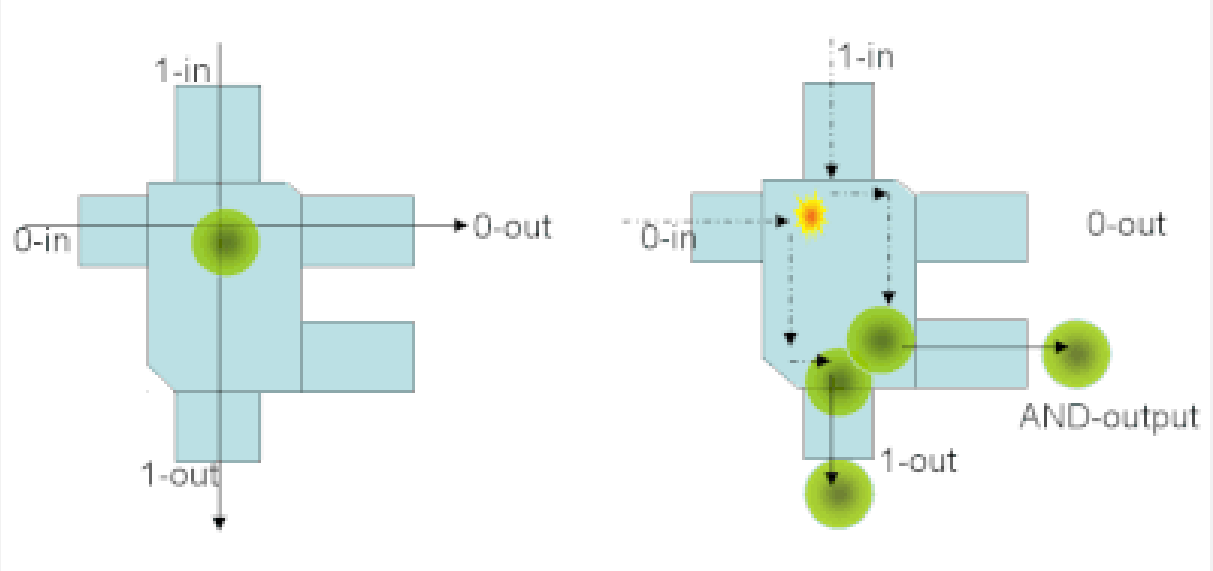

Figure 2: Billiard ball AND-gate

Relational Programming

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

Prolog basics

A logic programs 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:

```
?- parent(X, joe).
X = alice;
X = bob.

?- ancestor(X, mary).
X = joe;
X = gloria;
X = alice;
X = bob.

?- ancestor(X, mary), descendant(X, alice).
X = joe.
```

Demonstration - Type Inference

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 performing *type-inference*:

```
?- type(1 + 1, Type).
Type = int.
?- type("hello world", Type).
Type = string.
?- type(\x:int -> x, Type).
Type = int -> int.
?- type(\x -> x, Type).
Type = ?42 -> ?42.
?- type(\x -> x, int -> Type).
Type = int.
```

Going in the reverse direction, you can query the expression:

```
?- 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 - Program Synthesis

Assume you have implemented a *relational interpreter*:

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

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

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

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

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Quine generation is pretty straightforward:

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

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 raise the level of abstraction, by enabling the programmer to express *what* needs to be done, instead of *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 containing Haskell code
    -- e.g. janus_variable += <Haskell code>
  }
|]
encode :: Image -> [Byte]
encode = call encode
decode :: [Byte] -> Image
decode = uncall encode
```

Come and check out our poster in de Vagant!

Thanks!

Feel free to ask any questions

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]