Types for Programs and Proofs

Programming and proving in Agda

Thierry Coquand

Plan

First hour, `bb2.agda`

- Turing machines as Agda programs; the Busy Beaver champions
- running a machine is not proving anything about it
- "this machine never stops" as a *type*, and a proof of it

Second hour, `euclidean.agda`, `Gentzen.agda`

- propositions as types, proofs as programs (Curry–Howard): the connectives, $\forall$, $\exists$; what can be proved, and what cannot

A program with five states

A Turing machine: a tape of cells holding `0` or `1`, a head reading one cell, and a finite table. Each entry says what to write, where to move (`L` or `R`), and which state comes next

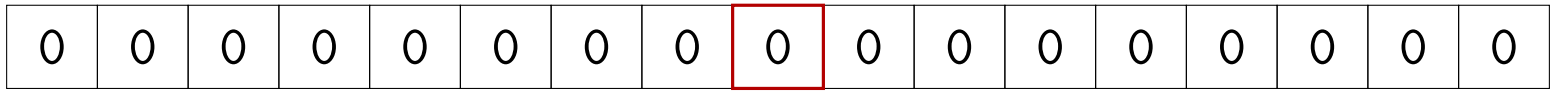
	0	1
A	1RB	1LC
B	1RC	1RB
C	1RD	0LE
D	1LA	1LD
E	---	0LA

`---`: no entry, the machine *halts*. Started on a blank tape in state `A`: does it halt?

Does it halt?

Step 0

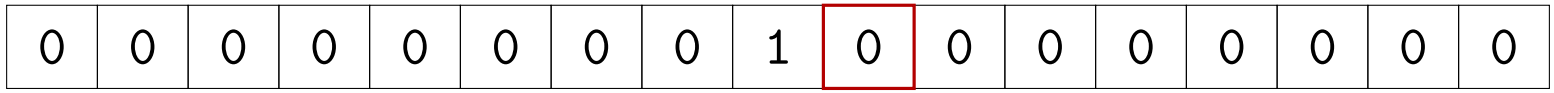
	0	1
A	1RB	1LC
B	1RC	1RB
C	1RD	0LE
D	1LA	1LD
E	---	0LA



Does it halt?

Step 1

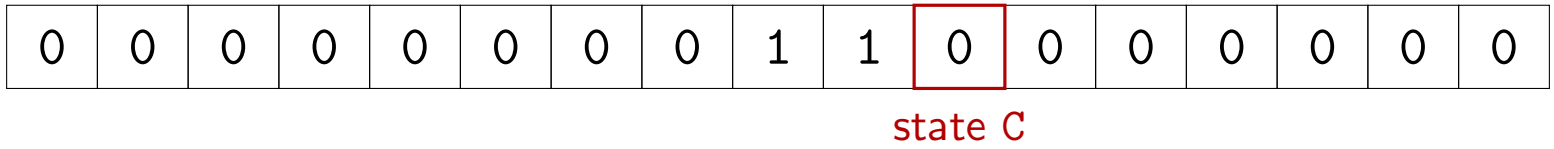
	0	1
A	1RB	1LC
B	1RC	1RB
C	1RD	0LE
D	1LA	1LD
E	---	0LA



Does it halt?

Step 2

	0	1
A	1RB	1LC
B	1RC	1RB
C	1RD	0LE
D	1LA	1LD
E	---	0LA



Does it halt?

Step 3

	0	1
A	1RB	1LC
B	1RC	1RB
C	1RD	0LE
D	1LA	1LD
E	---	0LA

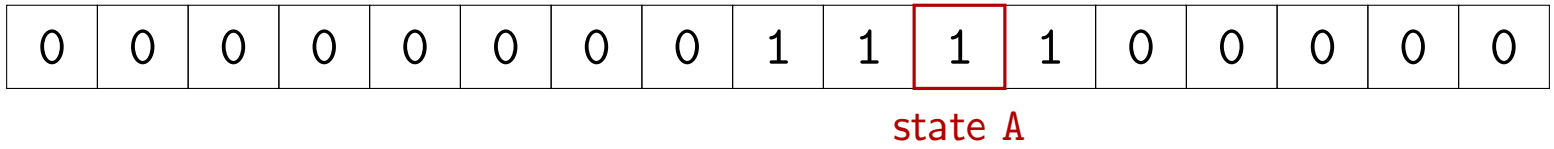
0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0

state D

Does it halt?

Step 4

	0	1
A	1RB	1LC
B	1RC	1RB
C	1RD	0LE
D	1LA	1LD
E	---	0LA



Does it halt?

Step 5

	0	1
A	1RB	1LC
B	1RC	1RB
C	1RD	0LE
D	1LA	1LD
E	---	0LA

0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0

state C

Does it halt?

Step 6

	0	1
A	1RB	1LC
B	1RC	1RB
C	1RD	0LE
D	1LA	1LD
E	---	0LA

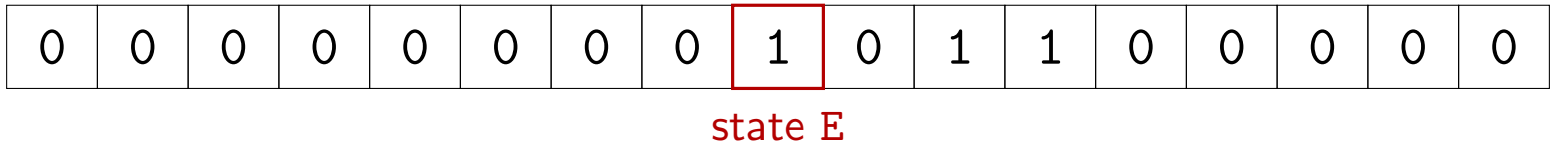
state E

Does it halt?

Step 6

Will we ever read a 0 in state E?

	0	1
A	1RB	1LC
B	1RC	1RB
C	1RD	0LE
D	1LA	1LD
E	---	0LA

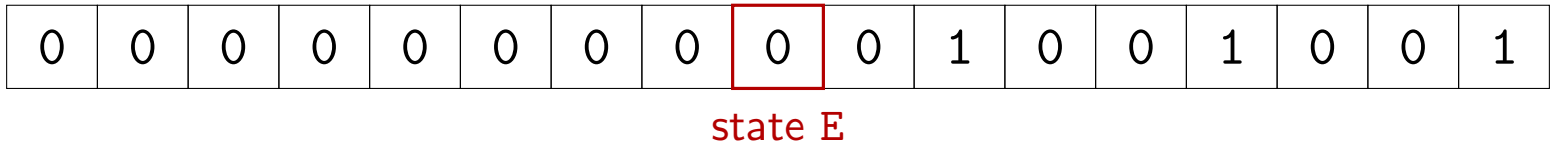


Does it halt? Yes!

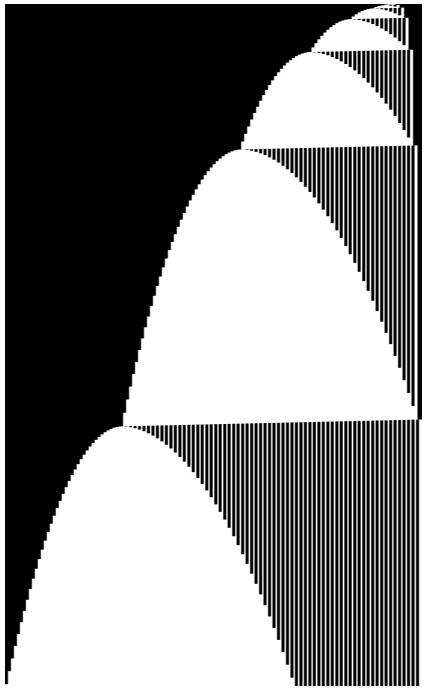
Step 47 176 870

State `E` reads a `0`: the machine *halts*, with 4 098 marks on the tape

	0	1
A	1RB	1LC
B	1RC	1RB
C	1RD	0LE
D	1LA	1LD
E	---	0LA



The first 20 000 steps



The *space-time diagram*: each row is the tape after one more step, white = 1, black = 0

The pattern grows, and comes back; nobody can see from this picture that the machine stops after 47 176 870 steps

Can another 5-state machine run longer and still halt?

A Turing machine in Agda

```
data Symbol : Set where zero one : Symbol
data State  : Set where qA qB qC qD qHalt : State
data Dir    : Set where L R : Dir

record Tape : Set where            -- cells left of the head, the head, cells right
  field left : List Symbol ; head : Symbol ; right : List Symbol

Transition = State → Symbol → State × Symbol × Dir      -- a machine is a function

step : Transition → State → Tape → State × Tape
run  : Nat → Transition → State → Tape → State × Tape   -- at most n steps
```

Everything is a total program: `run` is bounded by its first argument, so Agda accepts it. A machine that never stops cannot be run to the end

The Busy Beaver champions

$BB(n)$: the most steps an n-state machine can make and then halt

$BB(2) = 6$	$BB(3) = 21$	$BB(4) = 107$ (Brady 1983)
$BB(5) = 47\,176\,870$	bbchallenge, 2024	proved in Rocq
$BB(6)$	open	known to be larger than $2 \uparrow\uparrow\uparrow 5$

In Agda: `bb2`, `bb3`, `bb4` are transition functions. Evaluate `srun big bb4` with `C-c C-n`

Better: state it and let the type checker run the machine

```
bb4-halts : fst (srun big bb4) ≡ qHalt
bb4-halts = refl
```

Running is not proving

```
forever : Transition                    -- stay in qA, move right
forever qA zero = (qA , zero , R)
forever q  s    = (q  , s    , R)
```

After 320 steps it is still in `qA`. After a million steps too. No amount of running shows that it *never* stops

We need to say "never stops" in the language, and prove it

- "the state s is halting" must become a *type* `Stop s`
- "for every n" must become a function of n
- "not" must become a function into the empty type

Propositions as types (Curry 1934, Howard 1969, de Bruijn, Martin-Löf)

proposition	type	a proof is
$\bot$	`data ⊥ : Set where`	nothing: no constructor
$\top$	`data ⊤ : Set where tt : ⊤`	`tt`
$A \Rightarrow B$	`A → B`	a function
$\neg A$	`A → ⊥`	a function into the empty type
$A \wedge B$	`A × B`	a pair
$A \vee B$	`A + B`	`inl a` or `inr b`
$\forall x{:}A.\ B\,x$	`(x : A) → B x`	a function whose *type* depends on x
$\exists x{:}A.\ B\,x$	`Σ A B`	a pair `(a , b)` with `b : B a`

A proof is a program; checking the proof is type checking the program

$\forall$ and $\exists$ need types that depend on values: *dependent types*

Predicates defined by computation

```
Stop : State → Set                    -- "s is the halting state"
Stop qHalt = ⊤
Stop _     = ⊥

QA : State → Set                      -- "s is qA", likewise

QA-notStop : (s : State) → QA s → ¬ (Stop s)
QA-notStop qA    p h = h              -- h : ⊥
QA-notStop qB    p h = p              -- p : ⊥
```

Case analysis on `s`; in each case the types *compute* to $\top$ or $\bot$ and the proof is trivial

The checker unfolds definitions silently: the *Poincaré principle*

A proof that a machine never stops

```
neverStop : Transition → Set
neverStop δ = (n : Nat) → ¬ (Stop (fst (srun n δ)))

staysInQA : (n : Nat) (l : List Symbol) → QA (fst (run n forever qA (tape l zero [])))
staysInQA zero    l = tt
staysInQA (suc n) l = staysInQA n (zero :: l)

theorem : neverStop forever
theorem n = QA-notStop (fst (srun n forever)) (staysInQA n [])
```

- a proof of $\forall n$ is a function of `n`; defined by recursion $=$ proof by *induction*
- the statement is *generalised* to any tape on the left, since each step adds a cell; the step type checks because `run` *computes*

What we cannot expect to prove

```
willStop δ = Σ Nat (λ n → Stop (fst (srun n δ)))        -- exists n
Halting  δ = neverStop δ + willStop δ
```

A proof of $(\delta$: `Transition`$) \to$ `Halting` δ would be a program which, given any machine, answers `inl` or `inr`: it would decide the halting problem

So $A \vee \neg A$ is not provable for every A: a proof of $A + \neg A$ *tells which one*. This is the *constructive* reading of $\vee$

Nothing is lost: for a *given* machine we can still prove one side, as we just did

Second hour: natural deduction is programming

Gentzen 1934: each connective has *introduction* and *elimination* rules

	introduction	elimination
$\Rightarrow$, $\forall$	λ `x` $\rightarrow$ `...`	application `f a`
$\wedge$, $\exists$	the pair `(a , b)`	`fst`, `snd`, or pattern matching
$\vee$	`inl`, `inr`	pattern matching on `inl`/`inr`
$\bot$	none	the empty pattern `()`

Fitch's boxes for hypotheses are λ-abstractions

`euclidean.agda`: de Bruijn's example again, and $2 + 2 = 4$ by computation

`Gentzen.agda`: $(A \wedge B) \vee (A \wedge C) \Rightarrow A \wedge (B \vee C)$, written both as a term and by pattern matching

Beyond first-order logic

The elimination rule for $\vee$ is stronger than in logic: the conclusion may *depend* on the proof

```
elim-+ : (C : A + B → Set) → ((x : A) → C (inl x)) → ((y : B) → C (inr y))
       → (z : A + B) → C z
```

The axiom of choice is a one-line program

```
ax-choice : ((x : A) → Σ B (R x)) → Σ (A → B) (λ f → (x : A) → R x (f x))
ax-choice h = (λ x → fst (h x)) , (λ x → snd (h x))
```

And quantifiers and negation

```
not-exists : ¬ (Σ A B) → (x : A) → ¬ (B x)          -- provable
not-forall : ¬ ((x : A) → B x) → Σ A (λ x → ¬ (B x))  -- a hole that cannot be filled
```

Summary

- A proposition is a type; a proof is a program of that type; a predicate is a type depending on a value
- $\forall$ is a dependent function, $\exists$ is a dependent pair; induction is recursion
- Definitions unfold by computation, so many cases are trivial once the types have computed
- $\vee$ and $\exists$ carry information; $A \vee \neg A$ would decide the halting problem

Exercises: `exercises3.agda`; prove `neverStop alternate`; count the steps of `bb4`. Next: equality (lecture 4)