

COMP30026 Models of Computation

Well-foundedness and Termination

Harald Søndergaard

Lecture Week 10 Part 1

Semester 2, 2020

Partial Functions

We have silently assumed that the domain of a function is known, so that $f : X \rightarrow Y$ means that $f(x)$ is defined for each $x \in X$.

In computer science, however, it is often more appropriate to deal with functions that are **partial**.

We write $f : X \hookrightarrow Y$ to say that f has a domain which is a subset of X , but $f(x)$ may be undefined for some $x \in X$.

Note that a total function $f : X \rightarrow Y$ is by definition also partial: $f : X \hookrightarrow Y$.

Partial Functions: Example 1

The function f defined by

$$f(n) = \begin{cases} 42 & \text{if } n = 0 \\ f(n-2) & \text{if } n \neq 0 \end{cases}$$

is a **partial** function $f : \mathbb{Z} \hookrightarrow \mathbb{Z}$.

In a natural interpretation, it is **undefined** if n is odd and/or negative. Its range is $\{42\}$.

In this case, it is not too hard to determine the set of values for which f is defined. So we could also choose to say that f is a **total** function $X \rightarrow \mathbb{Z}$, where $X = \{n \in \mathbb{Z} \mid n \geq 0 \wedge n \text{ is even}\}$.

However, it is not always easy, or even possible, to determine a function's domain.

Partial Functions: Example 2

The function c defined by

$$c(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ c(n/2) & \text{if } n \text{ is even and } n > 1 \\ c(3n + 1) & \text{if } n \text{ is odd and } n > 1 \end{cases}$$

is a partial function $c : \mathbb{N} \hookrightarrow \mathbb{N}$ with range $\{1\}$.

Will the evaluation of $c(n)$ terminate for all n ?

It is not known whether c is total.

This is the so-called $3n + 1$ problem, or **Collatz's problem**.

Program Termination

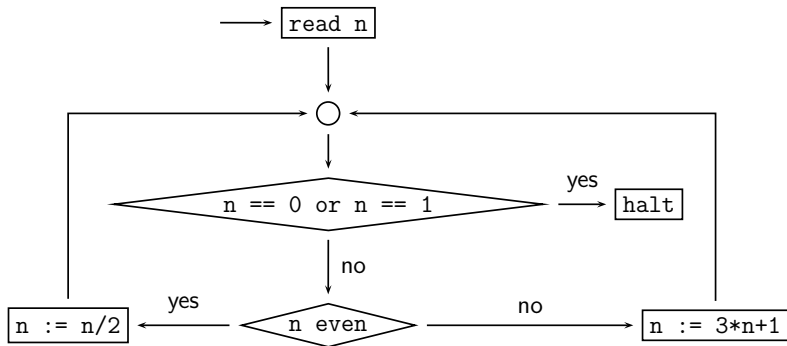
Here is a Haskell program that produces the list of n -values generated in successive recursive calls:

```
c :: Integer -> [Integer]
c 0 = [1]
c 1 = [1]
c n = n : c (if even n then n `div` 2 else 3*n+1)
```

Try it out for different values of n , such as 23 or 27.

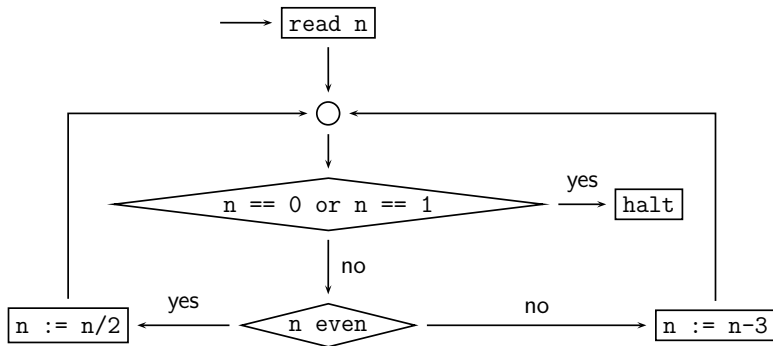
Program Termination

Here it is as a flowchart program:



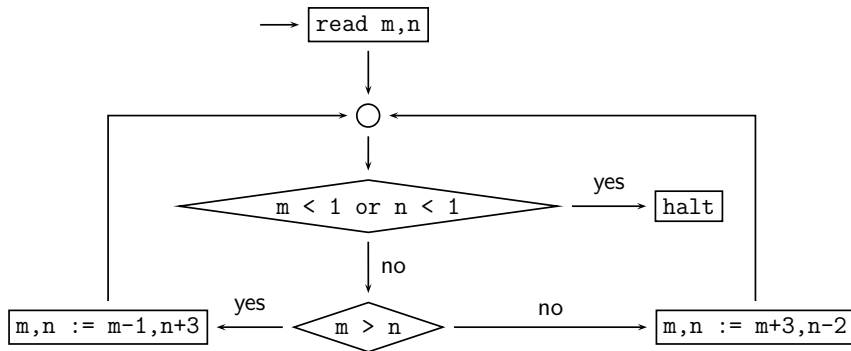
Quiz 1: Does It Terminate?

Here is a variant. Does this halt for all positive input?



Quiz 2: Does It Terminate?

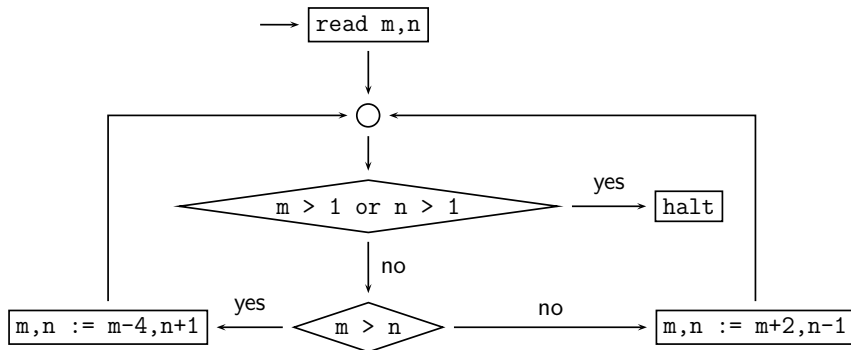
How about this program? Does it halt for all input?



Note that **something** (either m or n) decreases in each iteration.

Quiz 3: Does It Terminate?

And how about this? Does it halt for all input?



Proving Termination

Suppose that, for each loop in a program, we can find some “measure” (a function of the program variables) such that

- 1 the measure is a natural number, and
- 2 the measure gets smaller with each loop iteration.

Then the program must terminate for all input, because a natural number cannot be made smaller indefinitely.

The Termination Question

Termination of algorithms is a tricky problem (and the general problem is **undecidable**).

For example, we suggested earlier algorithms for translating propositional formulas to, say, DNF, including rules like

$$\begin{aligned}\neg(\alpha \wedge \beta) &\rightsquigarrow \neg\alpha \vee \neg\beta \\ \neg(\alpha \vee \beta) &\rightsquigarrow \neg\alpha \wedge \neg\beta \\ \neg\neg\alpha &\rightsquigarrow \alpha \\ \alpha \wedge (\beta \vee \gamma) &\rightsquigarrow (\alpha \wedge \beta) \vee (\alpha \wedge \gamma) \\ (\beta \vee \gamma) \wedge \alpha &\rightsquigarrow (\beta \wedge \alpha) \vee (\gamma \wedge \alpha)\end{aligned}$$

Note that some of the rules decrease the size of a term, while others increase it (and some duplicate certain subterms).

So why should this process terminate?

Quiz 4: A Marble Game

You have a bag of red and white marbles, plus a box of red marbles.

Repeat this process:

- If the bag contains exactly one marble, halt; otherwise take two random marbles from the bag.
- If they are of different colours, put them back.
- If they are of the same colour, discard both, but put one red marble (from the box) in the bag.

Does this terminate?

Quiz 5: Another Marble Game

You have a bag of red and white marbles, plus a box of red marbles.

Repeat this process:

- If the bag contains exactly one marble, halt; otherwise take two random marbles from the bag.
- If they are of different colours, return the white marble to the bag, and discard the red one.
- If they are of the same colour, discard both, but put one red marble (from the box) in the bag.

Clearly this must terminate.

What can we say about the last marble in the bag?

Well-Founded Orderings

The binary relation \prec over some set X is **well-founded** iff there is no infinite sequence of X -elements x_1, x_2, x_3, \dots such that

$$x_1 \succ x_2 \succ x_3 \succ \dots$$

We say that (X, \prec) is a **well-founded structure**.

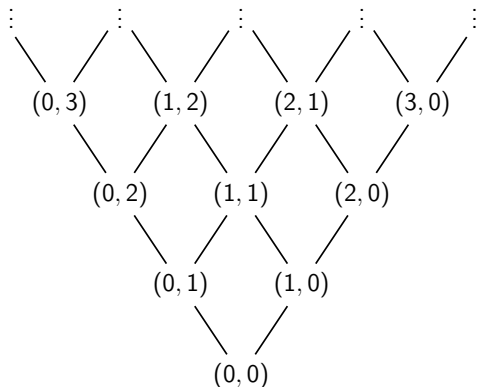
For example, $(\mathbb{N}, <)$ is a well-founded structure.

Given a finite number of well-founded structures $(X_1, \prec_1), \dots, (X_n, \prec_n)$, we can obtain well-founded orderings of $X = X_1 \times \dots \times X_n$ in a number of different ways.

Ordering Pairs: Component-Wise Ordering

$$(x_1, x_2) \preceq (y_1, y_2) \text{ iff } x_1 \leq y_1 \wedge x_2 \leq y_2$$

$$p \prec q \text{ iff } p \preceq q \wedge p \neq q$$



A **Hasse diagram** for component-wise ordering of $\mathbb{N} \times \mathbb{N}$.

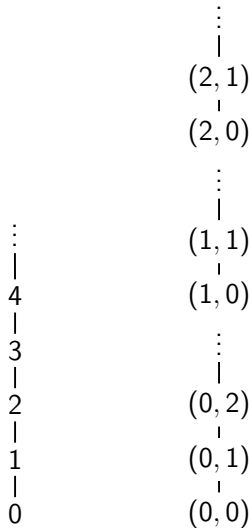
Values increase as you travel **up** along edges.

Ordering Pairs: Lexicographic Ordering

On the left: \mathbb{N} with the usual ordering.

On the right: $(\mathbb{N} \times \mathbb{N}, \preceq)$, where \preceq is **lexicographic** ordering: $(m, n) \preceq (m', n')$ iff $m \leq m' \wedge (m = m' \Rightarrow n \leq n')$.

Define again $p \prec q$ iff $p \preceq q \wedge p \neq q$.



Well-Founded Orderings on Tuples

Theorem: If \prec is well-founded then so is its component-wise extension to tuples.

Theorem: If \prec is well-founded then so is its lexicographic extension to tuples.

Component-Wise Ordering of Tuples

Ordering $X = X_1 \times \cdots \times X_n$ **component-wise**:

$$(x_1, \dots, x_n) \preceq (y_1, \dots, y_n) \text{ iff } \bigwedge_{i=1}^n x_i \preceq_i y_i$$

If each (X_i, \prec_i) is well-founded, then so is (X, \prec) .

Example using component-wise ordering:

$$(2, 2, 2) \succ (2, 2, 1) \succ (2, 1, 1) \succ (2, 0, 1) \succ (2, 0, 0) \succ (0, 0, 0)$$

Lexicographic Ordering of Tuples

Ordering $X = X_1 \times \dots \times X_n$ **lexicographically**:

$$(x_1, \dots, x_n) \prec (y_1, \dots, y_n) \text{ iff } \bigvee_{i=1}^n \left(x_i \prec_i y_i \wedge \bigwedge_{j=1}^{i-1} x_j = y_j \right)$$

If each (X_i, \prec_i) is well-founded, then so is (X, \prec) .

Example using lexicographic ordering:

$$(2, 2, 2) \succ (2, 1, 42) \succ (1, 3, 1000) \succ (1, 3, 999) \succ (1, 3, 0) \succ (0, 0, 15)$$

Well-Founded Induction

There is an **induction** principle to go with well-founded relations.





Given a well-founded structure (X, \prec) we can prove a statement “for all $x \in X$, $S(x)$ ” as follows:





We proceed in **one** step:





- Assume that $S(x')$ holds for all $x' \prec x$, and use that to establish $S(x)$.

Example 1: The Dutch Flag




We have 12 pebbles laid out in a row. Consider three rewrite rules:

  \rightsquigarrow   (for a white left of a red, swap)

  \rightsquigarrow   (for a blue left of a red, swap)

  \rightsquigarrow   (for a blue left of a white, swap)



To see that rewriting terminates, use  $<$  $<$  together with lexicographic ordering on 12-tuples.

Example 2: Ackermann's Function

The following is a definition of **Ackermann's function**:

$$\begin{aligned} \text{ack}(0, y) &= y + 1 \\ \text{ack}(x + 1, 0) &= \text{ack}(x, 1) \\ \text{ack}(x + 1, y + 1) &= \text{ack}(x, \text{ack}(x + 1, y)) \end{aligned}$$

It grows incredibly fast.

For example, $\text{ack}(3, 3) = 61$ and $\text{ack}(4, 4) = 2^{2^{2^{2^{16}}}} - 3$.

However, **lexicographic** well-founded induction allows us to conclude that the function is total—as a Haskell program it will terminate for all input (possibly after a very long time).

Example 2: Ackermann's Function

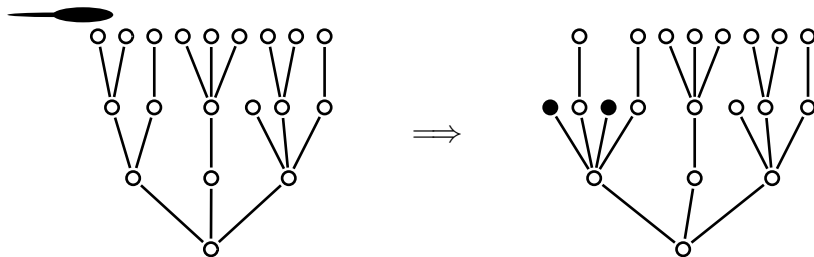
In each recursive call, the argument (x, y) decreases strictly:

$$\begin{aligned} \text{ack}(x + 1, 0) &= \overbrace{\text{ack}(x, 1)}^{x \text{ smaller}} \\ \text{ack}(x + 1, y + 1) &= \underbrace{\text{ack}(x, \underbrace{\text{ack}(x + 1, y)}_{x \text{ same, } y \text{ smaller}})}_{x \text{ smaller}} \end{aligned}$$

Example 3: Hercules vs Hydra

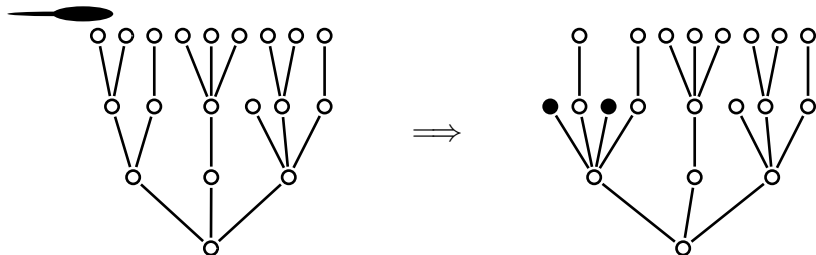
Hercules is up against the many-headed monster Hydra.

He can use his club to knock out a topmost head, but the head below will instantly gain two siblings.



Will Hercules manage to defeat the monster?

Example 3: Hercules vs Hydra



Look at the tuple $(n_{depth}, n_{depth-1}, \dots, n_1)$ where n_i is the number of heads at level i .

$$(9, 6, 3, 1) \implies (8, 8, 3, 1)$$

Each Herculean strike will reduce the tuple **lexicographically**, until (after 84 strikes) we reach $(0, 0, 0, 103)$, and another 103 strikes will finish the job.

Example 4: The Unification Algorithm

In Week 5 unification was presented as 6 rewrite rules. Will repeated application of those rules always halt? We have, for example,

$$\left\{ \begin{array}{l} f(a, b) = u \\ u = f(v, w) \\ x = f(u, h(a, b)) \\ y = g(x, h(z, x)) \end{array} \right\} \Rightarrow \left\{ \begin{array}{l} u = f(a, b) \\ u = f(v, w) \\ x = f(u, h(a, b)) \\ y = g(x, h(z, x)) \end{array} \right\} \Rightarrow$$

$$\left\{ \begin{array}{l} u = f(a, b) \\ f(a, b) = f(v, w) \\ x = f(f(a, b), h(a, b)) \\ y = g(x, h(z, x)) \end{array} \right\} \Rightarrow \left\{ \begin{array}{l} u = f(a, b) \\ a = v \\ b = w \\ x = f(f(a, b), h(a, b)) \\ y = g(x, h(z, x)) \end{array} \right\}$$

Almost anything can grow in one step, including the number of equations and the size and the depth of terms.

Example 4: The Unification Algorithm

Another rewrite sequence for the same set:

$$\left\{ \begin{array}{l} f(a, b) = u \\ u = f(v, w) \\ x = f(u, h(a, b)) \\ y = g(x, h(z, x)) \end{array} \right\} \Rightarrow \left\{ \begin{array}{l} f(a, b) = f(v, w) \\ u = f(v, w) \\ x = f(f(v, w), h(a, b)) \\ y = g(x, h(z, x)) \end{array} \right\}$$

$$\Rightarrow \left\{ \begin{array}{l} f(a, b) = f(v, w) \\ u = f(v, w) \\ x = f(f(v, w), h(a, b)) \\ y = g(f(f(v, w), h(a, b)), h(z, f(f(v, w), h(a, b)))) \end{array} \right\}$$

Note how the number of occurrences of function symbols (including constants) may grow.

Is there anything that is getting strictly smaller in each step?

Example 4: The Unification Algorithm

The proof that the rules really do terminate in all cases is non-trivial and involves a clever choice of a number of different “measures” which, when ordered the right way as a tuple, decrease lexicographically in each step.

Can you find a set of measures and place them in a tuple, so that lexicographic ordering of tuples is well-founded for this problem?

Next Up

After the break we will take a look at Turing machines, our gateway to the study of computability.