JFP **35**, e10, 21 pages, 2025.

doi:10.1017/S0956796825000061

FUNCTIONAL PEARL

Binary search—think positive

ALEXANDER DINGES
University of Kaiserslautern-Landau (RPTU), Rheinland-Pfalz, Germany
(*e-mail:* alexander.dinges@cs.rptu.de)

RALF HINZE
University of Kaiserslautern-Landau (RPTU), Rheinland-Pfalz, Germany
(*e-mail:* ralf.hinze@cs.rptu.de)

1 The cast and characters

The setting is a tutorial on program verification in Agda. Please consult the programme for further details. [*See also Appendix A.*]

Dramatis personæ

Tutor	A permanent resident of the ivory tower.
.....	[She wears socks and sandals.]
Lisa Lista ...	A math student with secondary subject CS.
.....	[She likes a good mathematical argument.]
Harry Hacker	A Haskell geek.
.....	[He likes to type, but is still a stranger to Agda.]
Ken	An Agda expert.
.....	[He even has Σ and Π printed on his bed sheets.]
Lambert	A retired math teacher.
.....	[He likes to calculate.]

Dramatis locus

A small but pleasant room in some medieval college building.

2 Monday

Synopsis: The tutor is on a mission: she wants to dispel the common misconception that binary search is about searching an *ordered* table.

Tutor: Good morning class. This week I would like to discuss formalising one of the standard algorithms every programmer and computer science student should know: *binary search*.

Harry: Aah, searching an ordered table as described by Knuth (1973, Section 6.2.1)? That's pretty straightforward, isn't it? We assume that we have a linearly sorted sequence of keys $K_1 \prec \cdots \prec K_n$. Given a key K, we would like to know if there is an index i such that $K_i \equiv K$. We pick an arbitrary index m with $1 \preceq m \preceq n$ and compare K to K_m. There are three possible outcomes with resulting actions:

- $K \prec K_m$: we eliminate the keys $K_m \, , \ldots \, , K_n$ from consideration.
- $K \equiv K_m$: the search is done.
- $K \succ K_m$: we eliminate the keys $K_1 \, , \ldots \, , K_m$ from consideration.

The problem can be solved in logarithmic time, if we iterate this step and the index picked is always the midpoint, $m \; := \; \lfloor \, (1 + n) \; / 2 \rfloor$. This is really old hat!

Lisa (*frowns*)**:** That's a bit too concrete for my taste. I actually prefer Bird&Wadler's formalisation (1988). Given naturals l and r and a predicate $P \; : \; \mathbb{N} \to \mathit{Set}$, we are looking for the smallest index $i \in [\; l \, , \, r \;]$ such that $P \; i$ holds. If the predicate is monotone, $\forall \; a \; b \to a \preceq b \to (P \; a \to P \; b)$, then the problem can be solved in sub-linear time. We pick an index $m \in [\; l \, , \, r \;]$ and check $P \; m$. There are two possible outcomes with resulting actions:

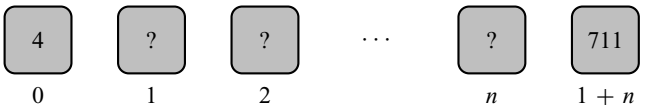
- $P \; m$ holds: we eliminate the indices $1 + m \, , \ldots \, , r$ from consideration.
- $P \; m$ does not hold: we eliminate the indices $l \, , \ldots \, , m$ from consideration.

Like in Harry's setup, the algorithm runs in logarithmic time, if we always pick the midpoint, $m \; := \; \lfloor \, (l + r) \; / 2 \rfloor$.

Lambert: But there is no guarantee that a suitable index actually exists. Did you notice that the programs provided by Bird&Wadler are partial? Likewise, searching an ordered table for a certain key may fail.

Tutor (*smiles*)**:** These points are all well taken, but I suggest that we ignore them for the moment and start afresh, solving a little puzzle instead. Agreed? [*The students nod.*]

You are presented with the following sequence of $2 + n$ boxes, each containing a natural number. The numbers are hidden from you; only the contents of the first and of the last box is revealed.



Your goal is to find an *even–odd pair*, two neighbouring boxes such that the contents of the left box is even and the contents of its right neighbour is odd. Open as few boxes as possible.

Lisa: The problem description seems to suggest that an even–odd pair always exists. Perhaps, we should first prove this implicit assumption.

Harry: But is it actually true? After all, the boxes might contain only even numbers, or only odd numbers, or a sequence of odd numbers followed by a sequence of even numbers.

Lisa (*frowns again*)**:** Harry, these possibilities are ruled out as we can observe that the first box contains an even number and the last box an odd number.

Lambert: So the existence of an even–odd pair depends on a *precondition*.

Harry: Ah, I see. And we know that there are at least two boxes. That's another precondition, right?

Lisa: Good point. The proof of existence then proceeds by induction on n. If n is 0, we have found an even–odd pair. Otherwise, we open the second box, the box numbered 1. If we are lucky, it contains an odd number. If the number is even, we remove the leftmost box, the box numbered 0, and apply the same strategy to the residual sequence of $1 + n$ boxes. All preconditions are satisfied: there are at least two boxes; the contents of the new leftmost box is even; the contents of the rightmost box is still odd. Consequently, an even–odd pair is bound to exist.

Ken: I guess we can start turning Lisa's proof into Agda code? I'd like to suggest a small change though: the problem description builds on zero-based intervals. It seems prudent to follow Bird&Wadler's lead, that is, to consider arbitrary intervals. Otherwise, we need to shift the interval in the recursive step. [*The others signal agreement.*] The proposition then reads:

$$
\begin{aligned}
\mathit{even\text{-}odd\text{-}pair} \;:\; & \forall\, (l\ r \;:\; \mathbb{N}) \\
& \to l \prec r \\
& \to (\mathit{box} \;:\; \mathbb{N} \to \mathbb{N}) \\
& \to \quad\quad \mathit{even}\ (\mathit{box}\ l) \times \mathit{odd}\ (\mathit{box}\ r) \\
& \to \exists\, (\lambda\, i \to \mathit{even}\ (\mathit{box}\ i) \times \mathit{odd}\ (\mathit{box}\ (1 + i)))
\end{aligned}
$$

The sequence of boxes is represented by a function $\mathit{box} \;:\; \mathbb{N} \to \mathbb{N}$ that maps positions to contents ...

Harry: ... and the precondition $l \prec r$ ensures that there are at least two boxes! Can you remind me what the type $\exists\, (\lambda\, i \to \ldots)$ means?

Ken: This is actually a Σ-type, a dependent pair type. In constructive logic, an existential quantification is given by a pair, consisting of a witness and a proof that the witness satisfies the stated proposition. It's a *dependent* pair as the proof depends on the witness. In our scenario, this pair consists of a natural number i and two proofs: one holding evidence that $\mathit{box}\ i$ is an even number, and the other showing that $\mathit{box}\ (1 + i)$ is an odd number.

Harry: Thanks. However, I am not sure that the setup works. I foresee problems with Agda's termination checker. [... *lowers his voice mumbling* ...] Agda is pretty picky when it comes to termination.

Lisa: Harry is right. My argument is neither inductive on l, nor on r. Perhaps we should represent the interval $[\ l\ ,\ r\ ]$ differently, say, by offset and size, so that we can induct over the size?

Tutor: Don't give up so easily. A few weeks ago, we formalised the strict order on the natural numbers in several different ways. [*She pauses.*] Anyone?

Lisa: Of course! The endpoint l is strictly smaller than r if and only if the difference $r \mathbin{\dot{-}} l$ is positive.

Tutor: Should you need a reminder about notation such as _$\dot{-}$_, please check the course material [*Appendix A*].

Harry: Ah, I see the light. For the task at hand, we could use the variant of the strict order that builds on the difference of the endpoints!

```
data _≺_ : ℕ → ℕ → Set where
  one  : n ≺ 1 + n
  succ : 1 + m ≺ n → m ≺ n

_≼_ : ℕ → ℕ → Set
m ≼ n = (m ≡ n) ⊎ (m ≺ n)
```

The proposition $4 \prec 7$, for example, is evidenced by *succ* (*succ one*), as the difference of seven and four is three. Honestly, at the time I didn't see the point of formalising the order in umpteen different ways.

Ken: I guess we can start turning Lisa's proof into Agda code now? May we assume that the predicates *even* and *odd* are provided from somewhere?

Tutor: Sure, and you may use the function $parity : \forall\, n \to even\ n \uplus odd\ n$ that decides whether a natural is even or odd.

Harry (*busily typing*)**:** This is a breeze. When I conduct the case analysis, Agda is able to fill in the rest ... Well, almost, I have to specify the left endpoint for the recursive call:

```
even-odd-pair l r (one) box (e , o) = l , e , o
even-odd-pair l r (succ 1+l≺r) box (e , o) with parity (box (1 + l))
... | inr o′ = l , e , o′
... | inl e′ = even-odd-pair (1 + l) r 1+l≺r box (e′ , o)
```

The first equation deals with the case that there are only two boxes. Otherwise, we open the box numbered $1 + l$. There are now two cases that we can investigate using ***with***. In the first one, the box numbered $1 + l$ contains an odd number, evidenced by o'. Thus, we have completed the task. In the second case, the box contains an even number, evidenced by e'. Here, we recurse on the smaller interval ranging from $1 + l$ to r.

Lambert: Sorry, I don't use Agda on a regular basis. What does $1{+}l{\prec}r$ mean?

Ken: That's an Agda idiosyncrasy. Remember that in Agda only a space separates.[1] The sequence of characters $1{+}l{\prec}r$ contains no space, so this is *one* identifier of type $1 + l \prec r$. The name typically reflects the type.

Tutor: Well done. Lisa's proof has algorithmic content, detailing a simple search strategy: open the boxes sequentially from left to right until an even–odd pair is found. So the Agda code captures left-to-right *sequential search*. But, we set out to formalise *binary search*.

Harry: Ha, that should be easy. We simply change the definition of the standard order on the naturals to reflect the new search strategy. [*Starts typing.*] Is there any notation for the midpoint or centre of an interval?

Lambert: Good question. I don't think there is. But, may I suggest one? What about $\lfloor\ a\ \mathbf{mid}\ b\ \rfloor$? The floor brackets indicate that the real midpoint is rounded downwards, and the notation is a slightly shorter than $\lfloor\ (a + b)\ /2 \rfloor$.

[1] There are a few exceptions to the rule, most notably parentheses, '(' and ')', and curly braces, '{' and '}'. These characters are not allowed to appear in a name at all.

Harry: Okey-dokey. [*Continues typing.*] Voilà. I have simply changed the final line of the datatype definition. The new constructor states that if the midpoint of l and r is greater than l and less than r, then $l \prec' r$. The new name reflects the fact that the sizes of the sub-intervals are added.

```
data _≺'_ : ℕ → ℕ → Set where
  one  : n ≺' 1 + n
  _+̇_  : l ≺' ⌊ l mid r ⌋ → ⌊ l mid r ⌋ ≺' r → l ≺' r
```

Lisa (*a connoisseur of sequence implementations*)**:** Interesting, this reminds me of the difference between standard lists and join lists, well, balanced join lists.

Harry (*ignoring Lisa*)**:** . . . and the implementation of *even-odd-pair* can be easily adapted, as well. The recursive cases are now nicely symmetric.

```
even-odd-pair' l r (one)              box (e , o) = l , e , o
even-odd-pair' l r (l≺'m +̇ m≺'r) box (e , o) with parity (box ⌊ l mid r ⌋)
... | inr o' = even-odd-pair' l ⌊ l mid r ⌋ l≺'m box (e , o')
... | inl e' = even-odd-pair' ⌊ l mid r ⌋ r m≺'r box (e' , o)
```

Tutor: Well done, again. We have stressed repeatedly that proving *is* programming. As programmers, we appreciate that an abstract datatype, like the type of sequences, can be implemented in a variety of different ways. Likewise, an abstract concept such as an ordering relation can be evidenced in many different ways. Choosing a concrete representation for an ordering is as important as choosing a concrete data structure for a sequence type.

Lisa (*with a sceptical undertone*)**:** It's a bit odd (pun intended) that the proof doesn't use any properties of the mid-point, isn't it? In particular, we do not make use of the fact that the midpoint lies between the endpoints.

Tutor: Oh, is that the time already? Think about Lisa's objection for tomorrow's tutorial.

3 Tuesday

Synopsis: By abstracting from a particular search strategy, the tutees steer the development in an unforeseen direction.

Ken: I have given Lisa's comment some thought and I think she has a point. However, I consider this a feature, not a bug, an opportunity to generalise the development. Do you mind if I share some code with you? [*He opens his laptop.*] First, I have generalised the two definitions of order.

```
data ⟦_,_⟧ : ℕ → ℕ → Set where
  Leaf : (n : ℕ) → ⟦ n , 1 + n ⟧
  Node : (m : ℕ) → ⟦ l , m ⟧ → ⟦ m , r ⟧ → ⟦ l , r ⟧
```

Both variants of the strict order discussed yesterday can be seen as detailing a *strategy*, a strategy for exploring an interval. The first captures a sequential search from left to right, the second a binary subdivision scheme. The new type allows us to specify arbitrary strategies. [*He glances at his fellow students who look confused.*] Ah, sorry, I have renamed the type—I like to think of its elements as *interval trees*; the names reflect this.

Harry (*still confused*)**:** So $one : n \prec 1 + n$ is now $Leaf\ n : ⟦ n , 1 + n ⟧$?

Ken: Yes, and the second constructor generalises _∔_. Both constructors take an explicit argument—you will see why in a moment. Perhaps, it is instructive to look at an example tree first? [*The others nod in approval.*] Here is a strategy for exploring the interval ⟦ 0 , 17 ⟫.

```
Node 8 (Node 4  (Node 2  (Node 1 (Leaf 0) (Leaf 1)) (Node 3 (Leaf 2) (Leaf 3)))
                (Node 6  (Node 5 (Leaf 4) (Leaf 5)) (Node 7 (Leaf 6) (Leaf 7))))
       (Node 12 (Node 10 (Node 9 (Leaf 8) (Leaf 9)) (Node 11 (Leaf 10) (Leaf 11)))
                (Node 14 (Node 13 (Leaf 12) (Leaf 13))
                         (Node 15 (Leaf 14) (Node 16 (Leaf 15) (Leaf 16)))))
```

Harry: Sorry, I need to draw a picture. [*He sketches the interval tree on the whiteboard.*] This looks like a binary sub-division scheme to me. Each inner node is labelled with the smallest, that is, the leftmost element of its right subtree. The leaves contain the numbers of the interval from left to right, including the left endpoint, but excluding the right endpoint.

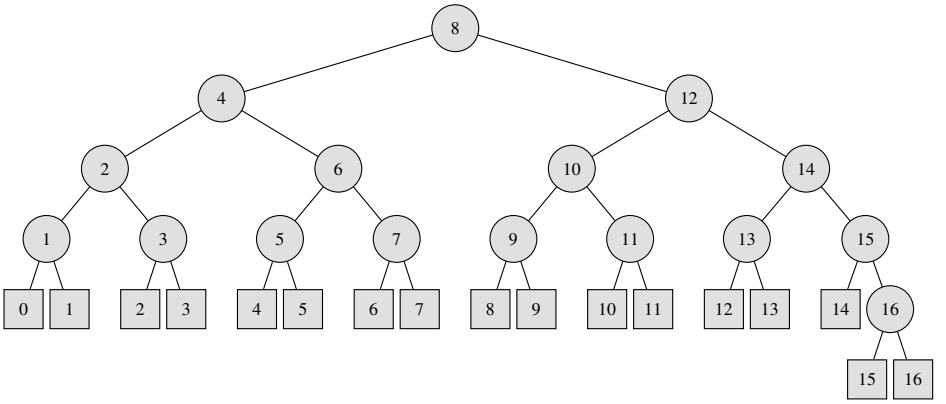


Fig. 1. The whiteboard: Harry's sketch of Ken's interval tree.

Lambert (*addressing Ken*)**:** I see, since 17 is not contained in the tree, you use the notation for right-open intervals[2]: ⟦ 0 , 17 ⟫. You certainly have taken Dijkstra's advice (1982) to heart!

Tutor (*addressing the others*)**:** Just in case you haven't read EWD831: Dijkstra makes an argument why the subsequence of natural numbers $2 , 3 , \ldots , 12$ should be written $2 \preceq i \prec 13$, ruling out $1 \prec i \preceq 12$, $2 \preceq i \preceq 12$, and $1 \prec i \prec 13$.

Harry (*who is not paying attention*)**:** I have seen this tree somewhere. [*He turns the pages of TAOCP 3.*] Here it is! Knuth introduces "comparison trees" to illustrate different strategies for searching an ordered table; this tree appears in Figure 5 (1973, Page 412) to visualise binary search. We are on the right track!

Ken: May I continue? I have tidied up the signature.

```
even-odd-pair : ⟦ l , r ⟫
              → (box : ℕ → ℕ)
```

[2] Parentheses are not allowed as parts of Agda identifiers. As a result, we opt for the less conventional ⟦ *l* , *r* ⟫ notation for half-open intervals, rather than the standard [*l* , *r*).

```
→          even (box l) × odd (box r)
→ ∃ (λ i → even (box i) × odd (box (1 + i)))
```

Since the tree constructors take explicit arguments, we no longer need to pass the endpoints explicitly—*l* and *r* are now implicit arguments. As a result, the proof is less cluttered.

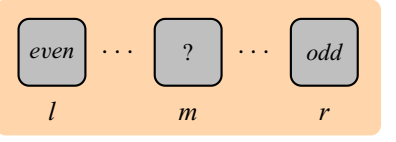
```
even-odd-pair (Leaf n)       box (e , o) = n , e , o
even-odd-pair (Node m t u) box (e , o) with parity (box m)
... | inr o′ = even-odd-pair t  box (e  , o′)
... | inl e′ = even-odd-pair u box (e′ , o )
```

A leaf signals success, whereas a node marks a decision point.

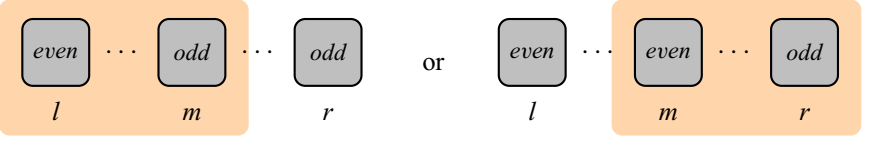
Harry: That's pretty slick.

Tutor: Indeed. As an aside, the proposition *even* (*box l*) × *odd* (*box r*) is a lovely example of a *functional invariant*. In case you have not come across this concept before, the life-cycle of an invariant can be divided into three phases:

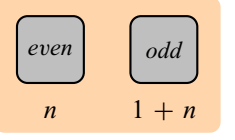
- The invariant is established (*birth*): this is the responsibility of the caller, the function that invokes *even-odd-pair*.
- The invariant is maintained (*life*): this is the responsibility of the callee. In our example, *even-odd-pair* adjusts the invariant using parity information.



The box numbered *m* with $l \prec m \prec r$ is inspected. If its contents is odd, the search continues on the left; if it is even, the search continues on the right.



- The invariant entails the desired result (*fulfilment*): again, this is the task of the callee. In our example, there is little to do as the invariant *is* the result.



The art of programming in Agda is to arrange things in such a way that each of these three tasks is as direct as possible.

Ken: In particular, the proof shows that the correctness of the search does not depend on the specific strategy we use. The strategy "only" affects the running time.

Lisa (*who has been unusually quiet*)**:** I am even more concerned now. It seems to me that *m*, the argument of *Node*, can be an arbitrary natural number. In particular, it may

lie outside the given interval. For example, if $t : [\![1 , 47]\!]$ and $u : [\![47 , 11]\!]$ then $Node\ 47\ t\ u : [\![1 , 11]\!]$!

Ken: Well, no. The expression $[\![47 , 11]\!]$ is certainly a legal type. It is, however, uninhabited: there is no element of this type. We can show that the existence of an interval tree implies that the left endpoint is below the right endpoint.

```
domain : [[ l , r ]] → l ≺ r
domain (Leaf n)      = one
domain (Node m t u)  = ≺-transitive (domain t) (domain u)
```

Lisa: I see. Putting the order-theoretic spectacles on, $Leaf\ n$ is indeed just *one*, evidence that the size of the interval $[\![n , 1 + n]\!]$ is one. And the *Node* constructor, which joins two intervals, amounts to transitivity. [*She pauses briefly.*] Since Ken's type generalises the order types, we can also show the converse of *domain*. [*Starts typing.*] Agda's code inference comes in handy.

```
sequential : l ≺ r → [[ l , r ]]
sequential (one)          = Leaf _
sequential (succ 1+l≺r)   = Node _ (Leaf _) (sequential 1+l≺r)
```

The endpoints l and r are not in scope. However, as the code is dictated by the type, I can simply use underscores for the arguments of the tree constructors. Agda will infer the correct arguments automatically. So, overall, we have demonstrated that the propositions $l \prec r$ and $[\![l , r]\!]$ are equivalent. Consequently, the corresponding implementations of *even-odd-pair* are equivalent, as well! Thanks, this puts my mind to rest.

Ken: You are welcome. I like the name *sequential* as it indicates that we are encoding the left-to-right sequential search as an interval tree. By the way, it is not more difficult to convert Harry's "midpoint trees" [*Page 5*] to interval trees.

```
binary-subdivision : l ≺′ r → [[ l , r ]]
binary-subdivision (one)    = Leaf _
binary-subdivision (t ∔ u)  = Node _ (binary-subdivision t) (binary-subdivision u)
```

We basically rename the constructors.

Harry: It is intriguing that an interval tree can be viewed in two different ways: as evidence for a relation, the strict order of the naturals, or as a data structure, fixing a search strategy.

Tutor: A nice closing, Harry. That will do for today. There is one missing puzzle piece though. We have not yet discussed how to construct *balanced* interval trees, capturing binary search—a challenging task for tomorrow!

4 Wednesday

Synopsis: The discussion gets sidetracked by a technical issue: proofs of termination in Agda.

Harry (*with a perfidious smile*)**:** Problem solved! Instead of creating balanced interval trees, I have shown that the propositions $l \prec r$ and $l \prec' r$ are equivalent. One direction is

straightforward—left as an exercise to you. The proof of the other direction constructs the desired balanced tree.

```
balanced : l ≺ r → l ≺′ r
balanced (one)     = one
balanced (succ p) = balanced (a≺a-mid-b p) ∔ balanced (a-mid-b≺b (succ p))
```

For one-point intervals there is nothing to do. If the size of the interval is greater than one, I need to show that the midpoint lies between the endpoints.

```
a≺a-mid-b : 1 + a ≺ b → a ≺ ⌊ a mid b ⌋
a-mid-b≺b :       a ≺ b → ⌊ a mid b ⌋ ≺ b
```

The proofs are pretty straightforward but not very instructive. As one would hope, *binary-subdivision* (*balanced* 0≺17) gives the tree that is still sitting on the whiteboard. [*See Figure 1.*]

Lisa: Good job Harry! So this is where we finally use properties of the midpoint. The propositions are somewhat asymmetric though. I guess this is due to the fact that we are rounding downwards?

Lambert: Yes, indeed. The floor midpoint is only strictly greater than the left endpoint if the difference of the endpoints is at least two: $1 + a \prec b$.

Lisa (*peeking over Harry's shoulder*)**:** May I? [*She takes Harry's laptop to scrutinise the Agda code.*] Harry, you have been cheating: *balanced* has a termination pragma attached to it!

Harry (*sighs*)**:** Guilty as charged. Agda's termination checker was recalcitrant. I am pretty sure, however, that the program terminates—I ported it to Haskell and tested it extensively. The Haskell compiler never complains about non-termination!

Ken: You have excessive expectations, Harry. Agda can only prove termination if the function is structurally recursive, which is not the case with your function. You may pass *p*, a sub-structure of the argument *succ p*, to a recursive invocation of *balanced* but not *a*≺*a-mid-b p* or *a-mid-b*≺*b* (*succ p*). Anyway, I would argue that a termination pragma is a cultural faux-pas in dependent type theory. It mars the entire development. Fortunately, a cure is readily at hand: we add a "recursion permit" to your function.

Tutor: Just to double-check: earlier this term, we discussed general recursion. In particular, we looked at a modular approach to *course-of-values* recursion using *accessibility* predicates. [*Looks around at blank faces.*] It seems some of you need a refresher. Ken, would you be willing to summarise the idea?

Ken: Sure. Say, we are programming a list consumer:

```
consume : List A → B
consume xs = ...
```

The function is, however, not structurally recursive; rather, it recurses over the length of the list argument *xs*. To establish termination, we add an argument of type *Accessible* (*length xs*), where *Accessible* is defined:

```
data Accessible (n : ℕ) : Set where
  permit : (∀ {m} → m ≺ n → Accessible m) → Accessible n
```

You may want to think of the additional argument as a "recursion permit": it allows us to pass *any* list to a recursive call as long as it is *shorter* than *xs*.

```
consume : (xs : List A) → Accessible (length xs) → B
consume xs (permit because) = … consume ys (because p) …
```

We only need to provide a proof, p : *length ys* $\prec$ *length xs*, that certifies that the length is actually decreasing. Agda happily accepts the definition as it is structurally recursive on the recursion permit—note that *Accessible* is an inductively defined predicate on the naturals.

Lisa: So to fix Harry's function, we add an argument of type *Accessible* $(r \dot{-} l)$, as we need to recurse over the *size* of the interval $[\![\, l\, ,\, r\,]\!]$, right? Then we have to show that the size of the interval is decreasing for both recursive calls: $m \dot{-} l \prec r \dot{-} l$ and $r \dot{-} m \prec r \dot{-} l$. [*Ponders.*]

Lambert: May I lend you a helping hand? These inequalities follow from order-theoretic properties of monus, which is monotone in its first argument and antitone in its second.

```
shrink-left  :       a ≺ b → ⌊ a mid b ⌋ ∸ a ≺ b ∸ a
shrink-right : 1 + a ≺ b → b ∸ ⌊ a mid b ⌋ ≺ b ∸ a
```

The first proposition is a consequence of $\lfloor\, a$ ***mid*** $b \,\rfloor \prec b$ and monotonicity; the second follows from $a \prec \lfloor\, a$ ***mid*** $b \,\rfloor$ and antitonicity. Well, at least in principle, the proofs are slightly more intricate as we require *strict* inequalities.

Harry: Nicely done Lambert. Surely, the standard library provides the necessary results. [*He grabs his laptop.*] Let me fix the definition of *balanced*. Ken's setup strikes me as an instance of the worker/wrapper scheme. Thanks to Lisa and Lambert, the definition of the worker function is pretty straightforward.

```
worker : l ≺ r → Accessible (r ∸ l) → l ≺′ r
worker (one)    (permit because) = one
worker (succ p) (permit because) =
    worker (a≺a-mid-b p)        (because (shrink-left (succ p)))
  ⊹ worker (a-mid-b≺b (succ p)) (because (shrink-right p))
```

[*Pauses.*] But I am unsure how to define the wrapper, the function *balanced* itself. It needs to *supply* an argument of type *Accessible* $(r \dot{-} l)$ …

Ken: I do like the worker/wrapper presentation. Yes, the worker *uses* the recursion permit, which the wrapper *provides*. Fortunately, permits can be produced in a completely generic way, independent of the application at hand.

```
accessible : ∀ {n} → Accessible n
accessible {zero}   = permit access-zero
accessible {succ n} = permit (access-succ (accessible {n}))
```

The "ticket machine" is defined by *structural induction* on the naturals. The natural number *zero* is accessible as there is no number below zero.

```
access-zero : m ≺ 0 → Accessible m
access-zero m≺0 = ex-falso-quodlibet (¬n≺zero m≺0)
```

Using the proof of $m \prec 0$ given to us, we construct a contradiction.

Lambert: Ex falso sequitur quodlibet—from contradiction, anything follows.

Ken: Your Latin is pretty good, Lambert! For the successor of a natural number,

```
access-succ : Accessible n → m ≺ 1 + n → Accessible m
access-succ (permit because) (one)                = permit because
access-succ (permit because) (succ succ-m≺succ-n) =
  because (succ-reflects-≺ succ-m≺succ-n)
```

we distinguish two cases. Either m equals n, in which case, we simply return the recursion permit for n provided as an argument. Otherwise, when m is strictly smaller than n, we *use* the recursion permit for n to show that m is accessible. As an aside, the first case demonstrates that structural recursion is a special case of course-of-values recursion—recall that *one* evidences $n \prec succ\ n$.

Harry: Great! Then we can drive the story home.

```
balanced : l ≺ r → l ≺′ r
balanced l≺r = worker l≺r accessible
```

Phew, this was rather hard work. [*He pauses to reflect on the development.*] But, do we really need the intermediate structure, my midpoint trees? Using a recursion permit, I could rewrite our very first definition of *even-odd-pair*, to directly implement a binary sub-division scheme.

Tutor: Well, you can certainly deforest midpoint trees, fusing producer and consumer—a rather mechanical optimisation step. I would argue, however, that the present design is preferable as it nicely separates concerns. The proof of *total correctness* is divided into two parts:

- A proof of *partial correctness*: *even-odd-pair* establishes the existence of an even-odd pair. The proof relies, however, on an assumption: the existence of a midpoint tree. A priori, we have no reason to believe that your type $l \prec' r$ is inhabited.
- A *termination* argument: *balanced* shows that suitable midpoint trees exist. The proof involves properties of the midpoint.

Different proof techniques are used in each part: functional invariants for partial correctness and accessibility predicates for termination. Midpoint trees serve as a connecting element.

Unfortunately, time is already up. Tomorrow, we look at further applications of binary search.

5 Thursday

Synopsis: The discussion returns to the original topic: the beauty and power of binary search.

Tutor: Good morning class. I must say I am very pleased that you have come up with a solution to the even-odd puzzle that abstracts away from the search strategy. Are there further opportunities for abstraction? For example, is there anything special about evenness and oddness? Or, could we replace the parity properties by arbitrary predicates, say, P like *petty* and Q like *quirky*?

Lisa: Well, inspecting the definition of *even-odd-pair*, it seems that we make only a single assumption: the predicates are *exhaustive*, each natural number is either even or odd. In order to generalise the proof, we could assume the existence of an *oracle* that classifies a given natural as petty or quirky.

Harry: Shouldn't we also require that the predicates P and Q are *exclusive*? In other words, Q should be the complement of P.

Lisa: Hmm, I don't see why. A number may be both petty and quirky—we leave it to the wisdom of the oracle to judge which property prevails.

Ken (*busily typing*)**:** The amendment goes through smoothly. To reduce clutter, I have passed the predicates $P\ Q\ :\ \mathbb{N} \to \mathit{Set}$ as implicit arguments.

```
interval-search : ⟦ l , r ⟧
                → (oracle : ∀ n → P n ⊎ Q n)
                →           P l × Q r
                → ∃ (λ i → P i × Q (1 + i))
interval-search (Leaf n)       oracle (p , q) = n , p , q
interval-search (Node m t u) oracle (p , q) with oracle m
... | inr q′ = interval-search t  oracle (p  , q′)
... | inl p′ = interval-search u oracle (p′ , q )
```

Thankfully, the even-odd puzzle is an instance of the general scheme.

```
even-odd-pair strategy box =
  interval-search strategy (λ n → parity (box n))
```

And now the grand finale: if we instantiate the search strategy to Harry's midpoint trees, we obtain a generic implementation of binary search!

```
binary-search : l ≺ r
              → (oracle : (n : ℕ) → P n ⊎ Q n)
              →           P l × Q r
              → ∃ (λ i → P i × Q (1 + i))
binary-search l≺r = interval-search (binary-subdivision (balanced l≺r))
```

Tutor: Excellent! Let us now discuss further applications of binary search. Do you know the game "Guess My Number"?

Someone thinks of a, say, three-digit number and the others try to guess it. After each guess, they are told whether the number is greater or less than their guess.

Lisa: Ha ha, that's a good one! [*Starts typing.*] I am not that familiar with the Agda library —how can I *decide* the ordering on the naturals?

Ken: The library offers several functions that you could use, for example, the two-way comparison $\_\preceq?\succ\_ : \forall\ m\ n \to (m \preceq n) \uplus (m \succ n)$. The operator looks perhaps a bit weird. A question mark typically indicates a decision procedure: $\_\preceq?\succ\_$ decides whether the arguments are related by $\_\preceq\_$ or by $\_\succ\_$.

Lisa: Thanks. Here we are.

$$
\begin{aligned}
&\mathit{guess\text{-}my\text{-}number} : (n : \mathbb{N}) \to n \prec 1000 \to \exists\ (\lambda\ i \to (i \preceq n) \times (n \prec 1 + i))\\
&\mathit{guess\text{-}my\text{-}number}\ n\ n{\prec}1000 =\\
&\quad \mathit{binary\text{-}search}\ 0{\prec}1000\ (\lambda\ i \to i \preceq?\succ n)\ (\mathit{zero}{\preceq}n\ n\ ,\ n{\prec}1000)
\end{aligned}
$$

Tutor: Very good. Do you remember the life-cycle of a functional invariant? The *guess-my-number* application *establishes* the invariant of *binary-search*, providing evidence that $0 \preceq n \prec 1000$ as $P\ i = (i \preceq n)$ and $Q\ i = (n \prec i)$.

Harry: Now I see why Lisa is laughing: the postcondition tells us that i is actually equal to n. Not a very useful function, but a reassuring result.

Tutor: The game was intended as a preparatory step. Let me tweak the oracle, applying a function, say, squaring to the index: $\lambda\ i \to i^2 \preceq?\succ n$.

Harry: We obtain the inverse of the function, the square root in your example?

Lambert: Not quite, squaring has no inverse in the naturals. The oracle implicitly defines the predicates: $P\ i = (i^2 \preceq n)$ and $Q\ i = (n \prec i^2)$. The postcondition of binary search, $i^2 \preceq n \prec (1 + i)^2$, then tells us that we obtain the square root *rounded downwards*.

Lisa: So we can basically copy the code for "Guess My Number".

$$
\begin{aligned}
&\mathit{square\text{-}root} : (n : \mathbb{N}) \to \exists\ (\lambda\ i \to i^2 \preceq n \times n \prec (1 + i)^2)\\
&\mathit{square\text{-}root}\ n = \mathit{binary\text{-}search}\ (\mathit{zero}{\prec}\mathit{succ\text{-}n}\ n)\ (\lambda\ i \to i^2 \preceq?\succ n)\\
&\qquad\qquad\qquad\qquad (\mathit{zero}{\preceq}n\ n\ ,\ \mathit{square\text{-}lemma}\ n)
\end{aligned}
$$

All that remains to be done is to update the code for installing the functional invariant. Since the search interval is $[\![\ 0\ ,\ 1 + n\ ]\!]$, we need to show $\mathit{square\text{-}lemma} : \forall\ n \to n \prec (1 + n)^2$.

Harry: This is certainly a valid proposition. [*Pauses.*] It seems to me that "searching an ordered table" is very similar to computing the square root. If we model the lookup table by a function $\mathit{table} : \mathbb{N} \to \mathbb{N}$ that maps positions to keys, then the oracle reads $\lambda\ i \to \mathit{table}\ i \preceq?\succ \mathit{key}$.

$$
\begin{aligned}
&\mathit{table\text{-}lookup} : l \prec r\\
&\qquad\qquad \to (\mathit{table} : \mathbb{N} \to \mathbb{N})\\
&\qquad\qquad \to (\mathit{key} : \mathbb{N})\\
&\qquad\qquad \to \qquad \mathit{table}\ l \preceq \mathit{key} \times \mathit{key} \prec \mathit{table}\ r\\
&\qquad\qquad \to \exists\ (\lambda\ i \to \mathit{table}\ i \preceq \mathit{key} \times \mathit{key} \prec \mathit{table}\ (1 + i))\\
&\mathit{table\text{-}lookup}\ l{\prec}r\ \mathit{table}\ \mathit{key} = \mathit{binary\text{-}search}\ l{\prec}r\ (\lambda\ i \to \mathit{table}\ i \preceq?\succ \mathit{key})
\end{aligned}
$$

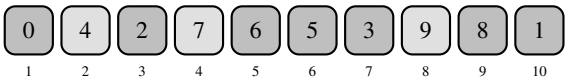
The difference to Knuth's setup is striking: his search returns an index i such that $K_i \equiv K$, but the search may fail; our search always succeeds, returning an index i such that $K_i \preceq K \prec K_{1+i}$.

Lisa: There is one further difference: Knuth assumes that the table is ordered, that the function *table* is monotone; we don't make this assumption.

Harry: Yes, you are right. “Searching an unordered table”? Weird!

Tutor: I see, you are getting into the mood. Ready for one of my favourite puzzles, taken from Jeff Erickson’s excellent lecture notes on algorithms (2011)?

You are a contestant on the game show “Beat Your Neighbours!”. You are presented with a sequence of n boxes, each containing a natural number. It costs 100€ to open a box. Your goal is to find a box whose number is larger than its left and right neighbours. If you spend less money than any of your opponents, you win a week-long trip to Berlin.

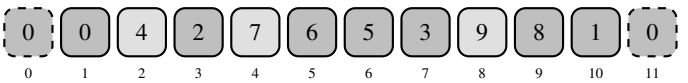


Harry: Does such a box always exist? In your example, the boxes numbered 2, 4, and 8 are winning boxes. But, what if all boxes contain the same number, or the numbers are strictly increasing from left to right?

Lisa: Perhaps we should first clarify what “larger” means: “strictly larger” or “no smaller”? Furthermore, the description is incomplete: the leftmost box has no left neighbour, and the rightmost box has no right neighbour.

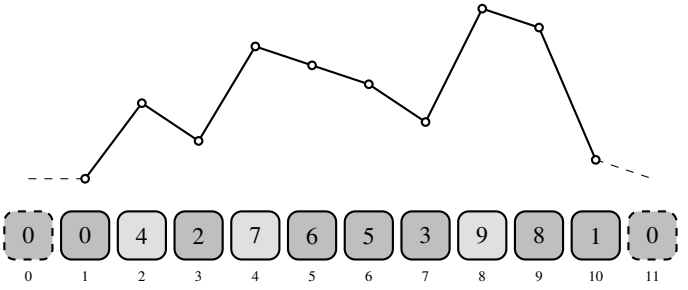
Tutor: Well, the boxes at the ends need to beat only one neighbour. Regarding the reading of “larger”: to avoid upsetting the candidates, we should guarantee the existence of a winning box, so “larger” means “no smaller”.

Lambert: Treating the outermost boxes separately, strikes me as inelegant. To avoid three different winning criteria, I suggest to add “virtual” boxes to the ends. [*He adds “sentinels” to the example.*]



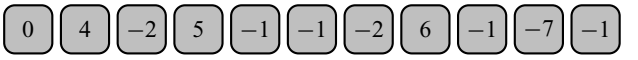
They are similar to “sentinel nodes”, which serve as traversal terminators in imperative programming. The virtual boxes contain the smallest natural number, so that the “beats” condition is trivially satisfied.

Harry: I don’t think that this puzzle is an instance of binary search. Say, you open the middle box and 6 is revealed to you. What conclusions can you draw? None! Perhaps a picture is helpful. [*He plots the graph of the function.*]



[*Mumbles.*] Looks like a mountain range to me.

Lisa: Harry Hacker, you are a genius. Imagine you are hiking in the mountains, aiming to climb *some* mountain peak. Which strategy do you follow? You consider the slope, not the height above sea level. [*Adds a line of numbers.*]



If we replace height by slope, $slope := (box\ (1 + i) - box\ i)\ /\ ((1 + i) - i)$, then we have a straightforward application of binary search: $P\ i$ holds if $slope\ i$ is non-negative, $Q\ i$ holds if $slope\ i$ is non-positive.

Lambert: So in abstract clothing, we are looking for a *local* maximum.

Ken (*again eagerly typing*)**:** The code is straightforward, indeed. I don't use *slope* though. Instead, I simply compare the contents of two adjacent boxes.

$$
\begin{aligned}
&beat\text{-}your\text{-}neighbours\ :\ (box\ :\ \mathbb{N} \to \mathbb{N}) \\
&\qquad\qquad\qquad\qquad \to l \prec r \\
&\qquad\qquad\qquad\qquad \to \qquad\quad box\ l \preceq box\ (1 + l) \times box\ r \succeq box\ (1 + r) \\
&\qquad\qquad\qquad\qquad \to \exists\ (\lambda\ i \to box\ i \preceq box\ (1 + i) \times box\ (1 + i) \succeq box\ (2 + i)) \\
&beat\text{-}your\text{-}neighbours\ box\ l{\prec}r\ = \\
&\quad binary\text{-}search\ l{\prec}r\ (\lambda\ i \to box\ i \preceq ?\succeq box\ (1 + i))
\end{aligned}
$$

The code assumes Lambert's sentinel trick as the winning criterion does not treat the boxes at the ends differently. Just in case you wonder, $\_\preceq ?\succeq\_ : \forall\ m\ n \to m \preceq n \uplus m \succeq n$ is a "non-deterministic" variant of $\_\preceq ?\succ\_$. If the arguments are equal, the operator either returns a proof of $m \preceq n$ or a proof of $m \succeq n$. We don't know which, but also we don't care.

Harry: So this is an example where P and Q are not exclusive. [*Pauses.*] On a more general note, I have to admit that I wasn't very thrilled by this week's topic, but binary search is much more general than I thought initially. In particular, I am intrigued that we do not need to assume that the data is ordered.

Ken: I like to think that we experience the effect of constructive logic. Both Knuth and Bird&Wadler are somewhat negative: in the recursive step, they *eliminate* data from consideration. In the end, their search may fail. We are much more positive: in the recursive step, we continue with the half that *guarantees* the existence of a petty-quirky pair—the other half possibly contains further pairs. Our search is always successful. Constructive logic favours a positive mindset.

Tutor: Sorry, we are already running overtime. Let's continue the discussion tomorrow. If you feel energetic, try to re-implement Bird&Wadler in Agda.

6 Friday

Synopsis: The rôle of monotonicity is discussed.

Tutor: Good morning class. Lisa, do you want to take the lead?

Lisa: Sure. A quick refresher is probably not amiss. Given a monotone predicate $P : \mathbb{N} \to Set$ and an interval $[\ l\ ,\ r\ ]$, Bird&Wadler seek to determine the *smallest*

index $j \in [\ l\ ,\ r\ ]$ such that $P\ j$ holds. In other words, they are looking for a position where the predicate P flips. This is no different than a petty-quirky pair, where i is petty if $\neg\ P\ i$ and i is quirky if $P\ i$. Now, approaching the task with a positive mindset, we would, of course, like to *guarantee* the existence of such a pair. These considerations motivate the following interface.

$$
\begin{aligned}
&\mathit{find}\ :\ l \prec r \\
&\quad \to (\mathit{monotone}\ :\ \forall\ \{a\ b\} \to a \preceq b \to (P\ a \to P\ b)) \\
&\quad \to (\mathit{decide}\ :\ \forall\ n \to \neg\ P\ n \uplus P\ n) \\
&\quad \to \neg\ P\ l \times P\ r \\
&\quad \to \exists\ (\lambda\ j \to P\ j \times (\forall\ i \to i \prec j \to \neg\ P\ i))
\end{aligned}
$$

The preconditions $l \prec r$ and $\neg\ P\ l \times P\ r$ make sure that a petty-quirky pair exists. Since P is monotone the pair is also unique. The postcondition rewords this observation to say that we have found the *smallest* index j such that $P\ j$ holds.

Harry: Are you suggesting that monotonicity plays no rôle for the actual search?

Lisa: Exactly! In particular, we can implement Bird&Wadler's search function *find* in terms of our *binary-search*.

$$
\begin{aligned}
&\mathit{find}\ l{\prec}r\ \mathit{monotone}\ \mathit{decide}\ \mathit{pre}\ \mathbf{\mathit{with}}\ \mathit{binary\text{-}search}\ l{\prec}r\ \mathit{decide}\ \mathit{pre} \\
&\ldots\ \mid j\ ,\ \neg P\text{-}j\ ,\ P\text{-}1{+}j \\
&\quad =\ 1 + j\ ,\ P\text{-}1{+}j\ ,\ \lambda\ i\ (i{\prec}1{+}j\ :\ i \prec 1 + j)\ P\text{-}i \to \\
&\qquad\qquad \neg P\text{-}j\ (\mathit{monotone}\ (\prec\text{-}\mathit{succ\text{-}is}\text{-}\preceq\ i{\prec}1{+}j)\ P\text{-}i)
\end{aligned}
$$

We obtain a petty-quirky pair, so $1 + j$ is the desired smallest quirky index. For the proof of minimality, we basically apply the contrapositive of monotonicity: $a \preceq b \to (\neg\ P\ b \to \neg\ P\ a)$. Concretely, from $i \prec 1 + j$, which is equivalent to $i \preceq j$ and $\neg\ P\ j$ we conclude $\neg\ P\ i$.

Lambert: It is interesting to see how things intertwine: the invariant guarantees the existence of a "quirky" index; monotonicity ensures that it is unique. I am really pleased to see that Lisa's program is total in contrast to Bird&Wadler's code. Their "Introduction to Functional Programming" is an excellent textbook, but the presentation of binary search is, well, slightly below standard.

Harry: In their defence, we place an additional burden on the caller of *find* who needs to install the functional invariant $\neg\ P\ l \times P\ r$.

Lisa: Well, yes and no. Yes, they have to provide evidence that the search is in some sense "promising". One option is to apply Lambert's cute sentinel trick: assuming l is non-zero, the original search interval $[\ l\ ,\ r\ ]$ is widened to $l - 1 \prec 1 + r$, setting $P\ i := \bot$ for $i \prec l$ and $P\ i := \top$ for $r \prec i$. The conditions now hold by definition, so no, I don't consider this a burden.

Harry: Hmm, okay. [*Pauses.*] So monotonicity is only used in an afterthought. [*Pauses again.*] Can we please revisit the problem of "searching an ordered table"? I really don't see how to define this as an instance of our binary search. Approaching problems with a positive mindset is all good and well, but we simply cannot guarantee that a given table contains a given key.

Ken: True. But you can use monotonicity to *decide* containment. You may want to implement the following interface:

```
ord-table-lookup : l ≺ r
                 → (table : ℕ → ℕ)
                 → (monotone : ∀ {a b} → a ≼ b → table a ≼ table b)
                 → (key : ℕ)
                 → table l ≼ key × key ≺ table r
                 → Decide (∃ (λ i → table i ≡ key))
```

A property is decidable if you either have concrete evidence that it holds or evidence that it does not hold.

```
data Decide (Prop : Set) : Set where
  yes :   Prop → Decide Prop
  no  : ¬ Prop → Decide Prop
```

Harry: I guess you are suggesting to use yesterday's *table-lookup*, post-processing its results? [*Starts typing.*]

```
ord-table-lookup l≺r table monotone key pre
  with table-lookup l≺r table key pre
... | i , inl p_i , q_i = yes (i , p_i)
... | i , inr p_i , q_i = no  key-not-found
  where key-not-found : ¬ ∃ (λ j → table j ≡ key)
```

The function *table-lookup* returns proofs that $K_i \preccurlyeq K \prec K_{1+i}$. I pattern match on the first proof to check whether $K_i \equiv K$ or $K_i \prec K$—recall that we defined $m \preccurlyeq n = (m \equiv n) \uplus (m \prec n)$. If the former, then I signal success; if the latter, I need to report that the key has not been found. [*Stops typing.*]

Ken: May I offer a helping hand? In the negative case, you can derive a contradiction using irreflexivity. Say, the adversary claims the existence of an index j with $K_j \equiv K$. Assuming $j \preccurlyeq i$, monotonicity implies $K_j \preccurlyeq K_i$. Since we already know that $K_i \prec K \equiv K_j$, witnessed by p_i, we have $K_i \prec K_i$, which is impossible.

```
key-not-found (j , reflexive) with j ≼?≻ i
... | inl j≼i = ≺-irreflexive (≺-then-≼ p_i (monotone j≼i))
... | inr j≻i = ≺-irreflexive (≺-then-≼ q_i (monotone (≺-is-succ-≼ j≻i)))
```

The other case enjoys a similar proof.

Harry: Cool. [*Smiles mischievously.*] I never thought that monotonicity would help with negative feelings.

Lisa: In the success case, monotonicity is also helpful: it implies that the index i lies in the specified interval: $l \preccurlyeq i \prec r$.

Tutor: Well done everybody. We certainly have earned an early weekend. I hope you enjoyed these tutorials as much as I did. As usual, the exercise sheet contains some ideas for further exploration, revisiting design decisions and generalising the setup. [*See Figure 2.*] Enjoy! Oh, and I recommend reading Joshua Bloch's blog post (2006)

Exercise 1 (warm-up). Show the analogue of *domain* for mid-point trees:

$$domain' \;:\; l \prec' r \to l \prec r$$

This is Harry's "straightforward" exercise from Wednesday. □

The predicates passed to interval search range over the naturals, $P\ Q\ :\ \mathbb{N} \to Set$. For applications that involve finite look-up tables it is prudent to restrict the domain to the given search interval $[\![\ l\ ,\ r\ ]\!]$:

$$P\ Q\ :\ (n\ :\ \mathbb{N}) \to l \preceq n \times n \preceq r \to Set$$

However, ordering constraints such as $l \preceq n$ are often cumbersome to work with. An attractive, implementation-oriented alternative builds on membership.

$$\begin{aligned}
&\mathbf{data}\ _\in_\ (i\ :\ \mathbb{N})\ :\ [\![\ l\ ,\ r\ ]\!] \to Set\ \mathbf{where}\\
&\quad here\ \ :\ i \in Leaf\ i\\
&\quad left\ \ :\ \{t\ :\ [\![\ l\ ,\ m\ ]\!]\} \to \{u\ :\ [\![\ m\ ,\ r\ ]\!]\} \to i \in t\ \to i \in Node\ m\ t\ u\\
&\quad right\ :\ \{t\ :\ [\![\ l\ ,\ m\ ]\!]\} \to \{u\ :\ [\![\ m\ ,\ r\ ]\!]\} \to i \in u \to i \in Node\ m\ t\ u
\end{aligned}$$

If $\mathrm{I}\ :\ [\![\ l\ ,\ r\ ]\!]$ is an interval tree, $n \in \mathrm{I}$ is a path from the root of the tree to an n-labelled leaf. A technicality: since the tree represents only a half-open interval, we also need a variant of membership that includes the right endpoint (as Q is applied to r).

$$\begin{aligned}
&\mathbf{data}\ _\dot\in_\ (i\ :\ \mathbb{N})\ :\ [\![\ l\ ,\ r\ ]\!] \to Set\ \mathbf{where}\\
&\quad half\text{-}open\ :\ \{\mathrm{I}\ :\ [\![\ l\ ,\ r\ ]\!]\} \to i \in \mathrm{I} \to i \mathrel{\dot\in} \mathrm{I}\\
&\quad rightmost\ :\ \{\mathrm{I}\ :\ [\![\ l\ ,\ i\ ]\!]\} \to i \mathrel{\dot\in} \mathrm{I}
\end{aligned}$$

Exercise 2. Given an interval tree $\mathrm{I}\ :\ [\![\ l\ ,\ r\ ]\!]$, prove that membership $n \in \mathrm{I}$ is equivalent to $l \preceq n \times n \prec r$ and $n \mathrel{\dot\in} \mathrm{I}$ to $l \preceq n \times n \preceq r$. In particular, show that the endpoints are included in the closed interval:

$$\begin{aligned}
&\swarrow_\ :\ (\mathrm{I}\ :\ [\![\ l\ ,\ r\ ]\!]) \to l \mathrel{\dot\in} \mathrm{I}\\
&_\searrow\ :\ (\mathrm{I}\ :\ [\![\ l\ ,\ r\ ]\!]) \to r \mathrel{\dot\in} \mathrm{I}
\end{aligned}$$

□

Exercise 3. Implement the following "finite" variant of *interval-search*, which guarantees that the oracle is not called outside the given range.

$$\begin{aligned}
interval\text{-}search^F\ \ :\ &(\mathrm{I}\ :\ [\![\ l\ ,\ r\ ]\!])\\
&\to \{P\ Q\ :\ (n\ :\ \mathbb{N}) \to n \mathrel{\dot\in} \mathrm{I} \to Set\}\\
&\to (oracle\ :\ (n\ :\ \mathbb{N}) \to (p\ :\ n \mathrel{\dot\in} \mathrm{I}) \to P\ n\ p \uplus Q\ n\ p)\\
&\to P\ l\ (\swarrow \mathrm{I}) \times Q\ r\ (\mathrm{I} \searrow)\\
&\to \exists\ (\lambda\ i \to \exists\ (\lambda\ p \to \exists\ (\lambda\ q \to P\ i\ p \times Q\ (1 + i)\ q)))
\end{aligned}$$

Specialise the generic search to finite look-up tables. □

Exercise 4. For simplicity, we have assumed that the endpoints of an interval are natural numbers. Generalise *interval-search* so that it works on the integers $\mathbb{Z}$ As a first step, change the interval datatype [_,_[to include integer endpoints. □

Fig. 2. Exercise sheet: binary search and its applications.

"Extra, Extra—Read All About It: Nearly All Binary Searches and Mergesorts Are Broken", which shows that formal proofs do not eliminate the need for systematic testing—typical programming languages use fixed-precision arithmetic, which can lead to overflow errors when computing the midpoint. Our Peano naturals support arbitrary-precision arithmetic, so we avoid this problem.

7 Afterword

This pearl grew out of the authors' frustration with textbook presentations of binary search. Given that binary search is one of the standard algorithms every programmer and computer science student should know, the subject is inadequately covered at best. Many textbooks mention binary search only in passing or they treat only special cases, such as computing the square root or searching an ordered table. A negative mindset prevails: the search possibly fails; in the divide&conquer step one half of the search space is excluded from consideration because searching this half will definitely fail. The correctness argument requires that the data is ordered, suggesting that monotonicity in some sense drives the search. One notable exception is Anne Kaldewaij's textbook (1990): when discussing "function inversion" (given n, find an argument i such that $f\,i \preceq n \prec f\,(1 + i)$) he emphasises repeatedly that the correctness of the search does not require that f is an ascending function.

The gist of this pearl is to approach search problems with a positive mind-set: the search always succeeds; in the divide&conquer step the search continues with the half that guarantees success. The correctness argument relies on a suitable functional invariant, not on monotonicity. The "Beat Your Neighbours!" problem, a concrete make-up for local maximum search, shows that the extra generality is actually needed.

Supplementary materials

For supplementary material for this article, please visit https://doi.org/10.1017/S0956796825000061

Acknowledgements

Thanks are due to Clare Martin, Lambert Meertens, Wouter Swierstra, and Nicolas Wu for carefully proofreading an earlier version of this pearl. Clare's comments made the Agda code more accessible. Lambert kindly allowed us to use his name for one of the characters. Nick's and Lambert's suggestions helped to improve style and grammar. Wouter's proposals improved precision and clarity. We are furthermore grateful to the anonymous reviewers for their detailed comments, especially on the storyline and take-home messages. Finally, a big thank you goes to Norman Ramsey for his editorial advice.

Conflicts of Interest

None.

References

Bird, R. & Wadler, P. (1988) *Introduction to Functional Programming*. Series in Computer Science. Prentice-Hall.
Bloch, J. (2006) Extra, extra — read all about it: Nearly all binary searches and mergesorts are broken.
Dijkstra, E. W. (1982) Why numbering should start at zero. circulated privately.
Erickson, J. (2011) Algorithms.
Kaldewaij, A. (1990) *Programming: The Derivation of Algorithms*. Prentice-Hall International Series in Computer Science. Prentice Hall.
Knuth, D. E. (1973) *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley.

A Appendix

The complete Agda code can be found in the accompanying material. This appendix lists some basic types and functions.

Constructive Logic. The disjoint union of two types is defined as follows:

```
data _⊎_ (A : Set) (B : Set) : Set where
  inl : A → A ⊎ B
  inr : B → A ⊎ B
```

An element of type $A \uplus B$ is either of the form *inl a* for some $a : A$ or of the form *inr b* for some $b : B$. In the "propositions as types" paradigm, the disjoint union corresponds to *logical or*.

In Agda, there is an empty type, denoted $\bot$, which corresponds to falsity. The empty type is quite special in that there is a unique function from $\bot$ to any type. This unique function is defined using the *absurd pattern* "()".

```
ex-falso-quodlibet : ∀ {A : Set} → ⊥ → A
ex-falso-quodlibet ()
```

The function implements the logical rule "ex falso (sequitur) quodlibet", hence the name.

Negation is a special case of implication: $\neg A := A \to \bot$ expresses that it is absurd to assume A. As $\bot \to A$ holds trivially, this means that A is, in fact, equivalent to the empty type, that A is unprovable and hence false.

Natural numbers. Following Peano, we define the natural numbers inductively as a type with two constructors: *zero* represents the natural number 0 and *succ* takes a natural number and returns its successor.

```
data ℕ : Set where
  zero : ℕ
  succ : ℕ → ℕ
```

In general, for a given natural number n, the partial application $\_+ n$ lacks an inverse. However, we can introduce a subtraction operation, denoted $\dot{-}$ and referred to as *monus*,

such that $\_\dot{-}\ n$ acts as the inverse on naturals greater than n. When m is less than n, the difference $m \dot{-} n$ is defined to be 0.

$$
\begin{array}{lll}
\_\dot{-}\_ & : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
\mathit{zero} \ \dot{-}\ n & = \mathit{zero} \\
\mathit{succ}\ m \ \dot{-}\ \mathit{zero} & = \mathit{succ}\ m \\
\mathit{succ}\ m \ \dot{-}\ \mathit{succ}\ n & = m \ \dot{-}\ n
\end{array}
$$

The "lower" midpoint of two natural numbers a and b is given by $\lfloor\ (a + b)\ /2 \rfloor$. For purely pragmatic reasons, we unfold the definition of $\_+\_$, obtaining

$$
\begin{array}{lll}
\lfloor \_\textbf{\textit{mid}}\_ \rfloor & : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
\lfloor\ \mathit{zero} \quad \textbf{\textit{mid}}\ b \quad\ \rfloor & = \lfloor\ b\ /2 \rfloor \\
\lfloor\ \mathit{succ}\ a\ \textbf{\textit{mid}}\ \mathit{zero} \quad \rfloor & = \lfloor\ \mathit{succ}\ a\ /2 \rfloor \\
\lfloor\ \mathit{succ}\ a\ \textbf{\textit{mid}}\ \mathit{succ}\ b\ \rfloor & = \mathit{succ}\ \lfloor\ a\ \textbf{\textit{mid}}\ b\ \rfloor
\end{array}
$$

The function $\lfloor \_/2 \rfloor : \mathbb{N} \to \mathbb{N}$ divides a natural by 2, rounding down.

Standard order. The following properties of $\_\prec\_$ and $\_\preceq\_$ are used in the main body of the article. The proofs are not too revealing and therefore omitted:

$$
\begin{array}{ll}
\prec\textit{-irreflexive} & : \neg\ n \prec n \\
\prec\textit{-transitive} & : k \prec m \to m \prec n \to k \prec n \\
\mathit{zero}{\prec}\mathit{succ\text{-}n} & : \forall\ n \to \mathit{zero} \prec \mathit{succ}\ n \\
\neg n{\prec}\mathit{zero} & : \neg\ n \prec \mathit{zero} \\
\mathit{succ\text{-}reflects\text{-}}\prec & : \mathit{succ}\ m \prec \mathit{succ}\ n \to m \prec n \\
\mathit{zero}{\preceq}n & : \forall\ n \to \mathit{zero} \preceq n
\end{array}
$$