28. Knobelaufgabe #13

Lisa Lista hat sich folgende Methode ausgedacht, um schrittweise einen Binärbaum zu konstruieren.

```
let rec push (n : 'a, t : Tree ⟨'a⟩) : Tree ⟨'a⟩ =
  match t with
  | Leaf           → Node (Leaf, n, Leaf)
  | Node (l, a, r) → Node (push (n, r), a, l)
```

☞ Um zu verhindern, dass der Baum zu einem Strunk entartet, fügt die Funktion stets rechts ein, vertauscht aber in jedem Schritt den linken und den rechten Teilbaum.

Helfen Sie Lisa bei der Analyse: Welche Form hat der Baum, wenn nacheinander die Zahlen 1 bis n eingefügt werden: $push\ (1, push\ (2, \ldots (push\ (n, Leaf))))$? Wie sieht der Baum für den Fall $n = 2^k - 1$ aus?

28. Suchen

Mini Softwareprojekt: Verwaltung von Personaldaten eines Unternehmens.

Der Personalstamm wird durch eine Liste von Einträgen der Typs *Entry* repräsentiert.

type $Entry = \{ key : Nat; person : Person \}$

☞ Jeder Eintrag besteht aus einer *eindeutigen* Personalnummer und den eigentlichen Personendaten.

$$\begin{aligned} \textbf{\textit{let}}\ team = [&\{ key = 7; \quad person = ralf \quad \}; \\ &\{ key = 815; \ person = melanie \}; \\ &\{ key = 4711; person = julia \quad \}; \\ &\{ key = 4712; person = andres \ \}] \end{aligned}$$

28. Suchen in einer Liste

Wiederkehrende Aufgabe: zu einer gegebenen Personalnummer die zugehörigen Personendaten heraussuchen.

look-up (*key* : *Nat*, *staff* : *List* ⟨*Entry*⟩) : *Person*

☞ Was machen wir, wenn kein passender Eintrag existiert?

Idee: die beiden möglichen Resultate einer Suche, erfolglos und erfolgreich, mit einem Datentyp darstellen. *Zur Erinnerung:*

type *Option* ⟨*'a*⟩ = | *None*
| *Some* ***of*** *'a*

Damit können wir die Signatur von *look-up* verfeinern.

look-up (*key* : *Nat*, *staff* : *List* ⟨*Entry*⟩) : *Option* ⟨*Person*⟩

☞ Schlägt die Suche fehl, wird *None* zurückgegeben, sonst *Some p*, wobei *p* die gesuchte Person ist.

28. Suchen in einer Liste

Mit dem Struktur Entwurfsmuster für *List* erhalten wir:

```
let rec look-up (key : Nat, staff : List ⟨Entry⟩) : Option ⟨Person⟩ =
  match staff with
  | []              → ...
  | entry :: entries → ... look-up (key, entries) ...
```

28. Suchen in einer Liste

- *Rekursionsbasis*: die Suche schlägt fehl.

```
let rec look-up (key : Nat, staff : List ⟨Entry⟩) : Option ⟨Person⟩ =
  match staff with
  | []                  → None
  | entry :: entries →  . . . look-up (key, entries) . . .
```

- *Rekursionsschritt*: Ist das Kopfelement die gesuchte Person? Anderenfalls weitersuchen.

```
let rec look-up (key : Nat, staff : List ⟨Entry⟩) : Option ⟨Person⟩ =
  match staff with
  | []                  → None
  | entry :: entries →
    if key = entry.key then Some entry.person
                       else look-up (key, entries)
```

28. Suchen in einer geordneten Liste

Die (erfolglose) Suche lässt sich etwas beschleunigen, wenn wir annehmen, dass die Liste nach der Personalnummer geordnet ist.

Mit dem Struktur Entwurfsmuster für *List* erhalten wir:

```
let rec look-up (key : Nat, staff : List ⟨Entry⟩) : Option ⟨Person⟩ =
  match staff with
  | []              → ...
  | entry :: entries → ... look-up (key, entries) ...
```

Vorbedingung: staff ist nach der Personalnummer geordnet. Das Typsystem stellt diese Eigenschaft nicht sicher, darum müssen wir uns kümmern.

28. Suchen in einer geordneten Liste

- *Rekursionsbasis*: die Suche schlägt fehl.

```
let rec look-up (key : Nat, staff : List ⟨Entry⟩) : Option ⟨Person⟩ =
  match staff with
  | []                  → None
  | entry :: entries →  . . . look-up (key, entries) . . .
```

- *Rekursionsschritt*: aus dem 2-Wege wird ein 3-Wege Vergleich.

```
let rec look-up (key : Nat, staff : List ⟨Entry⟩) : Option ⟨Person⟩ =
  match staff with
  | []                  → None
  | entry :: entries →
    if    key < entry.key    then None
    elif  key = entry.key    then Some entry.person
    (* key > entry.key *)    else  look-up (key, entries)
```

☞ Die erfolgreiche Suche wird nicht beschleunigt; im Durchschnitt werden genauso viele rekursive Aufrufe benötigt.

28. Binärbäume

☞ Die Implementierung von *look-up* erinnert an die lineare Suche aus Teil III. Können wir die binäre Suche adaptieren?
Nein, nicht ohne den Geschwindigkeitsvorteil zu verlieren: im Gegensatz zur Halbierung des Suchintervalls ist die Halbierung einer Liste aufwändig (wie aufwändig?).

Wenn wir schnell auf das mittlere Element zugreifen wollen, brauchen wir einen anderen Containertyp.

```
type Tree ⟨'a⟩ =
    | Leaf
    | Node of Tree ⟨'a⟩ * 'a * Tree ⟨'a⟩
```

- *Leaf* tritt an die Stelle von *Nil*;
- *Node* (l, x, r) tritt an die Stelle von *Cons* (x, xs) und repräsentiert eine mindestens einelementige Folge, bestehend aus der „linken“ Teilfolge l, dem Element x und der „rechten“ Teilfolge r.

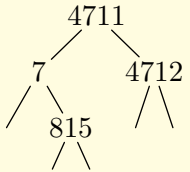
28. Binärbäume

Die Folge $7\ 815\ 4711\ 4712$ kann zum Beispiel durch

$$\begin{aligned}&\mathit{Node}\ (\mathit{Node}\ (\mathit{Leaf}, 7, \mathit{Node}\ (\mathit{Leaf}, 815, \mathit{Leaf})), 4711,\\ &\qquad \mathit{Node}\ (\mathit{Leaf}, 4712, \mathit{Leaf}))\end{aligned}$$

repräsentiert werden (es gibt noch 13 andere Möglichkeiten).

Die Elemente von *Tree* heißen auch *Binärbäume*; Baum wegen der hierarchischen Struktur; *binär*, da jeder nicht-leere Baum in zwei Teilbäume verzweigt.



☞ Sind die Elemente von links nach rechts geordnet (siehe oben), so spricht man weiterhin von einem *Suchbaum*.

28. Suchen in einem Binärbaum

Mit dem Struktur Entwurfsmuster für *Tree* erhalten wir:

```
let rec look-up (key : Nat, staff : Tree ⟨Entry⟩) : Option ⟨Person⟩ =
  match staff with
    | Leaf                    → ...
    | Node (left, entry, right) →
      ... look-up (key, left) ... look-up (key, right) ...
```

☞ Im Rekursionsschritt dürfen wir die Teillösungen für den linken *und* den rechten Teilbaum verwenden.

28. Suchen in einem Binärbaum

- *Rekursionsbasis*: die Suche schlägt fehl.

```
let rec look-up (key : Nat, staff : Tree ⟨Entry⟩) : Option ⟨Person⟩ =
  match staff with
  | Leaf                       → None
  | Node (left, entry, right) →
    ... look-up (key, left) ... look-up (key, right) ...
```

- *Rekursionsschritt*: der 3-Wege Vergleich bleibt erhalten.

```
let rec look-up (key : Nat, staff : Tree ⟨Entry⟩) : Option ⟨Person⟩ =
  match staff with
  | Leaf                       → None
  | Node (left, entry, right) →
    if    key < entry.key    then look-up (key, left)
    elif  key = entry.key    then Some entry.person
    (* key > entry.key *)    else look-up (key, right)
```

28. Suchen in einem Binärbaum

Welche Laufzeit hat die neue Version von *look-up*?

☞ Das kommt auf die Form des Suchbaums an.

- Ein Binärbaum heißt *ausgeglichen* oder *balanciert*, wenn die Elemente links und rechts jeweils „gleichmäßig“ verteilt sind.
- Ein Binärbaum heißt *degeneriert*, wenn einer der Teilbäume jeweils leer ist (der Baum entspricht einer Liste).

Laufzeit von *look-up*:

- ausgeglichener Baum: logarithmische Laufzeit,
- degenerierter Baum: lineare Laufzeit.

Die Suche ist linear zur *Höhe* des Suchbaums.

28. Schlag die Nachbarn!

- *Aufgabe:*

  Sie sind in der populären Spielshow „Schlag die Nachbarn!“ ins Finale gekommen und müssen die letzte Aufgabe meistern. Ihnen wird eine nicht-leere Folge von Schachteln präsentiert, die jeweils eine für Sie nicht sichtbare Zahl enthalten. Sie müssen eine Schachtel finden, deren Zahl größer ist als die ihrer Nachbarn. Eine Schachtel zu öffnen kostet 100€. Wenn Sie weniger Geld als Ihre Konkurrent*innen ausgeben, gewinnen Sie das Finale!

- Zum Beispiel:

0	1	2	3	4	5	6	7	8	9
0	4	2	7	6	5	3	9	8	1

- Schachteln #1, #3, and #7 schlagen ihre Nachbarn.
- Schachtel #7 enthält die größte Zahl.

Gibt es denn tatsächlich immer eine Schachtel, die ihre Nachbarn schlägt?

Nö — alle Schachteln enthalten die gleiche Zahl.

0	0	0	0	0	0	0	0	0	0

Vielleicht ist mit „größer“ tatsächlich „größer gleich“ gemeint?

Na ja, selbst dann nicht.

0	1	2	3	4	5	6	7	8	9

Du meinst, die letzte Schachtel zählt nicht, weil sie keinen rechten Nachbarn hat?

Genau ;-).

28. Schlag die Nachbarn: Spielregeln

- Die mittlere Schachtel schlägt ihre Nachbarn, wenn

$\cdots$	l	a	r	$\cdots$

$$l \leqslant a \geqslant r$$

- Wir nehmen an, dass es am linken und am rechten Rand jeweils eine „virtuelle“ Schachtel gibt, die $-\infty$ enthält.

-1	0	1	2	3	4	5	6	7	8	9	10
$-\infty$	0	4	2	7	6	5	3	9	8	1	$-\infty$

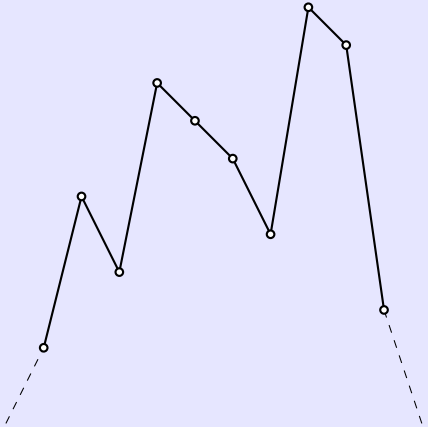
- (*Trick:* so lassen sich Sonderfälle vermeiden. Wenn wir später die Korrektheit beweisen, lassen wir -1 als Hausnummer zu, obwohl -1 keine natürliche Zahl ist.)

28. Schlag die Nachbarn: lokale Maxima

- *Einsicht*: wir suchen *ein* lokales Maximum.

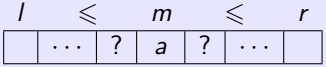
-1	0	1	2	3	4	5	6	7	8	9	10
$-\infty$	0	4	2	7	6	5	3	9	8	1	$-\infty$

- Insgesamt gibt es drei lokale Maxima (und zwei lokale Minima).

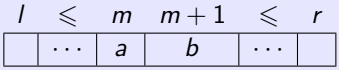


28. Schlag die Nachbarn: Lösungsideen

- *Naive Lösung:* wir öffnen alle Schachteln (globales Maximum).
- Wir suchen aber lediglich ein *lokales* Maximum.
- *Idee:* binäre Suche?

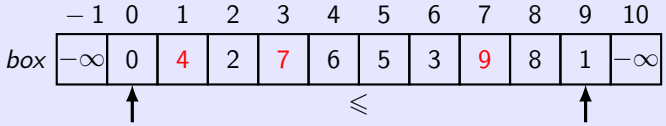


- ...nicht sehr informativ.
- *Idee:* wir öffnen zwei benachbarte Schachteln, zum Beispiel die beiden mittleren Schachteln:

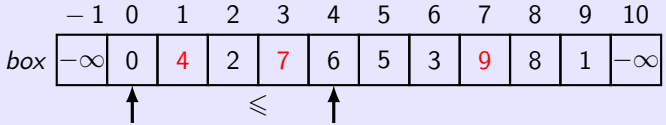


  - wenn $a \leqslant b$, setzen wir die Suche im Intervall $m+1 \ldots r$ fort;
  - wenn $a \geqslant b$, setzen wir die Suche im Intervall $l \ldots m$ fort.
- Finden wir so tasächlich ein lokales Maximum?

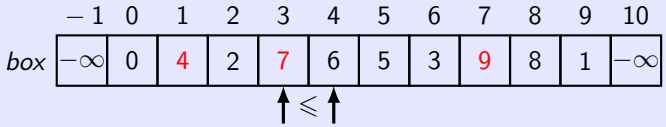
28. Schlag die Nachbarn: Beispiel



- *search* $(0, 9)$: *box* $4 \leqslant$ *box* 5? Nein!



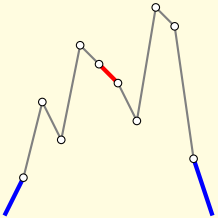
- *search* $(0, 4)$: *box* $2 \leqslant$ *box* 3? Ja!



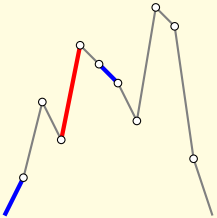
- *search* $(3, 4)$: *box* $3 \leqslant$ *box* 4? Nein!
- *search* $(3, 3)$: *box* $3 = 7$ ist ein lokales Maximum.

28. Schlag die Nachbarn: Beispiel

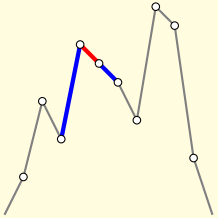
- Bisheriges Wissen in blau; Test in rot.



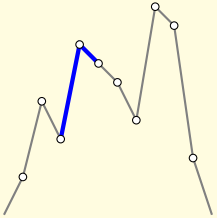
$box\ 4 > box\ 5$



$box\ 2 \leqslant box\ 3$



$box\ 3 > box\ 4$



$box\ 2 \leqslant box\ 3 \geqslant box\ 4$

28. Schlag die Nachbarn: Implementierung

Implementierung in Mini–F#:

```
let beat-your-neighbours (box : Nat → Nat)
                         (lower : Nat, upper : Nat) =
  let rec search (l, u) =
    if l = u then u
             else let m = (l + u) ÷ 2
                  if box m ⩽ box (m + 1) then search (m + 1, u)
                                         else search (l,     m)
  search (lower, upper)
```

28. Schlag die Nachbarn: Korrektheit

- Wie können wir die Korrektheit von *beat-your-neighbours* zeigen?
- Die Problembeschreibung suggeriert die *Spezifikation*:

$$box\ (i-1) \leqslant box\ i \geqslant box\ (i+1) \quad \text{wobei} \quad i = search\ (l, u)$$

- Eine Spezifikation beschreibt, *was* eine Funktion leisten soll; im Unterschied zu einer Implementierung, die genau festlegt, *wie* eine Funktion ihr Ergebnis ermittelt.

28. Schlag die Nachbarn: Korrektheit

- *Spezifikation*:

  $$box\ (i-1) \leqslant box\ i \geqslant box\ (i+1) \quad \text{wobei } i = search\ (l, u)$$

- Das Programm erfüllt die Spezifikation nicht!

-1	0	1	2	3	4	5	6	7	8	9	10
$-\infty$	0	4	2	7	6	5	3	9	8	1	$-\infty$

- Der Aufruf *search* $(5, 5)$ zum Beispiel ergibt 5; *box* $5 = 5$ ist aber kein lokales Maximum.
- Was läuft schief? Drei Möglichkeiten:
  - das Programm ist falsch oder
  - die Spezifikation ist falsch oder
  - beide sind falsch.

28. Schlag die Nachbarn: Korrektheit

Die Korrektheit von *beat-your-neighbours* hängt von einer Annahme ab.

- *Vorbedingung: search* erwartet, dass

$$box\ (l-1) \leqslant box\ l \quad \wedge \quad box\ u \geqslant box\ (u+1)$$

- *Nachbedingung: search* garantiert, dass

$$box\ (i-1) \leqslant box\ i \geqslant box\ (i+1) \quad \text{wobei} \quad i = search\ (l, u)$$

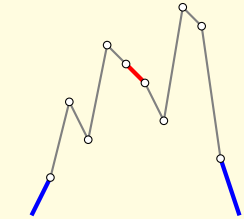
- Die Spezifikation hat die Form einer Implikation:

$$\text{Vorbedingung} \Longrightarrow \text{Nachbedingung}$$

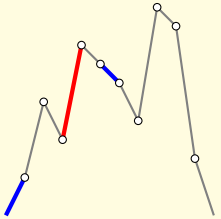
- Die Vorbedingung heißt auch *Invariante*, da sie über rekursive Aufrufe hinweg unverändert bleibt.

28. Schlag die Nachbarn: Korrektheit

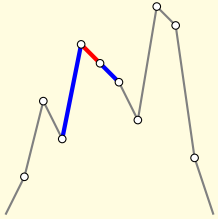
- Invariante in blau; Test in rot.



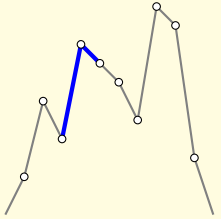
Invariante wird etabliert



Invariante wird erhalten



Invariante wird erhalten



Invariante impliziert das Ergebnis

28. Schlag den Nachbarn: Korrektheit

Invariante von *search* (l, u):
$box\ (l-1) \leqslant box\ l \ \wedge \ box\ u \geqslant box\ (u+1)$

- der initiale Aufruf *search* $(lower, upper)$ etabliert die Invariante:

$box\ (lower-1) = -\infty < box\ lower$
$box\ upper > -\infty = box\ (upper+1)$

- die rekursiven Aufrufe *search* $(m+1, u)$ und *search* (l, m) erhalten die Invariante: die Bedingungen

$box\ m \leqslant box\ (m+1) \ \wedge \ box\ u \geqslant box\ (u+1)$
$box\ (l-1) \leqslant box\ l \ \wedge \ box\ m \geqslant box\ (m+1)$

folgen aus der Invariante und der Abfrage $box\ m \leqslant box\ (m+1)$.
- die Invariante impliziert die gewünschte Nachbedingung:

$box\ (i-1) \leqslant box\ i \geqslant box\ (i+1)$

28. Schlag den Nachbarn: Korrektheit

Hmm, ist damit schon alles bewiesen? Müssen wir nicht auch sicherstellen, dass nur Schachteln aus dem Intervall *lower* . . *upper* geöffnet werden?

Gut beobachtet Lisa!

Und terminiert das Programm auch immer?

Wenden wir uns zunächst einem alten Bekannten zu ...

28. Binäre Suche — da capo

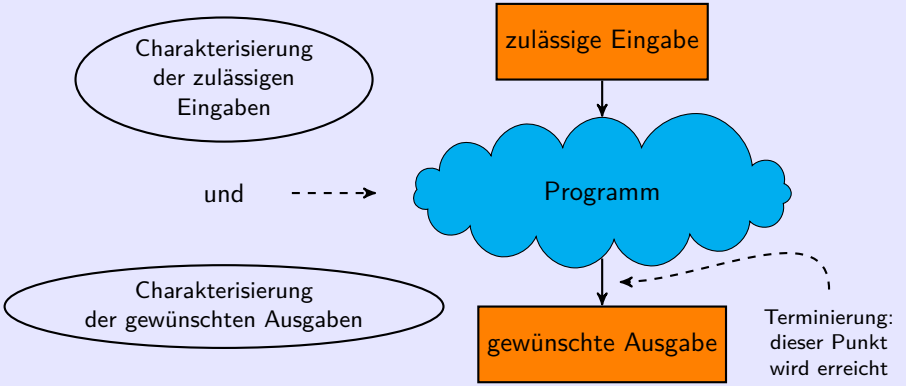
Binäre Suche:

```
let binary-search (oracle : Nat → Bool)
                 (lower : Nat, upper : Nat) : Nat =
  let rec search (l, u) =
    if l ⩾ u then u
             else let m = (l + u) ÷ 2
                  if oracle m then search (l,     m)
                              else search (m + 1, u)
  search (lower, upper)
```

Zur Erinnerung: das Orakel gibt zu einem gegebenen n Auskunft, ob die gesuchte Zahl gleich oder kleiner als n ist.

28. Partielle und totale Korrektheit

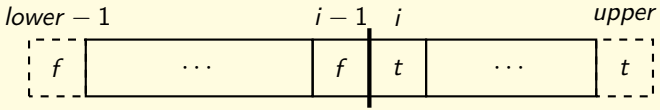
- *Partielle Korrektheit: wenn* das Programm terminiert, *dann* produziert es die gewünschte Ausgabe (für alle zulässigen Eingaben).
- *Totale Korrektheit:* das Programm terminiert *und* es produziert die gewünschte Ausgabe (für alle zulässigen Eingaben).



- totale Korrektheit $\cong$ partielle Korrektheit und Terminierung

28. Partielle Korrektheit der binären Suche

- Wie können wir formalisieren, dass das Orakel nicht mogelt?



- Zunächst gibt *oracle* immer *false* zurück; ab dem gesuchten Index immer *true*. Mit anderen Worten: *oracle* ist eine *monotone* Funktion.
- *Vorbedingungen:*

1. $i \leqslant j \implies \mathit{oracle}\ i \leqslant \mathit{oracle}\ j$
2. $\mathit{lower} \leqslant \mathit{upper}$
3. $\mathit{oracle}\ (\mathit{lower} - 1) < \mathit{oracle}\ \mathit{upper}$

- *Nachbedingung:*

$\mathit{oracle}\ (i-1) < \mathit{oracle}\ i \quad \text{wobei} \quad i = \mathit{binary\text{-}search}\ \mathit{oracle}\ (\mathit{lower}, \mathit{upper})$

28. Zwischenspiel: Ordnung auf Booleschen Werten

Die Booleschen Werte sind wie folgt angeordnet:

$$\mathit{false} < \mathit{true}$$

Wahrheitstafel für die Vergleichsoperationen:

a	b	$a \leqslant b$	$a < b$	$a = b$	$a <> b$
false	*false*	*true*	*false*	*true*	*false*
false	*true*	*true*	*true*	*false*	*true*
true	*false*	*false*	*false*	*false*	*true*
true	*true*	*true*	*false*	*true*	*false*

☞ $\leqslant$ ist die Implikation $\Longrightarrow$ („ex falso quodlibet“). (Gibt es auch andere Namen für die Vergleichsoperationen $<$, $=$ und $<>$?)

28. Partielle Korrektheit der binären Suche

1. Invariante von *search* (l, u):
$\mathit{oracle}\ (l-1) < \mathit{oracle}\ u$

- der initiale Aufruf *search* $(\mathit{lower}, \mathit{upper})$ etabliert die Invariante:

  $\mathit{oracle}\ (\mathit{lower}-1) < \mathit{oracle}\ \mathit{upper}$

- die rekursiven Aufrufe *search* $(m+1, u)$ und *search* (l, m) erhalten die Invariante: die Bedingungen

  $\mathit{oracle}\ m < \mathit{oracle}\ u$
  $\mathit{oracle}\ (l-1) < \mathit{oracle}\ m$

  folgen aus der Invariante und der Abfrage $\mathit{oracle}\ m$.
- die Invariante impliziert das gewünschte Ergebnis:

  $\mathit{oracle}\ (i-1) < \mathit{oracle}\ i$

  folgt aus $l = i = u$. *Aber:* Warum gilt $l = u$?

28. Partielle Korrektheit der binären Suche

Wir müssen zusätzlich garantieren, dass die Funktion *oracle* nur mit Werten aus ihrem Definitionsbereich aufgerufen wird.

2. Invariante von *search* (l, u):
$lower \leqslant l \leqslant u \leqslant upper$

- der initiale Aufruf *search* $(lower, upper)$ etabliert die Invariante.
- die rekursiven Aufrufe *search* $(m+1, u)$ und *search* (l, m) erhalten die Invariante: die Bedingungen folgen aus

$$l < u \implies l \leqslant m < m+1 \leqslant u$$

mit $m = (l+u) \div 2$. *Zur Erinnerung:* $m = \lfloor (l+u)/2 \rfloor$.

- die Invariante stellt sicher, dass *oracle* m definiert ist:

$$lower \leqslant m < upper$$

28. Terminierung der binären Suche

Wir müssen zeigen, dass bei jedem rekursiven Aufruf die Argumente „echt kleiner“ werden.

☞ Die Größe des Intervalls (l, u) ist $u \dot{-} l$.

☞ Um die Terminierung von *search* zu garantieren, müssen wir somit sicherstellen, dass das Intervall (l, u) bei jedem rekursiven Aufruf echt kleiner wird:

$$\begin{aligned} u \dot{-} (m+1) < u \dot{-} l &\iff l < m+1 \\ m \dot{-} l \quad\;\; < u \dot{-} l &\iff m < u \end{aligned}$$

Beide Bedingungen folgen aus der Eigenschaft des „Mittelwerts“ m mit $m = \lfloor (l+u)/2 \rfloor$.

$$l < u \implies l \leqslant m < m+1 \leqslant u$$

28. Anforderungen und Garantien

Metapher: die Benutzer*in und die Bibliotheksfunktion *binary-search* gehen einen Vertrag ein (engl. contract).

Beobachtung:

- die Benutzer*in stellt Anforderungen an das Ergebnis von *binary-search*;
- und an die Argumente von *oracle*;
- die Funktion *binary-search* stellt Anforderungen an ihr Argument;
- und an die Ergebnisse von *oracle*.

Die Gegenseite muss die Anforderungen jeweils erfüllen bzw. garantieren.

☞ *Allgemein:* die Benutzer*in stellt Anforderungen an das Ergebnis; die Gegenseite stellt Anforderungen an die Argumente. Für funktionale Argumente wie *oracle* kehren sich Anforderungen (engl. requirements) und Garantien (engl. guarantees) um.

Komisch: der Korrektheitsbeweis benutzt gar nicht, dass die Funktion *oracle* nicht mogelt.

In der Tat! Wenn ein Beweis eine Annahme nicht verwendet, hat das in der Regel zwei mögliche Ursachen: entweder der Beweis ist schlicht und einfach falsch, oder das Theorem ist tatsächlich allgemeiner als angenommen.

Ich hab noch mal fix nachgerechnet: der Korrektheitsbeweis ist wirklich korrekt ;-).

Okay, wenn das Orakel mogelt, dann gibt es möglicherweise mehrere Stellen i mit $\mathit{oracle}\ (i-1) < \mathit{oracle}\ i$. Die binäre Suche findet dann zumindest irgendeine dieser Stellen.

Unter der Annahme, dass $\mathit{oracle}\ (\mathit{lower}-1) < \mathit{oracle}\ \mathit{upper}$.

Heißt das nicht, dass sich *beat-your-neighbours* auf *binary-search* zurückführen lässt?

Hmm, die Programme sind sich in der Tat ziemlich ähnlich:

```
let rec search (l, u) =
  if l ⩾ u then u
  else
    let m = (l + u) ÷ 2
    if oracle m
      then search (l,     m)
      else search (m + 1, u)
```

```
let rec search (l, u) =
  if l = u then u
  else
    let m = (l + u) ÷ 2
    if box m ⩽ box (m + 1)
      then search (m + 1, u)
      else search (l,     m)
```

OK, die Tests $l \geqslant u$ und $l = u$ sind gleichwertig, da wir $l \leqslant u$ annehmen. Dann müssen wir noch die Zweige der zweiten Fallunterscheidung vertauschen ...

...indem wir die Bedingung negieren. Funzt:

```
let beat-your-neighbours (box : Nat → Nat) =
  binary-search (fun i → box i > box (i + 1))
```