Aufgabe 3b: Graphics

Graphische Formen

Abstrakte Idee der diskreten Punktmenge

- Wir interpretieren unsere graphischen Objekte als **Punktmengen**.
- Es gibt nur Punkte mit **ganzzahligen** Koordinaten.
  Das macht alles einfacher.

Direkte Repräsentation als Menge von Punkten

- Repräsentation als **Menge**(Set von Points in Ruby)
- Repräsentation als **Bildmatrix** (Bitmap, Array in Ruby)

Indirekte Repräsentationen durch Formeln

- Repräsentation als **Text** (Formel als geschachtelter Ausdruck)
- Repräsentation als **Datenstruktur** (Parsetree der Formel)

1d-Punkte als primitivste Form

- Als "**1-dim Punkte**" verwenden wir unsere normalen Integerzahlen. Das macht die Notation kompakt.
- Die "**2-dim Punkte**" sind kartesische Produkte von Integerzahlen.

Intervallbildung

- Wir bilden diskrete **Intervalle** in **1** und **2 Dimensionen.**
- Die **"1-dim Intervalle"** sind die normalen Intervalle auf Integer.
- Diese können wir sehr kompakt mit dem Intervalloperator in Ruby "_.._" notieren.
- Die **"2-dim Intervalle"** sind kartesische Produkte von **1-dim Intervallen.**

  Anschaulich sind das **Rechtecke**, die durch zwei 1-d Intervalle bestimmt sind**.**

Additive Kombination von Formen gleicher Dimension

- Wir setzen **komplexere** Objekte **innerhalb** einer Dimension durch **Kombination** von Objekten der gleichen Dimension zusammen.
- Als **Kombinationen** betrachten wir zunächst nur die **Vereinigung** von Formen. Semantisch ist damit die Mengen-Vereinigung der Punktmengen gemeint.
- Das ist eine Art von **Addition**. Dabei bleiben wir innerhalb einer Dimension, erhalten aber neue, zusammengesetzte Formen.
- Diese Addition läßt sich beliebig schachteln und führt zu einer Baumstruktur.

Kombination von 1d-Formen zu 2d-Formen

- Wir setzen graphische Objekte einer **höheren** Dimension aus Objekten einer **niederen** Dimension zusammen.
- Das ist eine Art von **struktureller Multiplikation**.
- Ziemlich interessant, denn diese "Multiplikation" führt uns in **höhere** Dimensionen.
- Die Multiplikation von zwei 1d-Punkten ergibt einen 2d-Punkt.
- Die Multiplikation von zwei 1d-Intervallen ergibt ein 2d-Intervall (also ein Rechteck).
- Das kann man auf beliebig viele Dimensionen erweitern, aber wir bleiben bei zwei Dimensionen.

Datendefinitionen

Typen für 1d-Objekte

Int ::= schon eingebaut

Point1d ::= Int

Range1d ::= Range[first,last] :: Point1d x Point1d

Union1d ::= Union1d[left,right] :: Shape1d x Shape1d

Shape1d ::= Range1d | Union1d

Typen für 2d-Objekte

Point2d :: = Point2d[x,y] :: Point1d x Point1d

Range2d ::= Range2d[x_range,y_range] ::
Range1d x Range1d)

Union2d ::= Union2d[left,right] :: Shape2d x Shape2d

Shape2d ::= Range2d | Union2d

Übergreifende Typen für 1d und 2d Objekte

Dieses ist eine Zusammenfassung von Eigenschaften über die Dimensionen hinweg.

Point ::= Point1d | Point2d

PrimShape ::= Range1d | Range2d

UnionShape ::= Union1d | Union2d

CompShape ::= UnionShape | …(bisher noch nicht mehr)

Shape ::= PrimShape | CompShape

GraphObj ::= Point | Shape

Automatisch definierte Typprädikate

- Die Prädikate für die Klassentypen werden automatisch erzeugt, da die Produkte direkt als Rubyklassen realisiert werden.

  Das **def_class** erzeugt automatisch ein **totales Typprädikat** in OO-Notation.

Selbst definierte Typprädikate

- Die restlichen Prädikate müssen Sie selbst implementieren.

  Diese sind dann Prädikate in **funktionaler** Notation.

- Es sind **viele**, aber sie sind einfach und sehr nützlich.

- Achten Sie dabei auf **Totalität auf Any**.

Relationalprädikat für "Enthaltensein" von Punkten in Formen

shape_include? ::= (shape,point) :: Shape x Point ->? Bool

Das Prädikat soll bestimmen, ob ein Punkt in einem Shape enthalten ist

- das Prädikat soll für alle Dimensionen definiert sein

- erfordert Rekursionen und Fallunterscheidungen

- wünschen Sie sich Funktionen, die die einzelnen Fälle lösen

Funktionen auf den Formen

Translation von Formen

translate ::= (shape, point) :: Shape x Point ->? Shape

- Diese Funktion erzeugt eine **neue** Form, die um einen **Translationsvektor** verschoben ist.
- Dieser Vektor kann als Punkt **point** repräsentiert werden
- Die Funktion soll für alle Dimensionen definiert sein, aber die Dimension von shape und point muß gleich sein.
- Es ist eine Rekursion erforderlich, die einen **neuen Baum** gleichartiger Form erzeugt.

Reduktion von Formen auf kleinste Bounding Boxen

bounds ::= (shape) :: Shape -> (Range1d | Range2d)

Eine **Bounding-Box** einer Form ist ein Intervall, das die Form enthält.

Interessanter ist die **kleinste Bounding-Box** einer Form.

Das ist im Allgemeinen ein algorithmisch ziemlich schwieriges Problem.

Solange wir aber nur Vereinigung betrachten ist es aber einfach.

Denn es gilt

- Die **Bounding-Box** einer **Vereinigung** von zwei Formen ist die **Bounding-Box** der **Vereinigung der Bounding-Boxen** der zwei Formen.
- machen Sie sich dieses an Beispielen klar (Zeichnungen)
- diese Funktion soll für alle Dimensionen definiert sein

Bounding Range von zwei Ranges

bounding_range ::= (r1,r2) :: (Range1d x Range1d) -> Range1d |
(Range2d x Range2d) -> Range2d

Diese zweistellige Operation bestimmt das **kleinste Intervall**, das zwei gegebene Intervalle umfaßt.

Diese Funktion macht es einfach, die Funktion bounds zu programmieren.

Äquivalenzprädikate (Gleichheiten)

Strukturelle Gleichheiten von Formen

Dimensionsgleichheit

equal_by_dim? ::= GraphObj x GraphObj -> Bool

- gleich, wenn Dimensionen gleich
- praktisch für Preconditions

Strukturelle Wertgleichheit von Form und Lage

equal_by_tree? ::= GraphObj x GraphObj -> Bool

- bedeutet, daß die Bäume strukturell wertgleich sind
- bedeutet, daß beide Objekte von der gleichen "Formel" in Ruby erzeugt wurden.

Strukturelle Wertgleichheit von Form ohne Lage

equal_by_trans? ::= GraphObj x GraphObj -> Bool

- In der Geometrie bezeichnet man zwei Formen als **kongruent**, wenn man sie "deckungsgleich" übereinanderlegen kann.
- Wir wollen nur den einfachsten Fall betrachten, nämlich, daß zwei Formen durch **Verschiebung** zur Deckung gebracht werden können. Rotation und Spiegelung lassen wir weg.
- Dieses ist eine zweite (deutlich gröbere) Variante einer Äquivalenzrelation.
- Wir betrachten dabei zwei "Formeln" als gleich, wenn sie durch Translation ineinander überführt werden können.

Denken Sie "geometrisch" nach, bevor Sie programmieren.

Sie dürfen selbstverständlich **alle anderen** Funktionen aus der Aufgabe verwenden.

Damit ist eine kurze und prägnante Lösung möglich.

Semantische Gleichheit von Formen

Wertgleichheit der Punktmengen

Nur der Vollständigkeit halber (noch nicht programmieren):

equal_by_points? ::= GraphObj x GraphObj -> Bool

- Damit ist gemeint, das zwei Formeln zwar verschieden "aussehen", aber die gleiche Punktmenge definieren.
- Der direkte Ansatz wäre, die Punktmengen zu erzeugen und diese dann auf Gleichheit zu untersuchen.
- Das können wir im Moment noch nicht programmieren, aber wir können schon ein **wichtiges Prädikat** implementieren, das dafür nützlich ist.
- Ein effizienterer Algorithmus, um festzustellen, ob zwei Formeln, die gleiche Punktmenge beschreiben, ist viel zu schwierig zu entwickeln.

Eine kleine "Programmiersprache" für Formen

Wir schaffen uns eine kleine **eigene Programmiersprache** für die Konstruktion graphischer Formen.

- Diese beschreibt Formen durch **Formeln** (Ausdrücke in Ruby)
- Jede Formel hat eine Interpretation.
- Das ist bei uns die durch die Formel "beschriebene" **Punktmenge**.
- Damit haben wir eine Definition der **Semantik** unserer Sprache.

Die Entwicklung einer kleinen Sprache für einen Problemkreis ist ein sehr wichtiges **Abstraktionsprinzip**.

- Mit selbst definierten **Operatoren** können wir die Notation noch kompakter und lesbarer machen.
- Das machen wir in einem späteren Schritt. Das ändert aber nichts an dem prinzipiellen Konzept.