CSC227
Formal Specification of Software(5)

John Fitzgerald
Centre for Software Reliability
University of New Castle

Translated by
Takahiko Ogino
Railway Technical Research Institute

Based upon the book
Modelling Systems: Practical Tools and Techniques in Software Development
(ISBN 0 521 623480 Cambridge University Press 1998)
by John Fitzgerald and Peter Gorm Larsen (IFAD)

凡例

- 本翻訳は、John Fitzgerald博士（Centre for Software Reliability、University of New Castle）の講義用OHPを翻訳したものである。
- 翻訳にあたっては、なるべく忠実な訳をしたつもりである。ただし原稿が、授業の教材であることを勘案し、出来るだけ意味的な注釈を付けたり、空白になっておりクラスで埋められるのであろうと思われる部分は、出来るだけ講義のベースとなっている本、Modelling Systemsからの引用によって中身を埋めた。
- そのような部分をオリジナルと区別するため、フォントの色を変化させることで出来るだけ区別している。
- また、同じ鉄道総研の寺田夏樹君には、日本語と内容のプルーフリーディングをお願いしたことを記載し感謝の意を表します。
- 原OHPの修正も含めて全ての日本語版内容に関する責任は、荻野にあります。
- 誤訳も含めて、内容に関するご指摘をお待ちしています。

集合を使用してのモデリング

- セット
  - 有限集合を構成するタイプ
  - 値の定義の方法:枚挙(enumeration)、サブレンジ、
    内包(comprehension) – 性質を示し集合をあらわす。
    <-> 外延(denotation) – 明示的に集合の要素を示す。(枚挙)
  - 集合の上のオペレータ

- 事例研究:爆発物保管庫

  - *タイプを定義するために:*
    - *タイプ構成子(a type constructor)*
    - *値を書き下す方法*
    - *値のオペレーションの方法*

集合とは

- 値の順序のない集まり
- 順序は問題で無い。
- 同じものも含まれない。

次のものを集合として表しなさい。なぜまずい？

- 航空管制の表示：管制下にある空域の航空機の位置と識別子を表示する。
- 出発の順にフライトを示す空港での出発ディスプレイ。
- 任意の与えられた時に特定のエリアにいる人間を追跡するデータベース

セットタイプ構成子

- 有限セットタイプ構成子: `set of _`
- {1、5、7} ： set of nat
- 次の式のタイプは何か？

```
{1, -3, 12}
set of int

{ {9, 13, 77}, {32, 8}, {}, {77} }
set of (set of nat1)
```

セットタイプ構成子

- タイプset of Xは、タイプXから取られた値からなる全ての有限集合を表すクラスである。例えば:

`set of nat1`	ゼロを含まない自然数の集合
`set of Student`	学生の記録の集合
`set of (seq of char)`	文字のシーケンスの集合 (例えば、名前の集合)
`set of (set of int)`	整数の集合達の集合、例えば `{ {3,56,2},{-2},{},{-33,5} }`

集合を定義する

(0) 空集合:　　{}

(1) 数え上げ(Enumeration):

(2) サブレンジ(整数のみ): {integer1,...,integer2}

例えば、　　{12,...,20} =

{12,13,14,15,16,17,18,19,20}

{12,...,12} =

{12}

{9,...,3} =

{ }

集合を定義する

(3) 内包(Comprehension)

{ *expression* | *binding* & *predicate* }

- 述語を満たす束縛変数の値に対して、その値を用いた式(expression)の計算値の集合。
  - 結合(binding)に書かれている全ての変数が取ることの出来る全ての値を考える。
  - その中で、述語を満たす値の組み合わせのみに限定する。
  - 各組合せにおいて、式(expression)を評価する。これによって集合の値が与えられる。
- 例えば、

```
{x**2 | x:nat & x<5 }
```

{0, 1, 4, 9, 16}

集合を定義する

Comprehensionの例:

1. {x | x in set {1,…,15} & x < 5}
2. {x | x:nat & x < 5}
3. {y | y in set {1,…,20} & exist x in set {1,…,3} &
   x*2 = y}
4. {y | y:nat & y < 0}
5. {x+y | x,y in set {1,…,4}}
6. {x+y | x,y:nat & x<3 and y<4}

集合を定義する

- 値を具体的に書き下すと:

1. {1,2,3,4}
2. {0,1,2,3,4}
3. {2,4,6}
4. { }
5. {2,3,4,5,6,7,8}
6. {0,1,2,3,4,5}

集合を定義する　　有限性

- VDM−SLでは、セットは有限でなければならない。従って、内包(comprehension)を定義するときには、無限の値によって満足される可能性がある述語は定義してはならない。
- 例:

```
{ x | x:nat & x > 10}
```

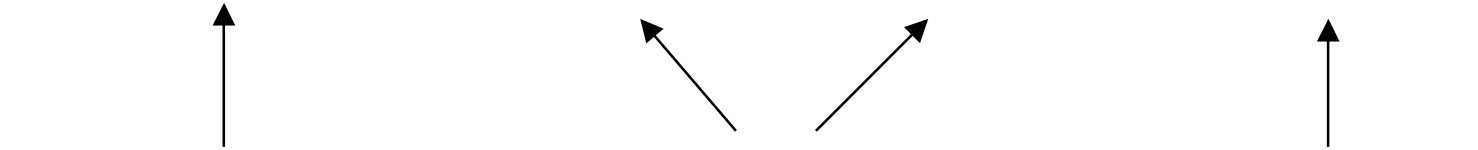
- 無限の値を取る集まりを定義するときは、集合というよりタイプとして定義すること。そのタイプを不変式を用いて制約すると良い。

```
BigNats = nat
inv x == x > 10
```

集合のオペレータ

- 集合に関する多くの(VDM-SL)内蔵のオペレータがある。
- 各々は、オペランドの数やタイプを定義するシグネチュアを持っている。例えば、セット結合オペレータの場合

```
_ union _ : set of A * set of A -> set of A
```



オペレータの名前。下線は、オペレータが使われる場合のパラメータの場所を示す。

順番に入力のタイプを表す。"*"は各入力タイプを分ける。

結果のタイプ

このシグネチュアからunionオペレータに関して何が言えるか？

集合のオペレータ

```
_ union _ : set of A * set of A -> set of A
```

結合(union)　両方の集合のすべての要素を集めたもの

```
{89, 33 , 5} union {2, 9, 5} = {89, 33, 5, 2, 9}
```

- 次の式はシグネチュアからすると正しいか？

1. union({4, 7, 9}, {23, 6})
2. 3 union {7, 1, 12}
3. {12,…,15} union {x-y | x,y:nat & x<4 and y<10}
4. {} union {}
5. {<Red>, <Amber>} union {3}
6. {12} union {x**y | x,y:nat & x<4 and y>2}

集合のオペレータ

- オペレータを有効に使用するために、あなたは、オペランドの数およびタイプを知らなければならない、また結果として返される値のタイプも知る必要がある。
- この情報はノートおよびテキストの中で与えられる。また、あなたは言語を有効に使用するために実際にそれを知る必要がある。

集合のオペレータ

```
_ union _  : set of A * set of A -> set of A
_ inter _  : set of A * set of A -> set of A
_ ¥ _      : set of A * set of A -> set of A
dunion     : set of (set of A)   -> set of A
dinter     : set of (set of A)   -> set of A
card       : set of A            -> nat
_ in set _ : A * set of A        -> bool
_ subset _ : set of A * set of A -> bool
```

- 注:オペレータがプレフィックス形式で使用される場合、下線は、示さない。例えば

```
card {12, 45, 12, 3} = 4
```

集合のオペレータ

```
_ inter _  : set of A * set of A -> set of A
```

交差(intersection)

両方の集合に共通に含まれる要素の集合

```
{89, 33, 5} inter {2, 9, 5} = {5}
```


```
_ ¥ _       : set of A * set of A -> set of A
```

差(difference)

1つ目の集合から2つ目の集合に含まれる要素を除いたもの

(2つめの集合の要素が1つ目の集合にある必要はない)

```
{89, 33, 5} ¥ {43, 5, 22} = {89, 33}
```

集合のオペレータ

- 最も一般的なオペレータは、二つだけでなく、パラメータの全体のセットまでそれらが延長出来る特殊フォームを持っている。

```
dunion    : set of (set of A)   -> set of A
```

distributed union

各集合の全ての要素を取り出したもの

```
dunion {{1,3,5},{12,4,3},{3,5,11}} = {1,3,4,5,11,12}
```

```
dinter    : set of (set of A)   -> set of A
```

distributed intersection

全ての集合に共通して含まれる要素

```
dinter {{1,3,5},{12,4,3},{3,5,11}} = {3}
```

集合のオペレータ

```
card        : set of A                -> nat
```

集合に含まれる要素の数(cardinality)

```
card {1, 5, 7, 5, 3}=4
card {} = 0
```

```
_ in set _ : A * set of A             -> bool
```

ある要素が集合に存在する

```
5 in set {1,3,5,7,9} = true
6 in set {1,3,5,7,9} = false
```

```
_ subset _ : set of A * set of A -> bool
```

部分集合(subset)(前者が後者の部分集合)

```
{89, 33, 5} subset {5} = false
{5} subset {89, 33, 5} = true
```

ケーススタディ　　爆発物保管庫の例

- モデル化するシステムは、倉庫の中に、ダイナマイトや信管等の爆発物を配置するロボット用の制御装置の一部である。
- 倉庫は長方形のビルディングである。建物の中の位置は、1つのかどを原点として設定された座標で表される。倉庫の大きさは、最大のxおよびy座標として表される。
- 倉庫内で対象物は、長方形のパッケージであり、向きを倉庫の壁にそろえて並べられる。各対象物は、xおよびy方向の次元を持っている。対象物の位置はその左下コーナーの座標として表される。対象物はすべて倉庫の中に収まらなければならない。また、対象物間にオーバラップはあってはならない。

ケーススタディ　　爆発物保管庫の例

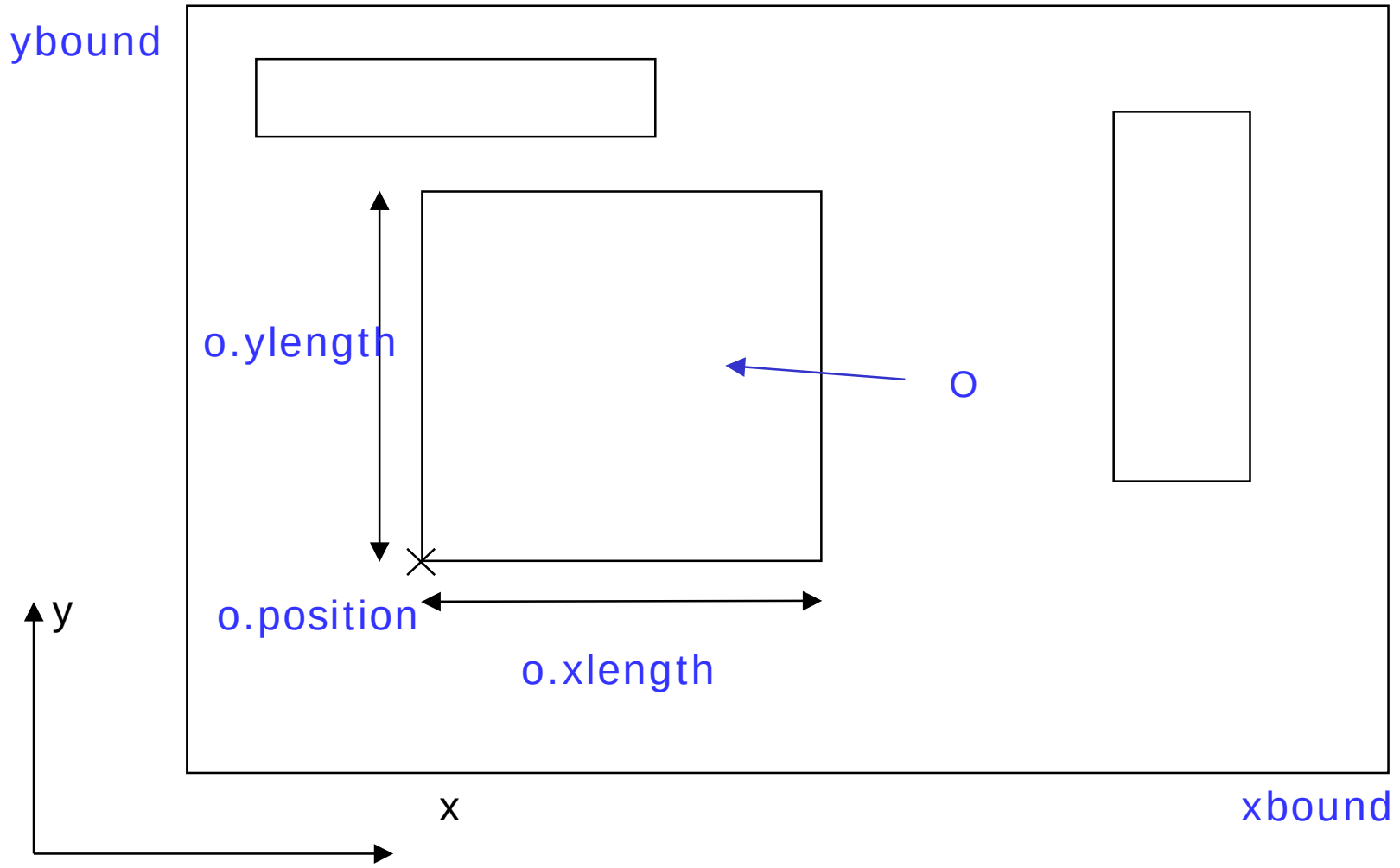
位置制御装置は次の機能を果さなければならない。

1. 与えられた倉庫内のオブジェクトの数を返す;
2. 与えられた対象物が与えられた倉庫内で、収まるための位置を示唆する;
3. 与えられた対象物が与えられた位置に置かれたことを記す倉庫の記録を更新する;
4. 位置の集合が与えられた場合、その中の全ての対象物がすべて移動させられたことを記す倉庫の記録を更新する

モデルの目的:

- 倉庫内の対象物の安全なポジショニングのルールを定めること
- そのため、対象物の表現、位置、倉庫の範囲に重点を置く

ケーススタディ　　爆発物保管庫の例



ケーススタディ　爆発物保管庫の例

```
Store  ::   contents :
            xbound   :
            ybound   :

Object ::   position :
            xlength  :
            ylength  :
```

ケーススタディ　　爆発物保管庫の例

```
Store ::  contents : set of Object
          xbound : nat
          ybound : nat
inv mk_Store(contents, xbound, ybound) ==
          対象物は、倉庫の範囲内にある ― (1)
          and
          異なった対象物は、重なって置かれていない ― (2)

Point ::  x : nat
          y : nat

Object :: position : Point
          xlength : nat
          ylength : nat
```

ケーススタディ　　爆発物保管庫の例

```
Store :: contents : set of Object
         xbound : nat
         ybound : nat
  inv mk_Store(contents, xbound, ybound) ==
  (forall o in set contents &
      InBounds(o,xbound,ybound)) and ―(1)
  not exists o1, o2 in set contents &
      o1 <> o2 and Overlap(o1,o2)  ―(2)
(2)の別の例 ・ forall o1,o2 in set contents &
                  not(o1<>o2) or not Overlap(o1,o2)
            ・ forall o1,o2 in set contents &
                   o1 <> o2 => not Overlap(o1,o2)
InBounds: Object * nat * nat -> bool
InBounds(o,xb,yb) ==
  o.position.x + o.xlength <= xb and
  o.position.y + o.ylength <= yb;
```

ケーススタディ　　爆発物保管庫の例

```
Store ::      contents : set of Object
              xbound : nat
              ybound : nat
   inv mk_Store(contents, xbound, ybound) ==
   forall o in set contents &
        InBounds(o,xbound,ybound) and
   not exists o1, o2 in set contents &
        o1 <> o2 and Overlap(o1,o2)
```

Overlap:

対象物をポイントの集合と考え、それらに共通部分があること

```
Overlap(o1,o2) == Points(o1) inter Points(o2) <> { }

Points: Object -> set of Point
Points(mk_Object(pos,xlen,ylen)) ==
   {mk_Point(x,y)|x in set {pos.x ,..., pos.x + xlen},
                  y in set {pos.y ,..., pos.y + ylen}}
```

ケーススタディ　　爆発物保管庫の例

1. 与えられた倉庫の対象物の数を返す;

```
NumObjects: Store -> nat
```

2. 与えられた対象物が与えられた倉庫内で、収まるための位置を示唆する;

```
SuggestPos: nat * nat * Store -> Store
```

3. 与えられた対象物が与えられた位置に置かれたことを記す倉庫記録を更新する;

```
Place: Object * Store * Point -> Store
```

4. 与えられた位置の集合にある対象物がすべて移動させられたことを記す倉庫記録を更新する。

```
Remove: Store * set of Point -> Store
```

ケーススタディ　爆発物保管庫の例

```
NumObjects: Store -> nat
NumObjects(store) == card store.contents
```

card 集合に含まれる要素の数を返す関数

```
card {1, 5, 7, 5, 3} = 4
card {} = 0
card {x*x | x in set {-5,...,3}} = 6
```

ケーススタディ　　爆発物保管庫の例

```
SuggestPos: nat * nat * Store -> Store

SuggestPos(xlength,ylength,s) == ???
```

- いくらでも実行可能な位置が存在するかもしれ無いが、要求仕様は、特にどの位置を返さなければならないかについて明確ではない。十分なスペースを備えたいかなる場所でも仕様を満足している
- 上記の理由により、特定の位置を与える必要が無いので、位置を見つけるためのアルゴリズムを与える必要ない:そのため、**陰定義**の関数を代わりに使用することができる。
- 言い換えると、ここのレベルの抽象化では最適化の話は無関係である。最適化を考えると、可能性のある位置から選択する必要があるかもしれない。
- どのように位置を与えるかは、実際にプログラムを作成する人の自由に任せる。(自由度がある:looseness性)
- $X^2=4$には、具体的には、2つの解があるが、looseness性により、具体的に−2か2かを指定していない。個々の値のことより、全体の見通しが重要なときに用いる。

ケーススタディ　　爆発物保管庫の例

- 陰定義は本体を持ってない。事後条件（postcondition）を用いて結果も含めて関数の実行後に満足されるべき条件のブール式を記述する。

functionName (input vars & types) result & type

pre　　*precondition*

post　　*postcondition*

```
sqrt(x:real) r:real
pre   x >= 0
post  r*r = x
```

- どうやってrを計算するかという記述はない。パラメータxと出力rとの関係を記述している。

ケーススタディ　　爆発物保管庫の例

```
SuggestPos: nat * nat * Store -> Store
SuggestPos(xlength,ylength,s) == ???


SuggestPos(xlength:nat,ylength:nat,s:Store) p:[Point]
post  -- その場所(point)に対象物が格納できる十分な空間がある
      -- ならば、その場所の値を返す。
      -- さもなければnilを返す。
  if exists poss:Point &
                RoomAt(xlength,ylength,s,poss)
     then RoomAt(xlength,ylength,s,p)
   --RoomAtを真とするようなpが出力のpであるという意味である。
   --（Postconditionを満足させるpが、SuggestPosが返す値となる。）
   --もし、RoomAtが何らかの値で真となることが保証されるのなら、
   --postconditionはRoomAt(xlength,ylength,s,p)だけである
  else p = nil
```

ケーススタディ　　爆発物保管庫の例

```
RoomAt: Object * Store * Point -> bool
RoomAt(o,s,p) ==
   let new_o = mk_Object(p,o.xlength,o.ylength) in
       InBounds(new_o,s.xbound,s.ybound) and
       not exists o1 in set s.contents &
                                Overlap(o1,new_o)
```

let *pattern* **=** *defining-expression*
in
　use-expression

この構文は次のように評価される:

- *defining-expression*を評価し、パターン（この場合は、new_o）に対してその結果を対応させる。
- *use-expression*中のパターンの中の識別子（この場合は単にnew_o）を、上のステップで得た値で置き換えて*use-expression*を評価する。

ケーススタディ　　爆発物保管庫の例

3. 与えられたオブジェクトが与えられた位置に置かれたことを記す倉庫の記録を更新する:

```
Place: Object * Store * Point -> Store
Place(o,s,p) ==
   let new_o = mk_Object(p,o.xlength,o.ylength) in
   mk_Store (s.contents union {new_o},
             s.xbound,
             s.ybound)
   pre RoomAt(o.xlength,o.ylength,s,p)
```

ケーススタディ　　爆発物保管庫の例

- 保管庫内で与えられた位置の集合にあるオブジェクトを取り除くことをモデルする関数:

```
Remove: Store * set of Point -> Store
Remove(mk_Store(contents,xbound,ybound),sp) == ...

Remove(mk_Store(contents,xbound,ybound),sp) ==
  let os =
    {o |o in set contents & o.position in set sp} in
     mk_Store(contents ¥ os, xbound, ybound)
pre sp subset {o.position | o in set contents}
```

ケーススタディ　　爆発物保管庫の例

- 拡張–倉庫の集まりから成るサイトを考える:

```
Store :: contents : set of Object
         xbound : nat
         ybound : nat
         name : token
inv mk_Store(contents, xbound, ybound, -) ==
  (forall o in set contents &
             InBounds(o,xbound,ybound)) and
   not exists o1, o2 in set contents &
                      o1 <> o2 and Overlap(o1,o2)

Site = set of Store
inv site ==
  forall store1, store 2 in set site &
    store1. name = store2.name => store1 = store2
```

ケーススタディ　爆発物保管庫の例

- また、サイトの内容目録を持つ必要がある:

```
Inventory = set of inventoryItem
InventoryItem :: store : token
                 item  : Object
```

- それぞれの倉庫の個々の内容目録のユニオンが可能である:

```
SiteInventory: Site -> Inventory
SiteInventory(site) ==
  dunion{StoreInventory(store) | store in set site}


StoreInventory: Store -> Inventory
StoreInventory(store) ==
  {mk_InventoryItem(store.name,o) |
   o in set store.contents}
```

レビュー

- セット構成子:
  - 個々のセットは、次の方法で作る:
    - 数え上げ(enumeration)
    - 部分範囲(subrange)
    - 内包(comprehension)

- 集合を扱うオペレータ:シグネチュアで定義;結合(union)と交差(intersection)の2つ以上のパラメータに適応可能なタイプ(dunion、dinter)のものも含む

- 爆発物の例
  - 特定の結果を明示する必要が無い時は、要求仕様(postcondition)を陰定義して与えるようにする。
  - モデルを構築するタスクを分解し、単純化するために、補助の関数定義を使用すること。

演習問題

- 実際には、火薬類のすぐ近くに信管があると重大な危険がある。
- この演習では、本章の中で発展されていたモデルがこれを考慮するために拡張される。
- 火薬類の隣が火薬類でも良く、信管の隣が信管でも良いが、信管は火薬類から少なくとも10ユニット離れていなければならない。
- 対象物を2つのクラスに定義することから始める:火薬類と信管
- 各対象物のまわりの安全なスペースを定義し、モデルの中の関数を修正し、1つのクラスの対象物が他方のクラスの対象物のまわりの安全なスペースに置かれないようにする。

. . . . . .

1. 2つの値を含んでいる列挙型クラスを定義せよ: <Expl>(火薬類)および<Fuse>(信管)。
2. それが対象物のクラスを指示するフィールドを持つように型Objectを再定義せよ

演習問題

3. 次の関数を定義する:

```
SafeSpace: Object * Object -> set of Point
SafeSpace(o,s) == ???
```

我々は、o.position.x - 10からo.position.x + 10までと、
o.position.y - 10からo.position.y + 10のポイントのセットとして
オブジェクトoのまわりの安全なスペースを定義する。

- ここで、Let表現を用いて、:

```
let xrange = {o.position.x - 10,...,o.position.x + 10}
   yrange = {o.position.y - 10,...,o.position.y + 10}
   in
        {mk_Point(x,y) | x in set xrange, y in set yrange}
```

演習問題

- オブジェクトoが、倉庫の壁の10ユニット以内にある場合、問題が発生する。
- 例えば、o がmk_Point(5,3)とすると、そのとき安全なスペースには、負の座標を持つことがあるが、座標は自然数であると定義してきた。
- モデル製作者にはここで選択子がある。
- 1つのアプローチは負数を許可するために座標タイプをintとする方法である。
- ここでは、安全なスペースを「切り取る」関数を定義する別のアプローチを行う。

```
Bottom: nat -> nat
Bottom(n) ==
   if n < 10 then 0 else n –10
```

上の関数を用いて、出力の座標が非負となるようなSafeSpaceを定義せよ

4. 関数Overlapの定義を変更し、o1とo2の中のポイントが異なるクラスを持っており、且つ、o1のSafeSpaceが，o2の中のポイントとオーバーラップする場合真となるようにせよ。

解答

```
1. Class = <Expl> | <Fuse>
2. Object :: class : Class
              position : Point
              xlength : nat
              ylength : nat
3. SafeSpace: Object -> set of Point
   SafeSpace(o) ==
   {mk_Point(x,y)|
      x in set {Bottom(o.position.x),...,o.position.x+10},
      y in set {Bottom(o.position.y),...,o.position.y+10}}
4. Overlap: Object * Object -> bool
   Overlap(o1,o2) ==
       Points(o1) inter Points(o2) <> {} or
              (o1.class <> o2.class and
              SafeSpace(o1) inter Points(o2) <> {});
```