ALICE

Pallinger Péter

2004. október 18.

Hanák Péter javításaival, kiegészítéseivel

1 Korlátalapú programozás

1.1 Korlátalapú programozás véges halmazokon

A korlátalapú programozást kombinatorikus feladatok megoldására szokták használni, ahol a megoldást jelentő változókra különböző kikötések tehetők, és a változók értékkészlete korlátos, tipikusan egész számok egy véges halmaza.

A korlátalapú programozás két alaptechnikája a korlátterjesztés (constraint propagation) és a korlátfelosztás (constraint distribution). A korlátterjesztés egy hatékony következtetési mechanizmus, amely egyidejűleg több olyan „terjesztőt” (propagator) használ, amelyek egy korláttárban (constraint store) gyűjtik a kikövetkeztetett információt. A korlátfelosztás a problémát egymást kölcsönösen kizáró részekre bontja, ha a korlátterjesztés mechanizmusa megakad. A két technika váltogatott alkalmazásával a terjesztő végül meg fogja találni a megoldást.

A felosztás könnyen exponenciális növekedéshez vezet a generált részproblémák számában. Szerencsére ez a potencionális robbanás megelőzhető megfelelően erős terjesztési (propagációs) szabályok használatával.

Az ALICE korlátmegoldó képességeit közvetlenül a MOZART-tól örökölte.

2 Példák

2.1 Egyszerű példa

Először is be töltsük be a lineáris korlátok felvételéhez szükséges csomagot.

```
open Linear;
```

Ezután hozzunk létre egy FD-változókat tartalmazó vektort (FD = Finite Domain, véges értelmezési tartomány):

```
val #[x,y,z] = vec (3, [1`#10]);
```

Végül adjuk meg pl. a következő korlátokat:

```
post (`2 `* x `= y);
post (z `< x);
post (y `< `7);
post (z `<> `1);
```

Egy inspektorral figyelhetjük meg, hogyan alakul a változók értékkészlete az egyes korlátok felvétele után.

Figyeljük meg, hogy itt nem volt szükségünk korlátfelosztásra, de ez persze nem mindig van így.

2.2 Egy kicsit bonyolultabb példa

2.2.1 A feladat

Álljon itt a sokak által ismert példa:

$$SEND + MORE = MONEY.$$

Ebben az egyenletben minden betű egy (tízes számrendszerbeli) számjegyet jelöl.
A feladat egyetlen megoldása:

$$9567 + 1085 = 10652.$$

2.2.2 ALICE-megoldás

Először definiálnunk kell egy ún. *script*-et a probléma leírására.

ALICE

```
fun money () =
  let
      val v as #[S,E,N,D,M,O,R,Y] = Linear.vec (8, [0`#9])
  in
      distinct v;
      post (S `<> `0);
      post (M `<> `0);
      post (`1000`*S `+ `100`*E `+ `10`*N `+ D `+
            `1000`*M `+ `100`*O `+ `10`*R `+ E `=
            `10000`*M `+ `1000`*O `+ `100`*N `+ `10`*E `+ Y );
      distribute (FD.FIRSTFAIL, v);
      {S,E,N,D,M,O,R,Y}
  end;
```

Mivel ez a probléma keresés nélkül nem oldható meg, használjuk az ALICE beépített keresőjét erre a célra:

```
Search.searchAll money;
```

Megnézhetjük a megoldást így:

ALICE

```
inspect it;
```

Vagy így is, ráadásul az alábbi parancs a keresési fát is megmutatja:

```
Explorer.exploreAll money;
```

2.3 Még egy klasszikus feladat – az N-királynő probléma

```
fun nQueens n () =
    let
        val v = FD.rangeVec (n, (0, n-1))
        val v1 = Vector.mapi (fn (i,x)=>(x, i)) v
        val v2 = Vector.mapi (fn (i,x)=>(x,~i)) v
    in
        FD.distinct v;
        FD.distinctOffset v1;
        FD.distinctOffset v2;
        FD.distribute(FD.FIRSTFAIL, v);
        v
    end
```

A megoldáshoz felhasználjuk az `FD.distinctOffset` korlátot, amelynek a szignatúrája:

```
FD.distinctOffset : (FD.fd * int) vector
```

és a jelentése, ha a vektor (fd_j, i_j) párokból áll:

$$distinctOffsetv = distinct(fd_0 + i_0, fd_1 + i_1, \ldots, fd_n + i_n)$$

A megoldás megkereséséhez az adott méretű táblára egy segédfüggvényt definiálunk:

```
fun fiveQueens () = nQueens 5 ();
Search.searchAll fiveQueens;
Explorer.exploreAll fiveQueens;
```


3 Áttekintés

Az ALICE-ban több modul is található korlátprogramozás megvalósítására.

3.1 FD

Véges tartományokon (értsd: nem negatív egészeken) értelmezett változókat hoz létre, szűkítő szabályokat vesz fel és alkalmaz rajtuk.

Ha egy propagátor felébred, akkor megpróbálja az általa figyelt változók értékkészletét leszűkíteni. A propagátorok egy része csak tartományszűkítést hajt végre, mások halmazszűkítést végeznek, míg megint másoknál beállítható, hogy melyik módszert használják. Egy propagátor megszűnik, ha minden hozzá rendelt változó behelyettesítődött. Persze vannak olyan propagátorok is, amelyek már előbb (de ez ritka).

3.1.1 A `FD` struktúra függvényei

```
type fd
type bin = fd
exception Tell
```

ALICE

```
datatype domain_element = SINGLE of int | RANGE of int * int
type domain = domain_element vector
datatype relation = LESS | LESSEQ | EQUAL | NOTEQUAL |
                    GREATER | GREATEREQ
datatype dist_mode = NAIVE | FIRSTFAIL | SPLIT_MIN |
                     SPLIT_MAX | NBSUSPS
datatype assign = MIN | MID | MAX
inf
sup
fd dom
fdVec (n, dom)
range (il, ih)
rangeVec (n, il, ih)
bin ()
binVec n
assign (as, v)
toInt f
future f
fromInt i
```


ALICE

```
isBin f
sum (v, rel, r)
sumC (v, rel, r)
sumAC (v, rel, r)
sumCN (v, rel, r)
sumACN (v, rel, r)
sumD (v, rel, r)
sumCD (v, rel, r)
plus (x, y, z)
minus (x, y, z)
times (x, y, z)
power (x, i, z)
divI (x, i, z)
modI (x, i, z)
plusD (x, y, z)
minusD (x, y, z)
timesD (x, y, z)
divD (x, i, z)
modD (x, i, z)
```

ALICE

```
min (x, y, z)
max (x, y, z)
equal (x, y)
notequal (x, y)
distance (x, y, rel, z)
less (x, y)
lessEq (x, y)
greater (x, y)
greaterEq (x, y)
disjoint (x, i1, y, i2)
disjointC (x, i1, y, i2, c)
tasksOverlap (x, i1, y, i2, c)
distinct v
distinctOffset v
distinct2 v
atMost (x, v, i)
atLeast (x, v, i)
exactly (x, v, i)
element (x, v, z)
```


ALICE

```
conj (x, y, z)
disj (x, y, z)
exor (x, y, z)
nega (x, y)
impl (x, y, z)
equi (x, y, z)
Reified.fd (dom, c)
Reified.fdVec (n, dom, c)
Reified.card (i1, v, i2, c)
Reified.sum (v, rel, r, c)
Reified.sumC (v, rel, r, c)
Reified.sumAC (v, rel, r, c)
Reified.sumCN (v, rel, r, c)
Reflect.min x
Reflect.max x
Reflect.mid x
Reflect.nextLarger (x, i)
Reflect.nextSmaller (x, i)
Reflect.size x
```

```
Reflect.dom x
Reflect.domList x
Reflect.nbSusps x
Reflect.eq (x, y)
distribute (spec, v)
choose (spec, v)
```

3.2 Linear

Lineáris egyenlőtlenségekként engedi megfogalmazni a korlátokat. A lineáris korlátokat az FD-modul összeg-korlátjaivá fogalmazza át, nem feltétlenül optimális módon.

Mivel a Linear modul az állandó kifejezéseken nagymértékű egyszerűsítést végez, bizonyos esetekben előfordulhat, hogy túl akar lépni az egészek implementációtól függő határán, amely az FD-korlátokra is érvényes. Ilyen esetben az egyszerűsítés egy olyan FD-változó bevezetésével akadályozható meg, amelyhez egyetlen értéket rendelünk.

A bekezdés az Alice Manuel-ből angolul: Note also that since the linear module extensively performs folding of constant expressions, it eventually might exceed the implementation specific integer constant limit of finite domain constraints. In such a case, folding can be prevented by introducing a finite domain variable that is assigned a singleton value.

3.2.1 A `Linear` struktúra szignatúrája

```
signature LINEAR =
    sig
        infix 7 `*
        infix 6 `+ `-
        infix 5 `#
        infix 4 `= `<> `> `>= `< `<=
        infix 3 `<->

        datatype domain_element =
            `` of int
          | `# of int * int

        type domain = domain_element list

        datatype term =
            FD of FD.fd
          | `  of int
```

```
          | `+ of term * term
          | `- of term * term
          | `* of term * term

        datatype rel =
            `<   of term * term
          | `<=  of term * term
          | `=   of term * term
          | `<>  of term * term
          | `>=  of term * term
          | `>   of term * term
          | `<-> of rel * term

        val var : domain option -> term
        val bin : unit -> term
        val vec : int * domain -> term vector

        val distribute : FD.dist_mode * term vector -> unit
        val distinct : term vector -> unit
```



```
        val post : rel -> unit
    end
```

3.3 FS

Véges halmazváltozókat és azokra felvehető korlátokat tartalmaz. A véges halmazváltozó (finite set variable) olyan változó, amelynek az értéke a nemnegatív számok egy halmaza.

Lássunk egy példát:

```
open FS;
val x = FS.fs NONE;
val y = FS.fs NONE;
FS.disjoint (x,y);
FS.subset (x,y);
FS.Int.max (y,FD.fd (SOME #[RANGE (1,2)]));
FS.incl (FD.fd (SOME #[SINGLE (3)]),y);
FS.excl (FD.fd (SOME #[SINGLE (0)]),y);
```

3.3.1 Az `FS` struktúrában található korlátok

```
type fd
type bin = fd
type fs
exception Tell of {cause : exn}
inf
sup
fs spec
fsVec (n,spec)
compl (x,y)
compl (x,y)
complIn (x,y,z)
incl (x,y)
excl (x,y)
card (x,y)
cardRange (l,u,x)
isIn (i,x)
difference (x,y,z)
```



```
intersect (x,y,z)
intersectN (v,x)
union (x,y,z)
unionN (v,x)
subset (x,y)
disjoint (x,y)
disjointN v
distinct (x,x)
distinctN v
partition (v,x)
value x
emptyValue ()
singletonValue i
universalValue ()
isValue x
Int.min (x,y)
Int.max (x,y)
Int.convex x
Int.match (x,v)
```

```
Int.minN (x,v)
Int.maxN (x,v)
Reified.isIn (i,x,c)
Reified.areIn (is,x,cs)
Reified.incl (x,y,c)
Reified.equal (x,y,c)
Reified.partition (vs,is,x,cs)
Reflect.card x
Reflect.lowerBound x
Reflect.unknown x
Reflect.upperBound x
Reflect.cardOfLowerBound x
Reflect.cardOfUnknown x
Reflect.cardOfUpperBound x
```

3.4 A keresési tér megvalósításáról

A keresési feladatokat az ALICE végzi a `Space` struktúra segítségével, amely úgynevezett elsőrendű keresési tereket valósít meg.

Ezek használatával pl. egy mélységi keresés elvileg könnyen implementálható:

```
fun searchOne s =
  case Space.ask s of
      Space.FAILED          => NONE
    | Space.SUCCEEDED       => SOME (Space.merge s)
    | Space.ALTERNATIVES(n) =>
        let
            val c = Space.clone s
        in
          (Space.commit(s, Space.SINGLE 1)
          ;case searchOne s of
               NONE   => (Space.commit(c, Space.RANGE(2,n))
                          ;searchOne c
                          )
             | SOME s => SOME s
          )
        end;
```

Megjegyés: Az eredetileg az Alice Manual „The Space structure” c. fejezetében közölt `searchOne` függvényt szintaktikai hiba miatt nem lehet lefordítani. Az Alice az alábbi szekvenciális kifejezést véli rossznak, feltehetően hibás a fordítóprogram:

ALICE

```
in
   Space.commit(s, SINGLE 1);
   case searchOne s of
        NONE   => ...
end;
```

Ha a szekvenciális kifejezést a következő változatok egyikében írjuk föl, a fordítás sikerül:

```
in
   Space.commit(s, SINGLE 1); case searchOne s of
        NONE   => ...
end;
```

vagy

```
in
   Space.commit(s, SINGLE 1)
 ; case searchOne s of
        NONE   => ...
end;
```

vagy

ALICE

```
in
 (Space.commit(s, SINGLE 1)
 ;case searchOne s of
       NONE   => ...
 )
end;
```

Példa `searchOne` alkalmazására:

```
searchOne (Space.space money);
```

A `Search` struktúrában ennél kifinomultabb keresési módszereket is találhatunk.

3.4.1 A `Space` struktúra szignatúrája

```
signature SPACE =
sig
   eqtype 'a space

   datatype state =
       MERGED
```

```
          | FAILED
          | SUCCEEDED
          | ALTERNATIVES of int

        datatype verbose_state =
            VERBOSE_SUSPENDED of verbose_state
          | VERBOSE_MERGED
          | VERBOSE_FAILED
          | VERBOSE_SUCCEEDED_STUCK
          | VERBOSE_SUCCEEDED_ENTAILED
          | VERBOSE_ALTERNATIVES of int

        datatype choice =
            SINGLE of int
          | RANGE of int * int

        val space : (unit -> 'a) -> 'a space

        val ask : 'a space -> state
```



```
        val askVerbose : 'a space -> verbose_state
        val clone : 'a space -> 'a space
        val commit : 'a space * choice -> unit
        val inject : 'a space * ('a -> unit) -> unit
        val merge : 'a space -> 'a
        val kill : 'a space -> unit
        val waitStable : 'a space -> unit
    end
```

3.4.2 A függvények és típusok leírása

```
eqtype 'a space
```

The type of computation spaces.

```
datatype state = MERGED | FAILED | SUCCEEDED |
                 ALTERNATIVES of int
```

This datatype is used to communicate the state of a computation space.

```
datatype verbose_state = VERBOSE_SUSPENDED of verbose_state |
                         VERBOSE_MERGED |
                         VERBOSE_FAILED |
                         VERBOSE_SUCCEEDED_STUCK |
                         VERBOSE_SUCCEEDED_ENTAILED |
                         VERBOSE_ALTERNATIVES of int
```

This datatype is used to communicate the verbose state of a computation space.

```
datatype choice = SINGLE of int | RANGE of int * int
```

This datatype is used to select alternatives of the selected choice of a space.

```
space p
```

returns a newly created space, in which a thread containing an application of the unary function `p` to the root variable of the space is created.

```
ask s
```


waits until `s` becomes stable or merged and then returns the state of `s`.
If `s` is merged, `MERGED` is returned.
If `s` is stable and failed, `FAILED` is returned.
If `s` is stable and succeeded and there are no threads in `s` synchronizing on choices, `SUCCEEDED` is returned.
If `s` is stable and succeeded and there is at least one thread in `s` which synchronizes on a choice, `ALTERNATIVES i` is returned, where `i` gives the number of alternatives on the selected choice.
Synchronizes on stability of `s`.
Raises a runtime error if the current space is not admissible for `s`.

```
askVerbose s
```

returns the state of `s` in verbose form. Reduces immediately, even if `s` is not yet stable.
If `s` becomes merged, `VERBOSE_MERGED` is returned.
If `s` becomes suspended (that is, blocked but not stable), `VERBOSE_SUSPENDED t` is returned. `t` is a future that is bound to the verbose state of `s` when `s` becomes stable again.
If `s` is stable and failed, `VERBOSE_FAILED` is returned.
If `s` is stable and succeeded and there are no threads in `s` synchronizing on choices, either `VERBOSE_SUCCEEDED_STUCK`, or `VERBOSE_SUCCEEDED_ENTAILED` is returned. The

former happens when `s` still contains threads.
If `s` is stable and succeeded and there is at least one thread in `s` which synchronizes on a choice, `VERBOSE_ALTERNATIVES i` is returned, where `i` gives the number of alternatives on the selected choice.
Does not synchronize on stability of `s`.
Raises a runtime error if the current space is not admissible for `s`.

```
clone s
```

blocks until `s` becomes stable and returns a new space which is a copy of `s`.
Synchronizes on stability of `s`.
Raises a runtime error if `s` is already merged, or if the current space is not admissible for `s`.

```
commit (s, c)
```

blocks until `s` becomes stable and then commits to alternatives of the selected choice of `s`.
if `c` is `RANGE(l, h)`, then all but the `l,l+1,...,h` alternatives of the selected choice of `s` are discarded. If a single alternative remains, the topmost choice is replaced by this alternative. If no alternative remains, the space is failed.
`SINGLE i` is an abbreviation for `RANGE(i, i)`.

Synchronizes on stability of `s`.
Raises a runtime error, if `s` has been merged already, if there exists no selected choice in `s`, or if the current space is not admissible for `s`.

```
inject (s, p)
```

creates a thread in `s` which contains an application of `p` to the root variable of `s`.
Does not synchronize on stability of `s`.
Raises a runtime error if `s` is already merged, or if the current space is not admissible for `s`.

```
merge s
```

merges `s` with the current space and returns the root variable of `s`.
Does not synchronizes on stability of `s`.
Raises a runtime error if `s` is already merged, or if the current space is not admissible for `s`.

```
kill s
```

kills `s` by injecting a failure into a space.

```
waitStable s
```

Synchronizes on stability of `s` and returns unit.

3.5 A `Search` struktúra

A struktúrában többféle keresési eljárás van implementálva, és mindegyiknek több (újraszámítást használó stb.) változata is szerepel benne.

3.5.1 A `Search` struktúra függvényei

```
searchOne script
searchOneDepth (script,rcd)
searchOneDepthS (script,rcd)
searchOneBound (script,bound,rcd)
searchOneBoundS (script,bound,rcd)
searchOneIter (script,rcd)
searchOneIterS (script,rcd)
searchOneLDS (script,m)
searchOneLDSS (script,m)
searchAll script
searchAllDepth (script,rcd)
searchAllDepthS (script,rcd)
```



```
searchBest (script,order)
searchBestBAB (script,order,rcd)
searchBestBABS (script,order,rcd)
searchBestRestart (script,order,rcd)
searchBestRestartS (script,order,rcd)
```

References

[1] The ALICE manual, 2004.10.05, *http://www.ps.uni-sb.de/alice/*

[2] Constraintprogrammierung, Niko Paltzer, 2004.04.05, *http://www.ps.uni-sb.de/courses/seminar-ws03/ConstraintProgrammierung.pdf*