Remember from last lecture combinatorial structure, combinatorial problem; constraint satisfaction problem, solution to a CSP; constraint store, propagator, branching, search tree; branch and bound search

A useful metaphor Propagation and branching is like proving a theorem: Propagation means applying proof rules. Branching is like doing case distinctions. Case distinctions are only heuristics, but good case distinctions pay off.

This lecture In this lecture, I want to give you an idea how to model CSPs on finite-domain integer variables using Alice and GeCoDE. To this end, I will present a number of constraint satisfaction problems, show how to model them in Alice, and point out important aspects that you often will encounter when modelling CSPs: local versus global constraints, defined constraints, symmetries and how to eliminate them, modelling languages, redundant constraints.

Send More Money State the problem and write down the constraint program.

```
import structure FD from "x-alice:/lib/gecode/FD"
import structure Linear from "x-alice:/lib/gecode/Linear"

open Linear

fun smm space =
  let
    val letters as #[s, e, n, d, m, o, r, y] = fdtermVec (space, 8, [0`#9])
  in
    distinct (space, letters, FD.BND);
    post (space, s `<> `0, FD.BND);
    post (space, m `<> `0, FD.BND);
    post (space,                `1000`*s `+ `100`*e `+ `10`*n `+ d
                 `+             `1000`*m `+ `100`*o `+ `10`*r `+ e
                 `= `10000`*m `+ `1000`*o `+ `100`*n `+ `10`*e `+ y, FD.BND);
    branch (space, letters, FD.B_SIZE_MIN, FD.B_MIN);
    {s, e, n, d, m, o, r, y}
  end
```

Note to self: Explain the individual parts: first the modelling itself, then the implementation in Alice/GeCoDE. Import the modules. Open `Linear` to get the identifiers available at top-level. Type of a constraint script (`space` → α). Notion of interface. Individual constraints. Forward pointer to explanation of `FD.BND` in *coming lectures. Specification of branching strategies (which variable to pick, how to split the domain; see documentation for available strategies).*

Send Most Money

Note to self: Start with the Send More Money problem.

For best-solution search, the type of the script changes to
`space` $\times$ (`space` $\times$ `space` $\rightarrow$ `unit`). Our task is to implement the betterness constraint. We want to write something like

```
fun better (current, lastSolution) =
   post (current, {MONEY in current} `> {MONEY in lastSolution}, FD.BND)
```

Because we post this in the current constraint, the value of `MONEY` in `current` simply is `MONEY` itself:

```
fun better (current, lastSolution) =
   post (current, MONEY `> {MONEY in lastSolution}, FD.BND)
```

For the value of `MONEY` in `lastSolution`, we need to use reflection:

```
fun better (current, lastSolution) =
   post (current, MONEY `> `(FD.Reflect.value(lastSolution, MONEY)), FD.BND)
```

But what is `MONEY`, really?

```
val money' = FD.range (space, (10234, 98765))
val moneyTerm = FD(money')
post (space, moneyTerm `= `10000`*m' `+ `1000`*o' `+ `100`*n' `+ `10`*e' `+ y', FD.BND)
```

Thus we finally get

```
fun better (current, lastSolution) =
   post (current, moneyTerm `> `(FD.Reflect.value(lastSolution, money')), FD.BND)
```

Some Background What about those backticks? Alice provides two structures ('modules') for writing constraint scripts on finite-domain variables: `FD` and `Linear`. The functions in `FD` are directly connected to the functions in the GeCoDE library; in this sense, `FD` is quite low-level. The `Linear` structure adds a more high-level view for a particularly frequent class of constraints: linear equations, inequations and disequations. These constraints have the following form, where $\sim$ is a simple relation such as equality, greater-than, not equal:

$$\sum_{1 \le i \le n} c_i \cdot x_i \sim c$$

This constraint can be enforced using `FD.linear`, which takes as its inputs a space, a vector of coefficient-variable pairs, a relation, a constant, and a consistency level. (Forget about the consistency level at the moment.) For example, the constraint $5x + 42 = y$ can be enforced in `space` by the constraint

```
FD.linear (space, #[(5, x), (~1, y)], FD.EQ, ~42, FD.BND)
```

However, it is more convenient to import `Linear`, open it (such that all functions and operators become available at top-level) and write

```
post (space, `5`*x `+ `42 `= `y, FD.BND)
```

All constraints we have encountered in Send More (Most) Money have this form, so therefore we used the `Linear` module.

Global constraints – Queens Problem specification: place 8 queens on an 8×8 chess board such that no two queens attack each other.

Note to self: Show a solution for the 8 queens problem. Show where the constraints come from. Show the sample solution. Iteration is an important ingredient in successful modelling.

```
fun loop i n f = if i >= n then nil else f i :: loop (i + 1) n f

fun upperTriangle n =
   List.concat (loop 0 n (fn i => loop (i + 1) n (fn j => (i, j))))

fun queens n space =
   let
      val row = Linear.fdtermVec (space, n, [0‘#(n - 1)])
   in
      Linear.distinct (space, row, FD.BND);
      List.app (fn (i, j) =>
         let
            val rowi = Vector.sub (row, i)
            val rowj = Vector.sub (row, j)
         in
            post (space, rowi ‘+ (‘j ‘- ‘i) ‘<> rowj, FD.BND);
            post (space, rowi ‘- (‘j ‘- ‘i) ‘<> rowj, FD.BND)
         end) (upperTriangle n);
      Linear.branch (space, row, FD.B_SIZE_MIN, FD.B_MED);
      row
   end
```

Note to self: Show how to parametrise the problem over n. Iteration is even more important now.

When problem sizes get big, the modelling that we have chosen turns out to be bad: we post a quadratic number of constraints on the problem variables, which leads to a quadratic number of propagators. Better solution: `distinctOffset`.

`FD.distinctOffset (`*`space`*`, [(c`$_1$`, x`$_1$`), ..., (c`$_n$`, x`$_n$`)])`

forces the sums $x_i + c_i$ to be pairwise distinct.

```
fun queens n space =
  let
    val row = FD.rangeVec (space, n, (0, n - 1))
    val add = Vector.tabulate (n, fn i => 0 + i)
    val sub = Vector.tabulate (n, fn i => 0 - i)
  in
    FD.distinct (space, row, FD.BND);
    FD.distinctOffset (space, VectorPair.zip (add, row), FD.BND);
    FD.distinctOffset (space, VectorPair.zip (sub, row), FD.BND);
    FD.branch (space, row, FD.B_SIZE_MIN, FD.B_MED);
    row
  end
```

Defined constraints Modelling often involves composing several primitive constraints from the library into more high-level constraints better suited to the problem. In Alice, we do not have to go far.

- We want to define n-ary sum constraints in terms of primitive constraints from the FD library: $\sum_{1 \le k \le n} x_k \sim y$ and $\sum_{1 \le k \le n} x_k \sim c$. This is straightforward; reduce to `FD.linear`.
- Constraints for n-ary products are more complicated: $\prod_{1 \le k \le n} x_k = y$ and $\prod_{1 \le k \le n} x_k = c$. The problem is that FD only supports (binary) `FD.mult`, so we need to create a new variable for each individual product and constrain it appropriately. Propagation will be quite inefficient because of that. (Compare that to `distinctOffset` in the Queens example.)

Custom constraint languages and symmetries – Grocery When we model a problem using primitive constraints and defined constraints, we effectively define a custom constraint language: to model this problem, we need these and those constraints. Making this language explicit sometimes helps to better understand the problem domain, and to modularise the problem solving. I will illustrate that using a very simple example.

Problem specification: A kid enters a grocery store and buys four items. The cashier charges \$7.11, the kid pays and is about to leave when the cashier calls the kid back, and says “Hold on, I multiplied the four items instead of adding them; I’ll try again—hah, with adding them, the price still comes to \$7.11”. What were the prices of the four items?

Two constraints: sum and product (we can use the defined constraints from the previous example).

Note to self: Define constraint language. Define evaluation function. Show the modelling.

Unfortunately, when feeding this, it will take hours. The problem is that the prices can be ordered in any possible permutation, so we get a factorial blow-up. We need to eliminate those symmetries.

Note to self: Extend the proof metaphor: symmetry elimination is like 'without loss of generality'.

Redundant constraints If time permits: the Pythagoras example. How many triples (a, b, c) exist such that $a^2 + b^2 = c^2$ and $a \leq b \leq c$? The script creates a propagator for a redundant constraint, which will not affect the size of the search tree, but reduce the time for propagation. (Unfortunately, this cannot currently be measured using the tools we have.)