Introduction to Narrowing

Main concepts of this unit:

> Variables
> > - existential, universal
>
> Residuation
> > - rigid function
>
> Narrowing
> > - flexible function
> > - non-determinism
>
> Strict Equality
>
> Programming with Equations
>
> Properties of Narrowing
> > - soundness, completeness

Variables

Variables in terms occurs in two situations. In a rewrite rule, a variable stands for any term. This use is like universal quantification. E.g.:

```
pop (push _ S) = S
```

is intended as: "for any element E and stack S, the result of *popping* $push(E, S)$ is S".

Functional logic programming supports the evaluation of expressions containing variables. Variables in expressions differ from variables in rules. They stand for some unknown value. This use is like existential quantification. E.g.:

```
data Suit = Club | Spade | ...
data Rank = Ace | King | ...
data Card = Card Rank Suit
rank (Card r _) = r
```

The following Curry code, which finds whether a poker hand is a *four-of-a-kind*, contains unknown variables: x, $\_$, z and r.

```
isFour (x++[_]++z)
    | map rank (x++z) == [r,r,r,r]
    = Just r where r free
isFour'default _ = Nothing
```

This unit's focus is to evaluate expressions of this kind.

Residuation

The problem is the "functional" evaluation of expressions (terms) containing uninstantiated variables. The functions are defined by rewrite rules.

One approach, called ==residuation,== assumes that some special functions, e.g., in Escher the equality operator, may instantiate variables to which are applied. Other functions wait (residuate) when applied to variables. The control is transfered to another expression hoping that the evaluation of this expression will instantiated the variables in question. E.g., in Curry:

```
digit = 0
digit = 1
digit = 2
...
test x y = x+x=:=y & x*x=:=y & digit=:=x
```

Suppose the expression to evaluate is `test X Y`, where the arguments are both uninstantiated variables.

Both "`+`" and "`*`" residuate, but "`=:=`" does not. The variable `x` is instantiated by the last equation, which in turn restarts the evaluation of the residuating expressions.

Narrowing

As for residuation, the problem is still the "functional" evaluation of expressions (terms) containing uninstantiated variables. The functions are defined by rewrite rules.

Another approach, called ==narrowing,== guesses values for the variables when this is necessary to keep the computation going. Of course, some guesses may be wrong. A guess comes from a pattern found in a rewrite rule of the function applied to variables. E.g., in the Curry's *prelude*:

```
length [] = 0
length (_:xs) = 1 + length xs
```

The evaluation of:

```
Prelude> length l where l free
```

gives:

```
{l=[]} 0
{l=[_a]} 1
{l=[_a,_b]} 2
...
```

Most functions in Curry behave in this way. Typical exceptions are arithmethic operations and I/O primitives.

Equality

Curry predefines a polymorphic <mark>Boolean equality,</mark> "==", to tell whether two expressions *are equal*.

```
data Color = Red | Green | Blue
test1 = Red == Red
test2 = Red == Green
test3 = Red == x where x free
```

As expected *test1* is `True`, *test2* is `False`, and *test3* produces 3 values, one for each color.

The expression e_1 `==` e_2 evaluates to `True` when e_1 and e_2 evaluate to a same ground data expression, to `False` when they evaluate to different data expressions.

Equality between a number or character and an unbound variable suspends, because there are too many (or infinite) cases to consider. E.g.:

```
x==2 where x free
```

Likewise, equality between two unbound variables suspends.

Exercise 5. Forget the predefined equality. Define your own equality for *lists* of colors. E.g. the following must return `False`.

```
[Red,Blue] `myeqlofc` [Red,Green]
```

Equations

An <mark>equation</mark> is an expression of the form $e_1 = e_2$ and typically containing free variables. <mark>Solving</mark> the equation consists in finding values for the free variables that evaluate the expression to `True`.

Curry predefines a polymorphic <mark>constrained equality,</mark> "=:=", to *solve equations*. Loosely speaking, compute only the positive outcomes "==".

Examples 6.

Compute a permutation of a list:

```
permute [] = []
permute (x:xs) | permute xs =:= u ++ v
               = u ++ x:v
               where u, v free
```

Pick a duplicate element in a list

```
pickdupl l | l =:= a++d:b++d:c
           = d
           where a,b,c,d free
```

Find the distance between two given elements in a list:

```
distance l i j | l =:= a++i:b++j:c
               = length b + 1
               where a,b,c free
```

Soundness and Completeness

Narrowing solves equations involving user-defined data types. Let e be an equation and $v_1, \ldots, v_n$ uninstantiated variables in e and $\sigma = \{v_1 \mapsto t_1, \ldots, v_n \mapsto t_n\}$ a substitution.

Narrowing is <mark>sound:</mark> if $e \overset{*}{\rightsquigarrow} True$ and computes σ, then $\sigma(e) \overset{*}{\rightarrow} True$ as well, i.e., (σ is a solution of e).

Narrowing is <mark>complete:</mark> if $\sigma(e) \overset{*}{\rightarrow} True$ (σ is a solution of e), then there exists a narrowing evaluation of e to $True$ that computes a substitution at least as general as σ.

The (operational) completeness of narrowing is not trivial to obtain. Many implementations of narrowing, including the one provided by Pakcs, are incomplete.

Narrowing is able to find all the solutions of an equation. If the number of solutions is finite, narrowing may or may not find them in finite time.

Exercise 7. Invoke the Packs interpreter and try to compute repeated solutions of both:

```
X =:= [1,2,3]
length X =:= length [1,2,3]
```

How do you interpret the behavior of Pakcs?

Narrowing Computations

Example 8. The following example shows how narrowing computes.

```
data Nat = Zero | Succ Nat

leq Zero _ = True
leq (Succ _) Zero = False
leq (Succ x) (Succ y) = leq x y
add Zero y = y
add (Succ x) y = Succ (add x y)
```

Trace the evaluation of `leq (add X Y) Y =:= True`.

```
- leq (add X Y) Y
- - add X Y                guess Zero for X
- - add Zero Y
- - Y
- leq Y Y                  guess Zero for Y
- leq Zero Zero
- True                     {X ↦ Zero, Y ↦ Zero}
```

find new solution, try other last guess

```
- leq Y Y                  guess (Succ U) for Y
- leq (Succ U) (Succ U)
- - leq U U                guess Zero for U
- - leq Zero Zero
- - True
- True                     {X ↦ Zero, Y ↦ Succ Zero}
```

Exercise 8. Which other solutions will Pakcs find? Is this OK?

Programming

The problem is a portion of the *Game of 24*, a puzzle for children (and CS students).

> Given four 1-digit positive integers find an arithmetic expression that evaluates to 24 in which each digit occurs exactly once. A number can be divided only by its factors. For example, a solution of the problem [2,3,6,8] is (2+8)*3-6. There are 26 distinct solutions of this problem.

A design pattern for search problems of this kind is called *generate and test*. Its application consists of: (1) generating a well-formed expression of the problem, e.g., (2+3)*(6+8), and testing whether it evaluates to 24 (no for the given example).

Exercise 9. Code a Curry program that generates any (all) well-formed expressions of a problem.

Hint: a plausible type for the expressions:

```
data Exp = Num Int
         | Add Exp Exp
         | Mul Exp Exp
         | Sub Exp Exp
         | Div Exp Exp
```

Programming cont'd

The problem is to compute the day schedule of a league of sport teams. A *day schedule* is a set of games (between two teams), where (1) every team, or every team but one if number of teams is odd, plays daily and (2) no team plays more than one game per day.

Exercise 10a. Code a Curry program that computes a day schedule for a league of sport teams.

Hints: a non-deterministic solution picks two teams of the league for a game and recurs on the remaining teams. There are two challenges: (1) ensuring that every schedule is generated, and (2) avoiding or minimizing repeated computations of the same schedule.

Exercise 10b. How many different schedules are there? Extend the previous program to compute the *set* of day schedules (use the SetFunctions library).