Strategies

Main concepts of this unit:

- Narrowing Step
  - narrex
- Subsumption Ordering
- Definitional Trees
  - leaf and branch patterns
  - inductive position
- Inductive sequentiality
- Strategies
  - needed narrowing
- Program classes
  - conditions, overlapping

Narrowing Step

Let t be a term, $l \to r$ a rule, p a non-variable position of t, and σ a substitution such that $\sigma(l) = \sigma(t|_p)$, i.e., l and $t|_p$ unify. The subterm of t at position p is a *narrex.*

A *narrowing step* is a pair of terms $t \to \sigma(t[r]_p)$, where the latter denotes the term obtained by replacing the subterm of $\sigma(t)$ at position p with $\sigma(r)$.

Example 2. Consider the following TRS:

```
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)
```

Let

$$t = \texttt{leq (add X Y) Y},$$
$$l \to r = \texttt{add Zero y = y},$$
$$p = \langle 1 \rangle,$$
$$\sigma = \{\texttt{X} \mapsto \texttt{Zero}, \texttt{y} \mapsto \texttt{Y}\}.$$

Then

$$\texttt{leq (add X Y) Y} \rightsquigarrow_{\langle l \to r, p, \sigma \rangle} \texttt{leq Y Y}$$

The problem is choosing $l \to r$, p, and σ for a term t.

Strategy

A strategy selects the rule, position, and unifier of a step. Formally, a *strategy* is a mapping from a term to a set of steps (triples). A naive strategy tries all possible steps with most general unifiers.

Efficient strategies compute only a subset of all possible steps of a term and forgo most general unifiers. Different strategies exist for different classes of TRS, e.g., confluent, constructor based, etc. We look at a strategy for constructor-based TRS.

All modern strategies for functional logic computations (narrowing) are based, directly or indirectly, on a hierarchical organization of the lhs of the rewrite rules of each function of a program. This structure is called a *definitional tree*.

A definitional tree is a set of terms (partially) ordered by *subsumption.* Given two terms, t and u, we write $t \leqslant u$ and say that t *preceeds* u, if there exists a substitution σ such that $\sigma(t) = u$, i.e., u is an instance of t.

Examples 3. (variable are in upper case)

$X \leqslant 0$

$X \leqslant Y$ and $Y \leqslant X$

$X \texttt{++} Y \leqslant [\,] \texttt{++} Y$

$(X{:}Xs) \texttt{++} Y \not\leqslant [\,] \texttt{++} Y$ and $[\,] \texttt{++} Y \not\leqslant (X{:}Xs) \texttt{++} Y$

Definitional Tree

A *definitional tree* of an operation f is a finite, non-empty set $\mathcal{T}$ of linear *patterns* partially ordered by subsumption and having the following properties up to renaming of variables:

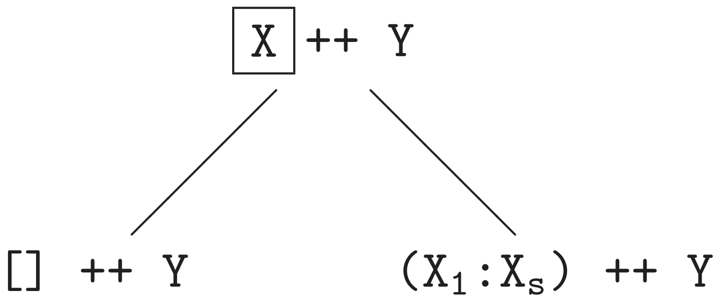
- [leaves property] The maximal elements, referred to as the *leaves,* of $\mathcal{T}$ are all and only variants of the left hand sides of the rules defining f. Non-maximal elements are referred to as *branches.*
- [root property] The minimum element, referred to as the *root,* of $\mathcal{T}$ is $f(X_1, \ldots, X_n)$, where $X_1, \ldots, X_n$ are fresh, distinct variables.
- [parent property] If π is a pattern of $\mathcal{T}$ different from the root, there exists in $\mathcal{T}$ a unique pattern π' strictly preceding π such that there exists no other pattern strictly between π and π'. π' is referred to as the *parent* of π and π as a *child* of π'.
- [induction property] All the children of a same parent differ from each other only at the position, referred to as *inductive,* of a variable of their parent.

Examples are in the next page . . .

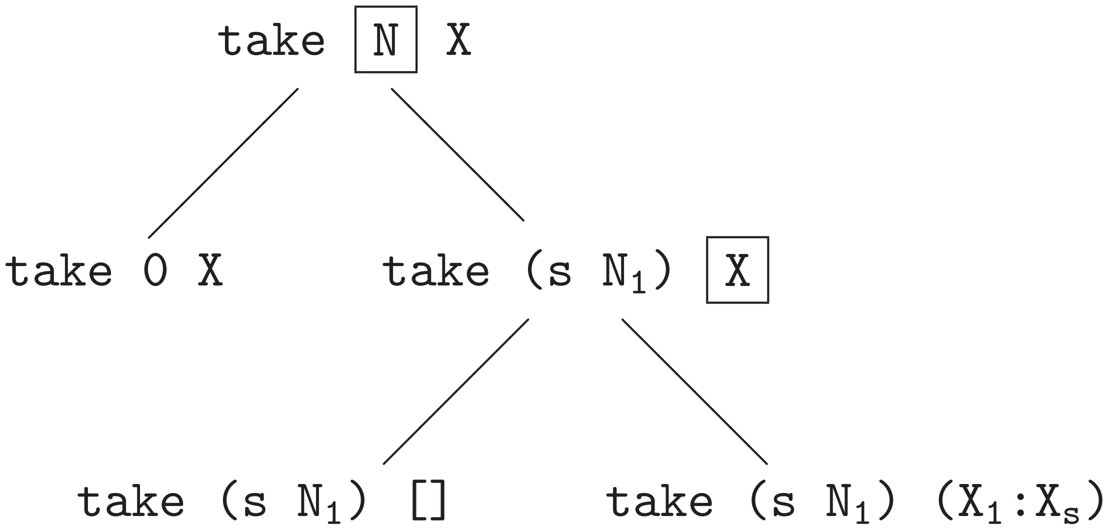
Examples

Examples 5. Some operations with their definitional trees. The *inductive variable* is boxed.

```
[] ++ Y = Y
(X:Xs) ++ Y = X : Xs++Y
```



```
take 0 _ = []
take (s N) [] = []
take (s N) (X:Xs) = X : take N Xs
```



Inductive Sequentiality

An operation is *inductively sequential* if it has a definitional tree. A program (TRS) is inductively sequential if all its operations are inductively sequential.

Each non-value expression of such a program having a *value* also has a step, called *needed,* that **must** be executed to compute the value.

Every (first-order) Haskell program is inductively sequential with the conventional reading of rules from top to bottom.

Inductively sequential programs are confluent. Some Curry programs, even confluent ones, are **not** inductively sequential, e.g.:

```
infixl 2 \/
True \/ _ = True
_ \/ True = True
False \/ False = False
```

PAKCS approximates the execution of the above operation.

Exercise 6. Prove that the operations of Example 2 are inductively sequential. Prove that "`\/`" defined above is not inductively sequential.

Needed Narrowing

Narrowing steps in inductively sequential programs are computed by the *needed narrowing* strategy.

Let $t = f(t_1, \ldots, t_k)$ be an operation-rooted term to narrow. We most-generally unify t with some non-deterministically chosen maximal pattern π in a definitional tree $\mathcal{T}$ of f. Let η be a most general unifier of t and π. If π is a leaf of $\mathcal{T}$, $\eta(t)$ is a redex and we replace it. If π is a branch of $\mathcal{T}$, we consider the subterm u of $\eta(t)$ at the inductive position of π. The term u cannot be a variable. If u is operation-rooted, we recursively attempt to narrow it. If u is constructor-rooted, we fail, since $\eta(t)$ cannot be narrowed to a value.

Since there can be many maximal patterns π that unify with t, distinct steps can be computed on t, i.e., the above definition is non-deterministic.

Note that the unifier of a step computed by needed narrowing is **not** necessarily most general. Without this condition, some narrowing steps are useless.

Needed narrowing is sound, complete and, for computations to a value, it computes only *unavoidable* steps and *disjoint* substitutions.

Example

Compute the needed steps of $t =$ `take N ([1]++[2])`, where `N` is an uninstantiated variable.

The term t unifies with both `take 0 X`, which is a leaf, and `take (s N`$_1$`) X`, which is a branch. The first is obviously a maximal element in its tree, since it is a leaf. The second is maximal as well, since t does not unify with either of its children. Therefore, needed narrowing computes the two steps shown below.

The step with the leaf has unifier $\{N \mapsto 0\}$:

$$\texttt{take N ([1]++[2])} \rightsquigarrow_{\Lambda,\{N\mapsto 0\}} \texttt{[]}$$

The step with the branch has unifier $\{N \mapsto (s\ N_1)\}$. The inductive position is 2 (counting from 1):

$$\texttt{take N ([1]++[2])} \rightsquigarrow_{2,\{N\mapsto (s\ N_1)\}} \texttt{take (s N}_1\texttt{) (1:[]++[2])}$$

Exercise 8.

- Verify that the inner step (at position 2) of above step is computed by needed narrowing.
- Verify that the above step could be computed with a more general unifier.
- Verify that executing the above step with a most general unifier may be useless (difficult).

Program Classes

Inductively sequential programs are too restrictive for functional logic programming. Two larger classes have been proposed for FLP.

Constructor-based, conditional programs: no restrictions except the constructor discipline.

Constructor-based, left-linear programs: no restrictions except the linearity of the lhss.

```
insert e xs = e:xs
insert e (x:xs) = x:insert e xs
```

Overlapping inductively sequential programs: the lhss of an operation have a definitional tree; distinct rhss are allowed for a single lhs.

```
insert e xs = e:xs
insert e xs = neins e xs
neins e (x:xs) = x:insert e xs
```

Every (first-order) program in the first two classes can be transformed (syntactically) into a program of the third class.

A strategy for the overlapping inductively sequential programs is very similar to needed narrowing: in addition to the other non-deterministic choices, non-deterministically pick one of the rhss, if many are available.