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, $l \rightarrow r$ a rule, $p$ a non-variable position of $t$, and $\sigma$ 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 $t \rightarrow \sigma(t[r]_p)$, 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:

\[
\begin{align*}
\text{data Nat} & = \text{Zero} \mid \text{Succ Nat} \\
\text{leq Zero _} & = \text{True} \\
\text{leq (Succ _)} \text{ Zero} & = \text{False} \\
\text{leq (Succ x)} \text{ (Succ y)} & = \text{leq x y} \\
\text{add Zero y} & = \text{y} \\
\text{add (Succ x) y} & = \text{Succ (add x y)}
\end{align*}
\]

Let

\[
\begin{align*}
t & = \text{leq (add X Y)} \text{ Y}, \\
l & \rightarrow r = \text{add Zero y} = \text{y}, \\
p & = \langle 1 \rangle, \\
\sigma & = \{ X \mapsto \text{Zero}, y \mapsto Y \}.
\end{align*}
\]

Then

\[
\text{leq (add X Y)} \text{ Y} \sim_{\langle l \rightarrow r, p, \sigma \rangle} \text{leq Y Y}
\]

The problem is choosing $l \rightarrow r$, $p$, and $\sigma$ 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 Haskell-like 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 \( t \leq u \) and say that \( t \) precedes \( u \), if there exists a substitution \( \sigma \) such that \( \sigma(t) = u \), i.e., \( u \) is an instance of \( t \).

Examples 3. (variable are in upper case)

\[
\begin{align*}
X & \leq 0 \\
X & \leq Y \quad \text{and} \quad Y \leq X \\
X ++ Y & \leq [] ++ Y \\
(X : Xs) ++ Y & \nleq [] ++ Y \quad \text{and} \quad [] ++ Y \nleq (X : Xs) ++ Y
\end{align*}
\]
Definitional Tree

A definitional tree of an operation \( f \) is a finite, non-empty set \( T \) 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 \( T \) 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 \( T \) is \( f(X_1, \ldots, X_n) \), where \( X_1, \ldots, X_n \) are fresh, distinct variables.

- **[parent property]** If \( \pi \) is a pattern of \( T \) different from the root, there exists in \( T \) a unique pattern \( \pi' \) strictly preceding \( \pi \) such that there exists no other pattern strictly between \( \pi \) and \( \pi' \). \( \pi' \) is referred to as the parent of \( \pi \) and \( \pi \) as a child of \( \pi' \).

- **[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.

\[
\begin{align*}
[] ++ Y &= Y \\
(X:Xs) ++ Y &= X : Xs++Y
\end{align*}
\]

\[
\begin{array}{c}
X \\
\end{array} \\
\begin{array}{c}
[] ++ Y \\
(X_1:X_s) ++ Y
\end{array}
\]

\[
\begin{align*}
take 0 \_ &= [] \\
take (s \: N) [] &= [] \\
take (s \: N) (X:Xs) &= X : take N Xs
\end{align*}
\]

\[
\begin{array}{c}
take N X \\
\end{array} \\
\begin{array}{c}
take 0 X \\
take (s \: N_1) X
\end{array}
\]

\[
\begin{align*}
take (s \: N_1) [] & \\
take (s \: N_1) (X_1:X_s)
\end{align*}
\]
Inductive Sequentiality

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

Each expression of such a program (term in the TRS) having a \textit{value} also has a step, called \textit{needed}, that \textbf{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 \textbf{not} inductively sequential, e.g.:

\begin{verbatim}
  infixl 2 \/
  True \/ _ = True
  _ \/ True = True
  False \/ False = False
\end{verbatim}

\textsc{Pakcs} does not execute the above operation correctly.

\textbf{Exercise 6.} Are the operations of Example 2 inductively sequential? Why?
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 $\pi$ in a definitional tree $\mathcal{T}$ of $f$. Let $\eta$ be a most general unifier of $t$ and $\pi$. If $\pi$ is a leaf of $\mathcal{T}$, $\eta(t)$ is a redex and we replace it. If $\pi$ is a branch of $\mathcal{T}$, we consider the subterm $u$ of $\eta(t)$ at the inductive position of $\pi$. 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 $\pi$ 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** 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 $t = \text{take } N ([1]++[2])$, where $N$ is an uninstantiated variable.

The term $t$ unifies with both $\text{take } 0 \ X$, which is a leaf, and $\text{take } (s \ N_1) \ X$, 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 $\{N \mapsto 0\}$:

$$\text{take } N ([1]++[2]) \triangleright_{A,\{N \mapsto 0\}} \ []$$

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

$$\text{take } N ([1]++[2]) \triangleright_{2,\{N \mapsto (s \ N_1)\}}$$

$$\text{take } (s \ N_1) (1:[]++[2])$$

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.

\[
\begin{align*}
\text{insert } e \; &\; \text{xs} = e : \text{xs} \\
\text{insert } e \; &\; (x : \text{xs}) = x : \text{insert } e \; \text{xs}
\end{align*}
\]

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

\[
\begin{align*}
\text{insert } e \; &\; \text{xs} = e : \text{xs} \\
\text{insert } e \; &\; \text{xs} = \text{neins } e \; \text{xs} \\
\text{neins } e \; &\; (x : \text{xs}) = x : \text{insert } e \; \text{xs}
\end{align*}
\]

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.