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 $\sigma(t) = \sigma(t_p)$, 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 $t \rightarrow \sigma(t[r])$, 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:

```haskell
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 $t = \text{leq (add }X Y\text{) Y}$,

$\rightarrow r = \text{add Zero }y = y$.

$p \equiv (1)$,

$\sigma \equiv \{X \mapsto \text{Zero}, y \mapsto Y\}$.

Then

$\text{leq (add }X Y\text{) Y} \leftrightarrow (l \rightarrow r, p, \sigma), \text{leq Y Y}$

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

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.

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

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

\[
\begin{align*}
\text{take } (s \ N_1) \ [] &= [] \\
\text{take } (s \ N_1) \ (X_1 : Xs) &= X_1 : \text{take } N_1 \ Xs
\end{align*}
\]

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 non-value expression of such a program having a value also has a step, called \textit{needed}, that must be executed to compute the value.

Every (first-order) Haskell program is inductively sequential. Some Curry programs, even confluent ones, are not inductively sequential, e.g.:

\[
\begin{align*}
\text{infixl 2 } \vee \\
\text{True } \vee _{} &= \text{True} \\
_{} \vee \text{True} &= \text{True} \\
\text{False } \vee \text{False} &= \text{False}
\end{align*}
\]

\textsc{Pakcs} approximates the execution of the above operation.

Exercise 6. Prove that the operations of Example 2 are inductively sequential. Prove that “\text{\textbackslash{}\textbackslash{}}” defined above is not inductively sequential.

Needed Narrowing

Narrowing steps in inductively sequential programs are computed by the \textit{needed narrowing} strategy.

Let \( t = f(t_1, \ldots, t_k) \) be an operation-rooted term to narrow. We most-generally attempt to narrow \( t \) with some non-deterministically chosen maximal pattern \( \pi \) in a definitional tree \( T \) of \( f \). Let \( \eta \) be a most general unifier of \( t \) and \( \pi \). If \( \pi \) is a leaf of \( T \), \( \eta(t) \) is a redex and we replace it. If \( \pi \) is a branch of \( T \), 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 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 \textit{unavoidable} steps and \textit{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 \( \{ N \mapsto 0 \} \):

\[
\text{take } N \ ((1: 2)) \xrightarrow{\Lambda, \{ N \mapsto 0 \}} []
\]

The step with the branch has unifier \( \{ N \mapsto (s \ N_1) \} \).

The inductive position is 2 (counting from 1):

\[
\begin{align*}
\text{take } N \ ((1: 2)) &\xrightarrow{\Lambda, \{ N \mapsto (s \ N_1) \}} \text{take } (s \ N_1) \ (1: 1: 2)
\end{align*}
\]

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 lhs.

```latex
\text{insert } e \text{ xs } = e:xs \\
\text{insert } e \text{ (x:xs) } = x:\text{insert } e \text{ xs }
```

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

```latex
\text{insert } e \text{ xs } = e:xs \\
\text{insert } e \text{ xs } = \text{neins } e \text{ xs} \\
\text{neins } e \text{ (x:xs) } = x:\text{insert } e \text{ 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 rhs, if many are available.