Introduction to Rewriting

Main concepts of this unit:
- **Signature**
  - variable
  - symbol, arity
- **Term**
  - root, arguments
  - ground
- **Position**
- **Substitution**
  - domain, range
- **Rule**
- **Rewrite System**
  - redex, normal form
  - step
  - constructor, many sorted
- **Confluence**
- **Termination**
  - strong, weak
- **Orthogonality**
  - linearity, ambiguity

Rewriting is a model of computation. It has been used in theorem proving and to abstract the execution of programs in declarative languages.

In this lecture, you will learn this model of computation. Several preliminary concepts are necessary, in particular, term and rewriting.

An **alphabet** or **signature**, denoted \( \Sigma \), consists of:
- A denumerable set of **variables** \( x_0, x_1, x_2, \ldots \) also denoted \( x, y, z, x', y', \ldots \)
- A non-empty, finite set of **symbols** \( f, g, h, \ldots \) also denoted by expressive names such as \( \text{pop}, \text{push}, +, 0, \ldots \) Each symbol has an **arity**, a natural number intended as the number of arguments a symbol takes.

**Example 2.** The symbols of the signature and their arities are:
- \( z \) 0
- \( s \) 1
- \( a \) 2
- \( m \) 2

A term or **expression** over a signature \( \Sigma \), denoted \( \text{Ter}(\Sigma) \), is defined inductively as follows:

- \( x_0, x_1, x_2, \ldots \in \text{Ter}(\Sigma) \)
- If \( f \) is a symbol of arity \( n, n \geq 0 \), and \( t_1, t_2, \ldots, t_n \in \text{Ter}(\Sigma) \), then \( f(t_1, t_2, \ldots, t_n) \in \text{Ter}(\Sigma) \).

In the second case, \( f \) is called the **root** of \( t_1, t_2, \ldots, t_n \) and \( t_1, t_2, \ldots, t_n \) are called the **arguments** of \( f \).

Terms without variables are called **ground** terms.

**Example 3.** A few terms:
- \( z \)
- \( x \)
- \( s(z) \)
- \( a(s(z), m(s(z), s(z))) \)
- \( a(z, z) \)

**Exercise 3.** Code a Curry program for representing terms. Consider two options: (simple) represent only ground terms of the signature implied by Example 2; (harder) represent terms of any arbitrary signature.

Terms are naturally represented as, and sometimes called, trees. The tree representation of the term \( a(s(z), z) \) is:

```
  a
 / \   /
s  z  z
 / \   /
z  z   z
```

A position is a sequence of natural numbers defining a path in a tree, therefore identifying a subterm of a term.

**Example 4.** The position \( \langle 1 \cdot 1 \rangle \) identifies the leftmost \( z \) in the above term, \( \langle 2 \rangle \) identifies the rightmost \( z \).

The positions in the above term are:

```
\langle \rangle
\langle 1 \rangle
\langle 2 \rangle
\langle 1 \cdot 1 \rangle
```

Brackets and dots separating the numbers in a position are often omitted. The empty position is also denoted by \( \Lambda \).
Position cont’d

The subterm of a term \( t \) at the position or occurrence \( p \), denoted \( t|_p \), is defined inductively as follows:

\[
\begin{align*}
\forall t \quad t|_\epsilon &= t, \\
\forall t_1, t_2, \ldots, t_n \quad f(t_1, t_2, \ldots, t_n)|_q &= t_i|_q, \\
&\text{if } 1 \leq i \leq n \text{ and } q \text{ is a position}
\end{align*}
\]

Example 5. Trace of the computation of the subterm at a given position:

\[
\begin{align*}
a(s(z), z)|_{(1,1)} &= s(z)|_{(1)} = z
\end{align*}
\]

Exercise 5a. Code a Curry function that takes a term \( t \) and a position \( p \) of \( t \) and returns \( t|_p \). Hint: use the representation of terms you chose for Exercise 2.

Exercise 5b. Code a Curry function that takes two terms, \( t \) and \( u \), and returns all the positions of \( u \) in \( t \).

Substitution

A substitution is a mapping from the set of variables \( \text{VAR}(\Sigma) \) to the set of terms \( \text{TER}(\Sigma) \). The set of variables “affected” by a substitution, \( \sigma \), namely:

\[
\text{DOM}(\sigma) = \{ x \mid \sigma(x) \neq x \}
\]

is called the domain of \( \sigma \). The set:

\[
\text{IMG}(\sigma) = \{ y \mid y \text{ occurs in } \sigma(x) \text{ for some } x \in \text{DOM}(\sigma) \}
\]

is called the image or range of \( \sigma \).

Most often in logic programming it is required that \( \text{DOM}(\sigma) \) is a finite set and that \( \text{DOM}(\sigma) \cap \text{IMG}(\sigma) = \emptyset \).

A substitution is denoted by:

\[
\{ v_1 \mapsto t_1, \ldots, v_n \mapsto t_n \}
\]

A substitution \( \sigma \) is extended to terms as follows:

\[
\sigma(f(t_1, t_2, \ldots, t_n)) = f(\sigma(t_1), \sigma(t_2), \ldots, \sigma(t_n))
\]

Example 6. If \( \sigma = \{ x \mapsto 0, y \mapsto 1 \} \) and \( t = x + y \), then \( \sigma(t) = 0 + 1 \).

Exercise 6. Define a substitution and code a Curry function that takes a substitution and a term and returns the result of applying the substitution to the term.

Rewrite System

A rewrite system is a pair \( \langle \Sigma, R \rangle \), where \( \Sigma \) is a signature and \( R \) is a set of rewrite rules over \( \Sigma \).

Let \( t \) be a term, \( l \rightarrow r \) a rule, \( p \) a position of \( t \), and \( \sigma \) a substitution such that \( \sigma(l) = t|_p \), i.e., the subterm of \( t \) at position \( p \) is a redux or an instance of \( l \).

A rewrite (step) is a pair of terms \( t \rightarrow t[\sigma(r)]|_p \), where the latter denotes the term obtained by replacing the subterm of \( t \) at position \( p \) with \( \sigma(r) \). The rewrite relation, denoted by \( \rightarrow^* \), of \( \langle \Sigma, R \rangle \) is the set of all the rewrite steps. \( \rightarrow^* \) denotes the reflexive transitive closure of \( \rightarrow \).

A rewrite relation is:

- **Confluent** if \( t \rightarrow t_1 \) and \( t \rightarrow t_2 \) imply the existence of a \( u \) such that \( t_1 \rightarrow u \) and \( t_2 \rightarrow u \), for all \( t, t_1 \), and \( t_2 \).

- **Strongly terminating** if there is no infinite sequence of rewrite steps.

- **Weakly terminating** if every term has a normal form.
Rewrite System cont’d

Example 9. The system of Example 7 is confluent and strongly terminating. How can one become convinced of this claim?

Exercise 9a. Discuss confluence and termination of:

\[ a \rightarrow b \]
\[ f(a) \rightarrow f(a) \]

Exercise 9b. Implementing rewriting for the system of Example 7 in Curry is easy if one is not picky about choosing steps. Code a small program to this aim. Hint: for testing, code two functions as follows:

- \( \text{encd} \) takes a natural number \( i \) and return \( s(s(\ldots s(z)\ldots)) \), where there are exactly \( i \) applications of \( s \),
- \( \text{decd} \) is the inverse of \( \text{encd} \), i.e., \( \text{decd}(\text{encd}(i)) = i \) and vice versa.

and inspect the normal form of

\[ \text{decd}(a(\text{encd}(i), \text{encd}(j))) \]

and likewise for \( m \).  

Orthogonality

Two key properties of rewrite systems are:

- **Left linearity:** no repeated variables in the rules’ llhs.
- **Non ambiguity:** let \( l_1 \rightarrow r_1 \) and \( l_2 \rightarrow r_2 \) be two rules such that \( l_1 \) unifies with a non-variable subterm of \( l_2 \), i.e., there exists a substitution \( \sigma \) and a position \( p \) of \( l_2 \) such that \( \sigma(l_2[p]) = \sigma(l_1) \). The term \( \sigma(l_2) \) can be rewritten in two ways, namely \( \sigma(r_2) \) and \( \sigma(l_2[r_1][p]) \). These two terms form a critical pair.

A system is **orthogonal** if it is left-linear and is non-ambiguous (has no critical pairs). Orthogonal systems are confluent.

Exercise 11. Consider the following Curry (and Haskell) program as a rewrite system:

\[
\begin{align*}
\text{insert} : \text{Color} &\times \text{Stack} \rightarrow \text{Stack} \\
\text{push} : \text{Stack} &\rightarrow \text{Stack} \\
\text{top} &\rightarrow \text{Color}
\end{align*}
\]

Types:

\[
\begin{align*}
\text{red} &\rightarrow \text{Color} \\
\text{blue} &\rightarrow \text{Color} \\
\text{green} &\rightarrow \text{Color} \\
\text{empty} &\rightarrow \text{Stack} \\
\text{push} &\rightarrow \text{Stack} \\
\text{pop} &\rightarrow \text{Stack} \\
\text{top} &\rightarrow \text{Color}
\end{align*}
\]

Rules:

\[
\begin{align*}
\text{pop} &\rightarrow \text{Stack} \\
\text{top} &\rightarrow \text{Color}
\end{align*}
\]

Abstract Systems

Modern treatments of rewrite systems consider abstract **reduction systems**, structures \( \langle A, \rightarrow \rangle \), where \( A \) is a set and \( \rightarrow \) is a binary relation (or family of relations) on \( A \).

Many key concepts of rewriting, e.g., step, redex, termination, confluence, etc., are independent of terms, hence can be defined for abstract systems.

Many fundamental theorems of rewriting can be proved for abstract systems.

We focus on term rewriting systems because they are more interesting for programming.
Graph Rewriting

Formalism similar to term rewriting, but expressions are graphs rather than trees. Consider the system:

\[
\begin{align*}
    \text{coin} & \rightarrow 0 \\
    \text{coin} & \rightarrow 1 \\
    \text{flip} \ 0 & \rightarrow 1 \\
    \text{flip} \ 1 & \rightarrow 0
\end{align*}
\]

and the expressions:

\[
\begin{align*}
    & (\text{flip} \ \text{coin}, \ \text{flip} \ \text{coin}) \\
    \quad \text{flip} \quad \text{flip} \quad \text{flip} \quad \text{flip} \\
    \quad \text{coin} \quad \text{coin} \quad \text{coin}
\end{align*}
\]

The Curry (textual) representations are:

\[
\begin{align*}
    (\text{flip} \ x, \ \text{flip} \ x) \text{ where } x = \text{coin} \\
    (\text{flip} \ \text{coin}, \ \text{flip} \ \text{coin})
\end{align*}
\]

The first one has 2 values only: \((0, 0)\) and \((1, 1)\). The second has 4, also: \((0, 1)\) and \((1, 0)\).

In Curry, variables are always shared, multiple occurrences of the same variable refer to the same object.