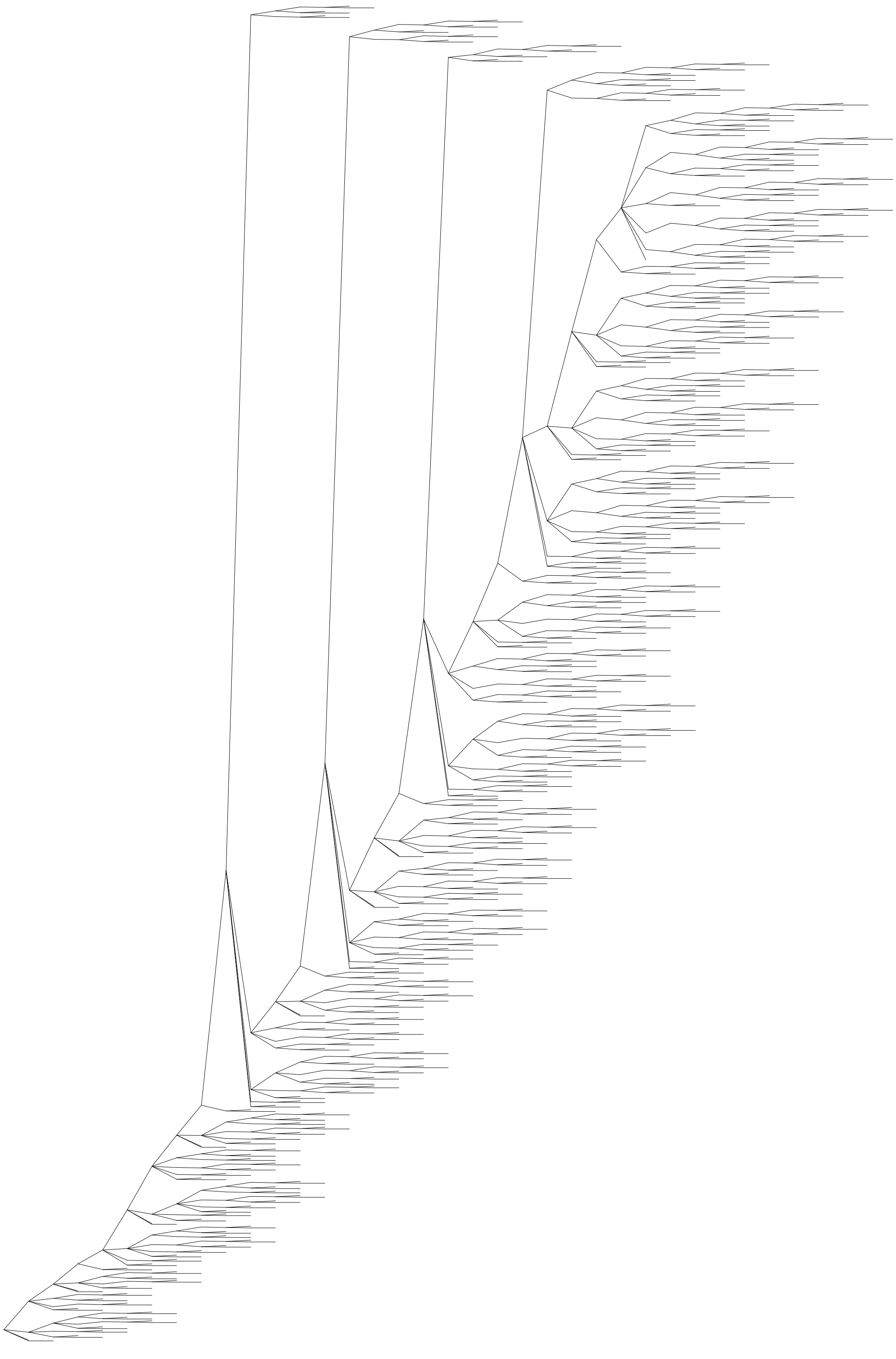
Part I

Fundamentals



S (**S S S**) **S** (**S** (**S S**) **S**)
after 50 rightmost innermost reductions

Chapter 1

Abstract Reduction Systems

1.1 Introduction

An abstract reduction system (ARS) is just a set M, equipped with a relation R.

Abstract reduction does completely disregard the *internal* structure of the elements of M. Instead it focuses on the *external* structure that R introduces on M as a whole.

We imagine reductions as computations, so in particular we are interested in *well-founded* relations R because these correspond to *finite* computations that eventually halt, yielding some result.

Of course we would like those results to be *unique*. In general this is not the case, as R is only a relation, not a function. So we additionally require *confluence* of R.

Later on, we will apply the abstract theory to two concrete reduction systems: term rewriting systems and lambda calculus.

The presentation of ARS closely follows Nipkow's course notes [Nip95].

1.2 Relations

1.2.1. Definition. A *relation* R on a set M is a subset of $M \times M$.

Often the intended meaning of $(x, y) \in R$ is "x rewrites to y in one step", so we conveniently write this as $x \rightarrow_R y$, or even $x \rightarrow y$ when R is clear from the context.

Constructions

Relations can be composed.

1.2.2. Definition. $R_1 \circ R_2 \ = \ \{(x, z) \mid \exists y : (x, y) \in R_1, (y, z) \in R_2\}$

The following denotes the identity for the $\circ$ operator.

1.2.3. Definition. $\rightarrow^0 \ = \ 1_M \ = \ \{(x, x) \mid x \in M\}$

1.2.4. Definition. $\stackrel{=}{\rightarrow} \ = \ \rightarrow \cup \rightarrow^0$

Relations can be inverted.

1.2.5. Definition. $R^{-1} \ = \ \{(y, x) \mid (x, y) \in R\}$

We write $\leftarrow$ for $\rightarrow^{-1}$.

1.2.6. Definition. $\leftrightarrow \ = \ \leftarrow \cup \rightarrow$

Composition can be repeated for a fixed number of times.

1.2.7. Definition. $\rightarrow^{n+1} \ = \ \rightarrow \circ \rightarrow^n$

where $\rightarrow^0$ is defined above.
Composition can be repeated finitely often.

1.2.8. Definition. $\rightarrow^+ \ = \ \bigcup_{k>0} \rightarrow^k; \quad \rightarrow^* \ = \ \bigcup_{k\geq 0} \rightarrow^k$

We often write $\twoheadrightarrow$ instead of $\rightarrow^*$.

Properties

1.2.9. Definition. $\rightarrow$ is *transitive* iff $\rightarrow \circ \rightarrow \ \subseteq \ \rightarrow$.

1.2.10. Proposition. $\rightarrow^+$ is the transitive closure of $\rightarrow$.

This means: $\rightarrow^+$ is the smallest transitive relation that contains $\rightarrow$. (The following propositions about closures are to be understood similarly.)

1.2.11. Definition. $\rightarrow$ is *symmetric* iff $\rightarrow \ = \ \leftarrow$.

1.2.12. Proposition. $\leftrightarrow$ is the symmetric closure of $\rightarrow$.

1.2.13. Definition. $\rightarrow$ is *antisymmetric* iff $\rightarrow \cap \leftarrow \ \subseteq \ 1_M$.

1.2.14. Definition. $\rightarrow$ is *reflexive* iff $1_M \ \subseteq \ \rightarrow$.

1.2.15. Proposition. $\stackrel{=}{\rightarrow}$ is the reflexive closure of $\rightarrow$.

1.2.16. Proposition. $\rightarrow^*$ is the transitive reflexive closure of $\rightarrow$.

1.2.17. Definition. $\rightarrow$ is *irreflexive* iff $1_M \cap \rightarrow \ = \ \emptyset$.

Orders

A relation on a set creates order.

1.2.18. Definition. A relation is a *quasi order* iff it is reflexive and transitive.

Quasi orders are denoted by $\gtrsim$, and $\lesssim$ stands for $\gtrsim^{-1}$.

1.2.19. Definition. A relation is an *equivalence* iff it is a quasi order and symmetric.

Equivalences are usually denoted by $\approx$.

1.2.20. Proposition. A quasi order $\gtrsim$ induces an equivalence by $x \approx y$ iff $x \gtrsim y$ and $x \lesssim y$.

1.2.21. Definition. A relation is a *partial order* iff it is a quasi order and antisymmetric.

Partial orders are usually denoted by $\leq$. Then $\geq$ stands for $\leq^{-1}$.

1.2.22. Proposition. The quasi order $\gtrsim$ induces a partial order on the equivalence classes of $\approx$.

1.2.23. Definition. $\gtrsim$ is *total* (or: *linear*) iff $\lesssim \cup \gtrsim \ = \ M \times M$.

1.2.24. Definition. A relation is an *order* iff it is a partial order and total.

1.2.25. Definition. The *strict part* $>$ of a quasi order $\gtrsim$ is defined by $> = \ \gtrsim \setminus \lesssim$.

1.3 Collections

Sequences

1.3.1. Definition. M^n is the set of n-tuples with elements from M, and $M^* = \cup_{k \geq 0} M^k$.

1.3.2. Definition. Sequences are written down by enclosing their elements in parentheses. The empty sequence is denoted by $()$, and $x : (y_1, \ldots, y_n) = (x, y_1, \ldots, y_n)$. Sequences can be *concatenated*: $(x_1, \ldots, x_n) +\!\!+ (y_1, \ldots, y_m) = (x_1, \ldots, x_n, y_1, \ldots, y_m)$

Sequences are partially ordered by the prefix order.

1.3.3. Definition (prefix order). $(p_1, \ldots, p_m) \sqsubseteq (q_1, \ldots, q_n)$ iff $m \leq n$ and for $i \in \{1, \ldots, m\}$: $p_i = q_i$.

1.3.4. Definition. Sequences p and q are *independent*, written $p \parallel q$, iff neither $p \sqsubseteq q$ nor $p \sqsupseteq q$.

Multisets

Multisets can be thought of as sets with repeated elements.

1.3.5. Definition. A *finite multiset* on M is a function f from M into the naturals, with only finitely many $x \in M$ that have $f(x) > 0$.

As we only consider finite multisets throughout this thesis, we omit the attribute "finite" from now on.

For multiset operations, the usual notation for sets is employed. In the following, $A, B\ldots$ denote multisets.

1.3.6. Definition. $x \in A$ iff $A(x) > 0$.

1.3.7. Definition. $A \cup B = f$ where $f(x) = A(x) + B(x)$.

1.3.8. Definition. $A \cap B = f$ where $f(x) = \max\{A(x), B(x)\}$.

1.3.9. Definition. $A \setminus B = f$ where $f(x) = \max\{0, A(x) - B(x)\}$.

1.4 Chains

Given a relation, we will be concerned with ordered chains of elements. As we have "reduction" in mind, we imagine that transition from one element of a chain to another describes a stepwise computation. Since a computation should halt, we're interested in avoiding infinite chains.

Normal Forms

1.4.1. Definition. If $x \rightarrow^+ y$, then y is a *successor* of x, and x is a *predecessor* of y. If $x \rightarrow y$, then y is an *immediate successor* of x, and x is an *immediate predecessor* of y.

1.4.2. Definition. An $x \in M$ is in *normal form* iff it has no successors.

1.4.3. Definition. An $x \in M$ *has a normal form* iff $\exists y:\ x \twoheadrightarrow y$ and y is in normal form.

1.4.4. Definition. A relation $\rightarrow$ on M is *normalizing* iff each $x \in M$ has a normal form.

1.4.5. Definition. A relation $\rightarrow$ is *terminating* iff there is no infinite chain $x_1 \rightarrow x_2 \rightarrow \ldots$.

1.4.6. Remark. Often, normalizing and terminating are called *weakly* and *strongly* normalizing, respectively.

1.4.7. Definition. We write $\downarrow(x)$ if x terminates, and $x \twoheadrightarrow \infty$ if it does not.

Well-foundedness

1.4.8. Definition. An element x is *minimal* for $\rightarrow$ iff it has no successors.

1.4.9. Definition. A relation $\rightarrow$ is *well-founded* iff each nonempty subset $B \subseteq M$ contains a minimal element.

1.4.10. Proposition. A relation is terminating iff it is well-founded.

Proof. If $\rightarrow$ is not terminating, then there is some infinite reduction $x_1 \rightarrow x_2 \rightarrow \ldots$. The set of all these x_i does not contain a minimal element.

On the other hand, if $\rightarrow$ is not well-founded, then there is some non-empty set B without a minimal element. Take an arbitrary $x_0 \in B$. Then x_0 must have at least one immediate successor $x_1 \in B$, otherwise x_0 would be minimal. Again x_1 has an immediate successor x_2, and so on. This constructs an infinite reduction $x_1 \rightarrow x_2 \rightarrow \ldots$. □

If M is equipped with a well-founded relation, we may prove properties of M by *well-founded induction.*

1.4.11. Proposition (Well-founded induction). Let P be a predicate on M.

$$\text{If} \quad \forall x \in M : \left(\Big(\forall y \in M : (x \to y) \Rightarrow P(y) \Big) \Rightarrow P(x) \right), \quad \text{then} \quad \forall x \in M : P(x).$$

Proof. Assume there is an $x_1 \in M$ for which $P(x_1)$ does not hold. Then there must be some $x_2 \in M$ with $x_1 \to x_2$ such that $P(x_2)$ is false as well. This construction can be repeated, yielding an infinite chain $x_1 \to x_2 \to \ldots$. This contradicts well-foundedness of $\to$. $\square$

1.4.12. Example. Classical induction on the naturals can be seen as an instance of well-founded induction. Define $x \to y$ as $x = 1 + y$.

Construction of well-founded relations

Two well-founded relations on the same set can be combined.

1.4.13. Proposition. If $\gtrsim_1$ and $\gtrsim_2$ are well-founded relations on M, then

$x \gtrsim y$ iff $x >_1 y$ or ($x \approx_1 y$ and $x >_2 y$)

defines a well-founded relation on M.

A relation on M can be extended to a relation on sequences with elements from M.

1.4.14. Proposition (lexicographic order). If $\gtrsim$ is a quasi order on M, then

$(x_1, \ldots, x_m) \gtrsim_{\text{lex}} (y_1, \ldots, y_n)$
iff there exists $k \leq \min\{m, n\}$ such that $\forall 1 \leq i < k : x_i \approx y_i$ and $x_k \gtrsim y_k$

defines a quasi order on M^*

1.4.15. Proposition. If $\gtrsim$ is well-founded on M, then for all n, $\gtrsim_{\text{lex}}$ is well-founded on M^n.

1.4.16. Remark. It is not well-founded on M^*.

A well-founded relation on a set M can be made into a well-founded relation on multisets of M. Intuitively we call a multiset B *smaller* than a multiset A if B can be obtained from A by replacing some elements (from the multiset X) with smaller ones (from Y).

1.4.17. Proposition (multiset order). If $\gtrsim$ is a well-founded relation on M, then

$A \gg B$ iff there exist multisets $X \neq \emptyset$ and Y such that $(A \setminus X) \cup Y = B$ and for each $y \in Y$, there is an $x \in X$ with $x > y$.

defines a well-founded relation on multisets of M.

Proof. Define $\gg_1$ just like $\gg$ but with the restriction that X be a singleton multiset. Then use $\gg \; = \; \gg_1^+$. $\square$

1.4.18. Proposition. If $\gtrsim$ is total on M, then $A \gg B$ iff $A' >_{\text{lex}} B'$ where X' is X, sorted in descending order.

1.5 Well quasi orders

1.5.1. Definition. A quasi order $\gtrsim$ is a *well quasi order* (wqo) iff in each infinite sequence $x_1, x_2, \ldots$ there are $i < j$ with $x_i \lesssim x_j$.

One can use a wqo to show that a relation is terminating.

1.5.2. Proposition. Let $\gtrsim$ be a wqo on M, and $\to$ any relation on M. If $X \to^+ Y$ implies $X \not\gtrsim Y$, then $\to$ is terminating.

Usually, $\gtrsim$ encodes knowledge on the internal structure of the elements of M, while $\to$ describes external structure generated by some rewrite system.
This is the idea underlying reduction orders on terms, presented in section 2.8.

1.6 Confluence and Termination

1.6.1. Definition. A relation $\to$ has the *diamond property* iff $\leftarrow \circ \to \ \subseteq \ \to \circ \leftarrow$.

In other words, for all x, y, z with $x \to y$ and $x \to z$, there must exist a w such that $y \to w$ and $z \to w$.

1.6.2. Definition. A relation $\to$ is *locally confluent* iff $\leftarrow \circ \to \ \subseteq \ \to^* \circ \leftarrow^*$.

In other words, for all x, y, z with $x \to y$ and $x \to z$, there must exist a w such that $y \twoheadrightarrow w$ and $z \twoheadrightarrow w$.

1.6.3. Definition. A relation $\to$ is *confluent* iff $\leftarrow^* \circ \to^* \ \subseteq \ \to^* \circ \leftarrow^*$.

Equivalently, $\to^*$ has the diamond property.
Confluence does not follow from local confluence. See proposition 1.6.8.
However simple induction proves

1.6.4. Proposition. If $\to$ has the diamond property, then $\to^*$ also has.

1.6.5. Definition. A relation is *Church-Rosser* iff $\leftrightarrow^* \ \subseteq \ \to^* \circ \leftarrow^*$.

1.6.6. Proposition. A relation is Church-Rosser iff it is confluent.

Proof. By induction on the number of changes of direction in $\leftrightarrow^*$. □

1.6.7. Definition. A relation is *convergent* iff it is terminating and confluent.

1.6.8. Proposition (Newman-Lemma). If a relation $\to$ is terminating, then confluence and local confluence coincide.

1.6.9. Proposition. If $\to$ is confluent, then each x has at most one normal form.

So confluent relations describe computations that are well-behaved: if they halt with a result, then it might have been obtained through different paths, but it is unique.

Chapter 2

Term Rewriting Systems

2.1 Introduction

This chapter presents term rewriting systems (TRS) as a concrete examples of an abstract reduction systems, by specifying the set M of elements that are being rewritten.

Here M is the set of *terms*. These are elements of some free algebra. They have *internal* structure (via the subterm relation), and rewriting rules impose the *external* structure.

Historically, *string* rewriting had been considered first. We may regard strings as special terms, and therefore omit a separate treatment of string rewriting.

An extension of term rewriting is *higher order* rewriting, where terms may contain bound variables. We will not be interested in its general theory (for instance, see [KvOvR93]), but only in one particular example, lambda calculus, which we will deal with in the next chapter.

The presentation of TRS closely follows Avenhaus' textbook [Ave95]. Standard references are the surveys by Klop [Klo92] and that by Dershowitz and Jouannaud [DJ90]. We will follow their suggestions for notation [DJ91].

2.2 The Term Algebra

Terms

2.2.1. Definition. An *signature* is a pair (F, a) where F is a set and a is a mapping F to $\mathbb{N}$.
$a(f)$ is called the *arity* of $f \in F$.
Elements f with $a(f) = 0$ are called *constants.*

Usually, a is clear from the context so we just write down F.

2.2.2. Definition. A tuple (F, a, D) is an *algebra with carrier set* D if (F, a) is a signature and and for each $f \in F$, $f : D^{a(f)} \to D$.

The *term algebra* generated by F is the free algebra on some set of variables.

2.2.3. Definition. Given a signature F and a set V of variables, disjoint from F, the set of *terms* $T(F, V)$ is defined by

$$\begin{array}{rl} v \in V : & v \in T(F,V) \\ f \in F,\ n = a(f),\ t_1 \in T(F,V),\ \ldots,\ t_n \in T(F,V) : & f(t_1, \ldots, t_n) \in T(F,V) \end{array}$$

2.2.4. Definition. The set of *ground* terms on F, denoted by $T(F)$, is $T(F, \emptyset)$.

2.2.5. Definition. The size $|t|$ of a term $t \in T(F, V)$ is defined by

$$\begin{array}{l} |v| = 1 \quad \text{for} \quad v \in V \\ |f(t_1, \ldots, t_n)| = 1 + |t_1| + \ldots + |t_n| \end{array}$$

2.2.6. Definition. The depth of a term $t \in T(F, V)$ is defined by

$$\begin{array}{l} \text{depth}(v) = 0 \quad \text{for} \quad v \in V \\ \text{depth}(f(t_1, \ldots, t_n)) = 1 + \max\{\text{depth}(t_1), \ldots, \text{depth}(t_n)\} \end{array}$$

Positions

A *position* is a sequence of natural numbers that indicates the location of some subterm.

2.2.7. Definition. The set of subterm positions, $\text{Pos}(t)$, is

$$\begin{array}{l} \text{Pos}(v) = \{()\} \quad \text{for} \quad v \in V \\ \text{Pos}(f(t_1, \ldots, t_n)) = \{()\} \cup \bigcup_{i \in \{1,\ldots,n\}, s \in \text{Pos}(t_i)} (i : s) \end{array}$$

Subterms

2.2.8. Definition. $t\,[p]$, the subterm of a term t at position p, is defined by

$$\begin{aligned} t\,[()] &= t \\ f(t_1,\ldots,t_n)\,[(p_1,p_2,\ldots)] &= t_{p_1}\,[(p_2,\ldots)] \end{aligned}$$

This is only defined in case $p \in \mathrm{Pos}(t)$.

2.2.9. Definition. $t\,[p \leftarrow t']$ is the result of replacing the subterm at position p in t with t'.

$$\begin{aligned} t\,[() \leftarrow t'] &= t' \\ f(t_1,\ldots,t_n)\,[(p_1,p_2,\ldots) \leftarrow t'] &= f(\ldots,\quad \underbrace{t_{p_1}\,[(p_2,\ldots) \leftarrow t']}_{\text{the } p_1\text{-th argument of } f} \quad ,\ldots) \end{aligned}$$

This is only defined in case $p \in \mathrm{Pos}(t)$.

2.2.10. Definition. s *is a subterm of* t, written $s \trianglelefteq t$ iff there is $p \in \mathrm{Pos}(t)$ such that $t\,[p] = s$. s is a *strict* subterm, written $s \triangleleft t$, iff $p \neq ()$.

2.2.11. Proposition. $\triangleright$ is a well-founded partial order.

2.2.12. Proposition. $\triangleright$ is not a wqo.

2.2.13. Definition. The set of variables of a term $t \in T(F,V)$, denoted by $\mathrm{Var}(t)$, is $\{v \mid v \in V, v \trianglelefteq t\}$.

We define a notation for the set of all positions in t where a certain subterm x occurs:

2.2.14. Definition. $\mathrm{Pos}(x,t) = \{p \in \mathrm{Pos}(t) \mid t\,[p] = x\}$

Contexts

A context is a term with a hole.

2.2.15. Definition. A term $C \in T(F \cup [])$, where $[]$ is a new nullary constructor, with $\mathrm{Pos}([],t) = \{p\}$, is called *context*. It is written as $C[]$. The expression $C[X]$ stands for $C\,[p \leftarrow X]$.

Words as Terms

2.2.16. Definition. A term algebra where there is exactly one $z \in F$ with $a(z) = 0$, and all other $f \in F$ have $a(f) = 1$, is called *word algebra*.

In such cases, we write $fgffg$ instead of $f(g(f(f(g(z)))))$.
This gives a natural mapping between $T(F)$ and the sequences (words) $(F \setminus \{z\})^*$.
Words have additional structure because they can be concatenated, and with this operation they form a monoid. It is not directly possible to do this with terms. One may consider terms with designated "concatenation leaves". We will not use this in the sequel.

2.3 Substitutions

2.3.1. Definition. A *substitution* is a mapping from variables to terms $\sigma : V \to T(F,V)$, with only finitely many $v \in V$ for which $v\sigma \neq v$. Those v are called the *domain* of σ.

Usually $\sigma(X)$, the application of the substitution σ to a term X is written as $X\sigma$.

σ is extended to a mapping $\sigma' : T(F,V) \to T(F,V)$ by

$$\begin{array}{rcl} v\sigma' & = & v\sigma \quad \text{for all} \quad v \in V \\ f(t_1,\ldots,t_n)\sigma' & = & f(t_1\sigma',\ldots,t_n\sigma') \end{array}$$

Usually, σ' is denoted by σ as well.

If σ is the substitution with $x\sigma = X, y\sigma = Y$ and $v\sigma = v$ for all other $v \in V$, then we write σ as $[x \leftarrow X, y \leftarrow Y]$.

Subsumption

2.3.2. Definition. A term t_1 is an *instance* of a term t_2 iff there is a substitution σ such that $t_1 = t_2\sigma$. This is also called t_2 *subsumes* t_1 and denoted by $t_1 \succeq t_2$. Equivalently, we say t_1 *matches the pattern* t_2, and the substitution σ is called a *match*.

2.3.3. Proposition. $t_1 \succeq t_2$ iff $\mathrm{Inst}(t_1) \subseteq \mathrm{Inst}(t_2)$,

where $\mathrm{Inst}(t)$ denotes the set of all instances of t.

2.3.4. Proposition. $\succeq$ is a well-founded quasi order on $T(F,V)$.

Subsumption, which was defined on terms, can also be extended to substitutions.

2.3.5. Definition. Let σ_1 and σ_2 be substitutions. We say that σ_1 *subsumes* σ_2, written $\sigma_1 \succeq \sigma_2$, iff for all v from their common domain we have $\sigma_1(t) \succeq \sigma_2(t)$

2.3.6. Proposition. $\succeq$ is a quasi order on substitutions.

Unification

2.3.7. Definition. A *unifier* of terms t_1 and t_2 is a substitution σ such that $t_1\sigma = t_2\sigma$. The terms t_1 and t_2 are called *unifyable.*

2.3.8. Definition. A *most general unifier* (mgu) of terms t_1 and t_2 is a unifier σ of t_1 and t_2 that subsumes all unifiers of t_1 and t_2.

mgus are computable, if the terms are unifyable. mgus are essentially unique.

2.4 Rewrites

Term Rewriting Systems

2.4.1. Definition. A *term rewriting system* (TRS for short) on a signature (F,a) is a set of pairs $(l,r) \in T(F,V) \times T(F,V)$ with $\mathrm{Var}(l) \supseteq \mathrm{Var}(r)$. These pairs are called *rules*, and a rule (l,r) is written as $l \to r$. Usually, we will consider TRS with finitely many rules.

2.4.2. Definition. A TRS R defines a relation $\to_R$ on terms $T(F,V)$ by $t_1 \to_R t_2$ iff

- there is $(l,r) \in R$ such that
- there is a position $p \in \mathrm{Pos}(t_1)$ such that
- there is a substitution σ that matches $t_1\,[p]$ to l and
- $t_2 = t_1\,[p \leftarrow r\,\sigma]$.

$t_1\,[p]$ is called *redex* (from: *red*ucible *ex*pression), p is called *redex position*, $r\,\sigma$ is called *contractum* of the redex.

We simply write $\to$ instead of $\to_R$ whenever R is clear from the context.

2.4.3. Definition. $X \hookrightarrow Y$ means when reducing X, Y occurs as a subterm: $X \to Z$ and $Z \trianglerighteq Y$. We write $\hookrightarrow\!\!\!\!\rightarrow$ instead of $\hookrightarrow^*$.

Syntactic Properties of Rewrite Rules

2.4.4. Definition. A term $t \in T(F,V)$ is called *linear* iff no variable occurs more than once: for all $v \in V$, $|\mathrm{Pos}(v,t)| \leq 1$.

2.4.5. Definition. A rule (l,r) is called *left* (resp. *right) linear*, iff l (resp. r) is a linear term. It is called *linear* iff both l and r are linear and $\mathrm{Var}(l) = \mathrm{Var}(r)$.

2.4.6. Definition. A TRS R is called *(left* resp. *right) linear* iff *all* its rules are.

2.4.7. Definition. A rule (l,r) is called *(left, right) ground* iff $(l,\ r)$ both of l and r are ground terms.

2.4.8. Definition. A TRS R is called *(left resp. right) ground* iff *all* its rules are.

2.4.9. Definition. A rule (l,r) is called *duplicating* iff there is $v \in V$ with $1 \leq |\mathrm{Pos}(v,l)| < |\mathrm{Pos}(v,r)|$.

2.4.10. Definition. A rule (l,r) is called *erasing* iff there is $v \in V$ with $|\mathrm{Pos}(v,l)| > |\mathrm{Pos}(v,r)| = 0$.

2.4.11. Definition. A rule (l,r) is called *collapsing* iff $r \in V$.

2.4.12. Definition. A TRS is called *duplicating* resp. *erasing* resp. *collapsing* iff *some* of its rules are.

2.5 Confluence of TRS

For rewrite systems in general, confluence is a desirable property, because it describes well-behaved computations.

In a TRS, non-confluence might arise because in a term, redexes might overlap, and reducing one of them destroys the other. This situation is captured in the following definition.

2.5.1. Definition. In a TRS R, let

- $(l_1 \to r_1)$ and $(l_2 \to r_2)$ be two rules, (They need not be distinct. It is assumed that $\mathrm{Var}(l_1)$ is disjoint from $\mathrm{Var}(l_2)$, which can be achieved by re-naming.)
- $p \in \mathrm{Pos}(l_1)$ with $l_1\,[p] \notin V$, where $p \neq ()$ if the rules are the same,
- $l_1\,[p]$ and l_2 are unifyable, where σ is their mgu.

Then $(l_1\,[p \leftarrow r_2]\;\sigma, r_1\,\sigma)$ is called a *critical pair* of R.

2.5.2. Remark. A rule might overlap with itself, but not at root position.

Some critical pairs that are joinable: there is some z with $x \twoheadrightarrow z$ and $y \twoheadrightarrow t$). Non-joinable critical pairs (x, y) are obstacles for confluence. For a system with such critical pairs, confluence can hopefully be obtained by adding one of the rules $x \to y$ and $y \to x$ to the TRS.

This process of adding rules that keep $\leftrightarrow^*$ invariant and terminating, but make $\to$ confluent, is called *completion*. Since combinatory logic is non-overlapping, we don't go into details of completion here.

2.5.3. Definition. A TRS is called *non-overlapping* iff it has no critical pairs.

2.5.4. Definition. A TRS is called *orthogonal* iff it is non-overlapping and left linear.

2.5.5. Proposition. Orthogonal TRS are confluent.

Proof. Use the relation of *parallel reduction*, defined below. $\xrightarrow{\parallel}$ that has the diamond property, and $\to \subseteq \xrightarrow{\parallel} \subseteq \to^*$. Then $\xrightarrow{\parallel}^*$ has the diamond property as well, and $\to^* \subseteq \xrightarrow{\parallel}^* \subseteq (\to^*)^* = \to^*$, which means $\to^* = \xrightarrow{\parallel}^*$, so $\to^*$ has the diamond property. $\square$

2.5.6. Definition (parallel reduction). $s_0 \xrightarrow{\parallel} s_n$ iff $s_0 \to s_1 \to \ldots \to s_n$ and the reductions happen at positions $p_0, \ldots, p_{n-1}$ that are pairwise independent: $i \neq j \;\Rightarrow\; p_i \parallel p_j$.

2.6 Reduction Strategies

A *strategy* maps a term to a reduction sequence starting with that term.

2.6.1. Definition. A strategy is *normalizing* iff it leads to a normal form, provided there is one.

2.6.2. Definition. A reduction step is *outermost* iff the contracted redex is not a proper subterm of a redex.

2.6.3. Proposition. For orthogonal systems, any outermost strategy is normalizing.

2.6.4. Definition. A reduction step is *innermost* iff the contracted redex has no proper subterm that is itself a redex.

An innermost strategy need not be normalizing. It is, however, in case the TRS is non-erasing.

2.6.5. Proposition. If a TRS is both non-overlapping and non-erasing, then normalization and termination coincide.

This says that *any* reduction strategy finds the normal form, provided it exists. In particular, we can use innermost reduction.

For a detailed analysis, see Gramlich's survey paper [Gra92].

Since **CL**(**S**) is orthogonal and non-erasing, we don't have to worry much about strategies.

2.7 Ground TRS

2.7.1. Definition. A TRS is *ground* iff in all of its rules $l \to r$, both l and r are ground terms.

Ground TRS are very restricted in their rewriting power. Some of the problems that are hard for general TRS are manageable for ground TRS.

2.7.2. Proposition. It is decidable whether a ground TRS is terminating.

2.7.3. Proposition. It is decidable whether a ground TRS is confluent.

A proof is in [Oya87].

2.8 Orders on Terms

Homeomorphic embedding

2.8.1. Definition. The relation $\geq_{\mathrm{HE}}$ of *homeomorphic embedding* on $T(F)$ is given by $\to_E^+$ where E is the TRS

$$\{f(x_1, \ldots, x_{a(f)}) \to x_i \mid f \in F, i \in \{1, \ldots, a(f)\}\}$$

2.8.2. Proposition. $\leq_{\mathrm{HE}}$ is a well-founded partial order.

This relation is "better" than the subterm relation $\trianglerighteq$ because we have

2.8.3. Theorem (Kruskal's Tree Theorem). $\geq_{\mathrm{HE}}$ is a wqo on $T(F)$.

This theorem will prove very useful in the analysis of infinite sequences of terms.

Reduction orders

2.8.4. Definition. A relation $>$ on $T(F, V)$ is *closed w.r.t. substitutions* iff for all substitutions $s : V \to T(F, V)$ and all terms $t_1, t_2 \in T(F, V)$:

$$t_1 > t_2 \quad \Rightarrow \quad s(t_1) > s(t_2)$$

2.8.5. Definition. A relation $>$ on $T(F)$ is *closed w.r.t. contexts* iff for all terms $t \in T(F,V)$, for all positions $p \in \mathrm{Pos}(t)$, for all terms $t_1, t_2 \in T(F,V)$:

$$t_1 > t_2 \quad \Rightarrow \quad t\,[p \leftarrow t_1] > t\,[p \leftarrow t_2]$$

2.8.6. Definition. A relation $>$ on $T(F,V)$ *contains the subterm relation* iff for all terms t, for all positions $p \neq ()$ in $\mathrm{Pos}(t)$:

$$t \quad > \quad t\,[p]$$

2.8.7. Definition. A *reduction order* is a quasi order $\gtrsim$ on terms $T(F,V)$ such that

- $>$ is well-founded
- $>$ is closed w.r.t. substitutions
- $>$ is closed w.r.t. contexts

2.8.8. Definition. A reduction order $\gtrsim$ is *compatible* with a TRS R iff $l > r$ for all rules $l \to r$ from R.

2.8.9. Proposition. A TRS R is terminating iff there is a reduction order compatible with R.

Simplification orders

2.8.10. Definition. A *simplification order* is a quasi order $\gtrsim$ on terms $T(F,V)$ such that

- $>$ is closed w.r.t. contexts
- $>$ contains the subterm relation

2.8.11. Proposition. Each simplification order $\gtrsim$ on $T(F,V)$ contains the relation of homeomorphic embedding: $\geq_{\mathrm{HE}} \ \subseteq \ \gtrsim$

That's good. Kruskal's theorem 2.8.3 can be applied.

2.8.12. Proposition. Each simplification order on $T(F)$ is well-founded.

2.8.13. Proposition. Each simplification order on $T(F,V)$ that is closed w.r.t. substitutions, is a reduction order.

Recursive Path Orders

2.8.14. Definition. If $\gtrsim$ is a quasi order on F, then the *recursive path order* $>_{\mathrm{rpo}}$ on $T(F,V)$ is given by $a = f(a_1,\ldots,a_n) >_{\mathrm{rpo}} g(b_1,\ldots,b_m) = b$ iff

- there exists i such that $a_i >_{\mathrm{rpo}} b$,
- or $f > g$ and for all j we have $s >_{\mathrm{rpo}} b_j$,
- or $f \approx g$ and $\{a_1,\ldots,a_n\} \gg_{\mathrm{rpo}} \{b_1,\ldots,b_m\}$

and $a = f(a_1,\ldots,a_n) >_{\mathrm{rpo}} x$ if $x \in \mathrm{Var}(a)$.

2.8.15. Proposition. $>_{\mathrm{rpo}}$ is a quasi order on $T(F,V)$.

2.8.16. Theorem. $>_{\mathrm{rpo}}$ is a reduction order.

This is again a deep theorem. It works because of Kruskal's theorem 2.8.3, and because the multiset order respects well-foundedness (proposition 1.4.17).

Chapter 3

Lambda Calculus and Combinatory Logic

3.1 Introduction

Lambda Calculus is an abstract reduction system that describes rewriting terms with *bound* variables. (So it's *not* a TRS.) It can be regarded as the syntactic theory of function abstraction and application. In this respect, it is a "quintessential" system because *all* its terms denote functions.

Instead of the extensional view as a set of (argument, value)-pairs, in lambda calculus, functions are described intensionally, as algorithms that compute values. So lambda calculus is a model of computation. Its complexity is equivalent to other models such as Turing machines or recursive functions. As is the case with these models, some properties of computations are necessarily undecidable.

Combinatory logic is a reduction system that is closely related to lambda calculus. The systems share the view that "everything is a function". However, terms in combinatory logic do *not* contain variables at all. So combinatory logic *is* a TRS. It is in fact an orthogonal TRS and therefore enjoys nice properties.

Yet there are translations between combinatory logic and lambda calculus, showing that both systems are computationally equivalent. Therefore undecidability results from lambda calculus carry over to combinatory logic (and to TRS in general).

In the main part of this thesis, we will consider the system **CL**(**S**) which is a subsystem of combinatory logic. We will show that certain undecidable properties of the full system become manageable in the subsystem.

The presentation of lambda calculus and combinatory logic follows Barendregt [Bar84], of course.

3.2 Lambda Terms

3.2.1. Definition. The set Λ of lambda terms is given by

1. (Variables) for $i > 0 : x_i \in \Lambda$
2. (Abstractions) for $i > 0, M \in \Lambda : \lambda x_i.M \in \Lambda$
3. (Applications) for $M, N \in \Lambda : (M \cdot N) \in \Lambda$.

The notions for terms introduced in section 2.2 will be carried over to Λ. There is one interesting point, however: in an abstraction $\lambda x.M$, we regard M as immediate subterm at position (1), and the variable x does not count as a subterm.
We will follow these standard conventions when writing down lambda terms:

- Application is left-associative, so $M \cdot N \cdot P$ stands for $(M \cdot N) \cdot P$.
- The application symbol is completely omitted, so (MN) stands for $(M \cdot N)$.
- Nested abstractions are combined into one λ symbol, so $\lambda xy.x$ stands for $\lambda x.(\lambda y.x)$.

3.2.2. Definition. The set $\mathrm{FVar}(t)$ of *free* variables of a lambda term t is

1. $\mathrm{FVar}(x_i) = \{x_i\}$
2. $\mathrm{FVar}(\lambda x_i.M) = \mathrm{FVar}(M) \setminus \{x_i\}$
3. $\mathrm{FVar}(MN) = \mathrm{FVar}(M) \cup \mathrm{FVar}(N)$

3.2.3. Definition. A lambda term M is *closed* iff $\mathrm{FVar}(M) = \emptyset$. The set of all closed terms is Λ^0.

3.2.4. Definition. A variable x_i is *bound* in a term t if there is some subterm $\lambda x_i.M$ of t.

3.3 Beta reduction

3.3.1. Definition. A lambda term of the form $(\lambda x.M)N$ is called *redex.* Its *contractum* is the term $M[x \leftarrow N]$. The relation β consists of all pairs (redex, contractum).

Again notations for substitutions for terms, as defined in section 2.3, are carried over to Λ.
Note however that in $M[x \leftarrow N]$ it is understood that only free occurrence of x in M are replaced, and that no free variable of N gets bound. This can be achieved by some renaming of variables. We omit details.

3.3.2. Definition. The closure of β w.r.t. contexts (see definition 2.8.5) is called $\rightarrow_\beta$.

We write $=_\beta$ for $\leftrightarrow_\beta^*$.
We often omit the subscript and just write $\rightarrow$ or $=$.

3.4 Confluence

As had been mentioned for ARS, confluence is a desirable property which ensures that computations are well-behaved.

3.4.1. Theorem (Church, Tait, Martin-Löf). $\twoheadrightarrow$ is confluent.

There are several ways to prove this theorem.

(Just proving local confluence is easy, but not enough, as the system is non-terminating.)

We mention one possibility that is similar to that of proposition 2.5.5, thereby showing that Λ in some respect behaves like an orthogonal TRS.

A *development* $X \twoheadrightarrow_{\text{dev}} Y$ of a term X containing *marked* redexes is a reduction chain that only reduces redexes that were marked in X. During such a reduction, existing markers may get copied but no new markers are introduced.

3.4.2. Proposition (Finite Developments). Developments $X \twoheadrightarrow_{\text{dev}} Y$ are always finite. The result Y of a *complete* development $X \twoheadrightarrow_{\text{cpl}} Y$ which does not contain markers any more, is unique.

Confluence then follows from

3.4.3. Proposition. $\twoheadrightarrow_{\text{cpl}}$ has the diamond property.

and the obvious

3.4.4. Proposition. $\rightarrow \subseteq \twoheadrightarrow_{\text{cpl}} \subseteq \twoheadrightarrow^*$

3.5 Non-erasing lambda terms

In analogy to non-erasing TRS, we define a subset of non-erasing lambda terms.

3.5.1. Definition. The set $\Lambda_{\mathbf{I}}$ is given by

1. (Variables) for $i > 0 : x_i \in \Lambda_{\mathbf{I}}$
2. (Abstractions) for $i > 0, M \in \Lambda_{\mathbf{I}}$: if $x_i \in \text{FVar}(M)$, then $\lambda x_i.M \in \Lambda_{\mathbf{I}}$
3. (Applications) for $M, N \in \Lambda_{\mathbf{I}} : (MN) \in \Lambda_{\mathbf{I}}$.

3.5.2. Proposition. $\Lambda_{\mathbf{I}}$ is forward closed under $\twoheadrightarrow$: If $M \in \Lambda_{\mathbf{I}}$ and $M \rightarrow N$, then $N \in \Lambda_{\mathbf{I}}$.

The subsystem $\Lambda_{\mathbf{I}}$ bears resemblance to an orthogonal and non-erasing TRS.

3.5.3. Theorem (Conservation Theorem). In $\Lambda_{\mathbf{I}}$, normalization and termination coincide.

3.6 Standardization

3.6.1. Definition. Each lambda term M is of the form $\lambda x_1.\cdots.\lambda x_n.HM_1\ldots M_m$ where either H is a variable and $n\geq 0$, $m\geq 0$, or $H=\lambda y.M_0$ and $n\geq 0$, $m\geq 1$.

In the first case, M is in *head normal form.* In the second case, HM_1 is called the *head redex* of M. Any redex in one of the M_i is called *internal.*

3.6.2. Definition. An expression is in *head normal form* iff it has no head redex.

3.6.3. Definition. A *head reduction* $\rightarrow_h$ contracts only head redexes; an *internal reduction* $\rightarrow_i$ contracts only internal redexes.

3.6.4. Definition. A *standard* reduction $M\longrightarrow\!\!\!\!\rightarrow_s P$ is a (possibly empty) sequence of head reductions $M\longrightarrow\!\!\!\!\rightarrow_h N=N_0N_1\ldots N_n$ followed by internal reductions that are standard w.r.t. the N_i.

3.6.5. Theorem (Curry and Feys). For all reductions $M\twoheadrightarrow N$, there exists a standard reduction $M\longrightarrow\!\!\!\!\rightarrow_s N$.

Proof. First one shows that each reduction $M\twoheadrightarrow N$ can be re-arranged in the form $M\longrightarrow\!\!\!\!\rightarrow_h P\longrightarrow\!\!\!\!\rightarrow_i N$.

Then $N=N_0N_1\ldots N_k$, and also $P=P_0P_1\ldots P_k$. Now the theorem is applied to $P_i\twoheadrightarrow N_i$ recursively.

This is a finite process by well-founded induction on the size of the N_i. □

3.6.6. Corollary. M has a head normal form N iff the head reduction sequence $M\longrightarrow\!\!\!\!\rightarrow_h\ldots$ terminates.

Proof. Assume $M\twoheadrightarrow N$ where $N=N_0N_1\ldots N_k$ is in head normal form. We have $M\longrightarrow\!\!\!\!\rightarrow_h P\longrightarrow\!\!\!\!\rightarrow_i N$ by the standardization theorem, but then P must be in head normal for as well, and $M\longrightarrow\!\!\!\!\rightarrow_h P$ is a terminating head reduction chain. □

Head reductions and the standardization theorem, albeit for combinatory logic, will be used when top-termination of $\mathbf{CL}(\mathbf{S})$ is considered in chapter 7.

3.7 Lambda-definable Functions

Lambda calculus can simulate computations on numbers. The presentation here closely follows [Bar84].

3.7.1. Definition (lambda-definable). Given a suitable encoding of numbers $\lceil\cdot\rceil:\mathbb{N}\rightarrow\Lambda^0$, the function $f:N^k\rightarrow N$ is *lambda-defined* by $f_\lambda\in\Lambda^0$ if $f_\lambda\lceil x_1\rceil\ldots\lceil x_k\rceil=\lceil f(x_1,\ldots,x_k)\rceil$

We mention the difficulty of representing undefined values of computations.

With partial recursive functions, an undefined sub-computation implies the failure of the whole computation. This has to be simulated in lambda calculus.

One has the option of using terms from $\Lambda^0_{\mathbf{I}}$ for encoding. This had been done by Church originally. [Chu41] By the conservation theorem, each subterm of a term with normal form itself has a normal form. So we might represent "undefined" by any term without normal form. But the translation $\lceil 0\rceil$ does not seem to fit well into $\Lambda^0_{\mathbf{I}}$ and therefore creates technical problems.

The other option is to use terms from Λ^0. There, $\lceil 0 \rceil$ can easily be encoded. But in Λ^0, there are terms with normal form that have subterms without normal form. So "undefined" must be expressed by some other class of terms. Here, *unsolvable* terms (terms without head normal form) are appropriate. But then another technical problem arises. The composition of encodings of functions not necessarily corresponds to the encoding of the composition.

As we just intend to give the flavour of lambda-definability, we avoid all the technicalities by restricting ourselves to total functions from now on.

We first define a suitable encoding $\mathbb{N} \to \Lambda^0$. (There are many of them. We don't define the exact meaning of *suitable*.)

3.7.2. Definition (Truth values). $\mathbf{T} = \lambda xy.x$, $\mathbf{F} = \lambda xy.y$

3.7.3. Definition (Pairing, selectors). $\langle M, N \rangle = \lambda x.xMN$, $\langle P \rangle_1 = P\,\mathbf{T}$, $\langle P \rangle_2 = P\,\mathbf{F}$.

3.7.4. Proposition. $\langle\langle M, N \rangle\rangle_1 \twoheadrightarrow M$, $\langle\langle M, N \rangle\rangle_2 \twoheadrightarrow N$.

3.7.5. Definition (Numbers). $\lceil 0 \rceil = \lambda x.x$, $\lceil n+1 \rceil = \langle \mathbf{F}, \lceil n \rceil \rangle$

Note that numbers are distinct normal forms.

3.7.6. Proposition. $\lceil 0 \rceil\,\mathbf{T} = \mathbf{T}$, $\lceil n+1 \rceil\,\mathbf{T} = \mathbf{F}$

Now we do some basic computations with numbers.

3.7.7. Proposition. There are expressions $\mathbf{succ}, \mathbf{pred}, \mathbf{zero} \in \Lambda^0$ such that $\mathbf{succ}\,\lceil n \rceil = \lceil n+1 \rceil$, $\mathbf{pred}\,\lceil n+1 \rceil = \lceil n \rceil$, $\mathbf{zero}\,\lceil 0 \rceil = \mathbf{T}$, $\mathbf{zero}\,\lceil n+1 \rceil = \mathbf{F}$.

Proof. Define $\mathbf{succ} = \lambda x.\langle \mathbf{F}, x \rangle$, $\mathbf{pred} = \lambda x.x\,\mathbf{F}$, $\mathbf{zero} = \lambda x.x\,\mathbf{T}$. ☐

Obviously constant functions, selectors, the successor function and the substitution operator are lambda-definable.

To define iterative computations, we employ recursion.

3.7.8. Proposition. Each lambda-term M has a fixed point F such that $MF =_\beta F$.

It can be uniformly constructed:

3.7.9. Definition (Turing's Fixpoint combinator). $\Theta = HH$ where $H = \lambda hf.f(hhf)$.

3.7.10. Proposition. For any $f \in \Lambda$, $\Theta f \twoheadrightarrow f(\Theta f)$, that is, Θf is a fixed point of f.

Proof. $\Theta f = HHf \twoheadrightarrow f(HHf) = f(\Theta f)$ ☐

3.7.11. Proposition (Primitive recursion). If $g : \mathbb{N}^k \to \mathbb{N}$ and $h : \mathbb{N}^{k+2} \to \mathbb{N}$ are lambda-definable, then $f : \mathbb{N}^{k+1} \to \mathbb{N}$ with

$$\begin{array}{rcl} f(0, x_1, \ldots, x_k) & = & g(x_1, \ldots, x_k) \\ f(y+1, x_1, \ldots, x_k) & = & h(f(y, x_1, \ldots, x_k), y, x_1, \ldots, x_k) \end{array}$$

is lambda-definable.

Proof. $f_\lambda = \Theta(\lambda F.\lambda\, y\, x_1 \ldots x_k.\mathbf{zero}\, x\, (g_\lambda\, x_1 \ldots x_k)(h_\lambda\, (F\, (\mathbf{pred}\, y)\, x_1 \ldots x_k)\, x_1 \ldots x_k))$. ☐

3.7.12. Proposition (Minimalization). If $g : \mathbb{N}^{k+1} \to \mathbb{N}$ is lambda-definable, then $f : \mathbb{N}^k \to \mathbb{N}$ with

$$f(x_1, \ldots, x_k) = \min\{y \mid g(y, x_1, \ldots, x_k) = 0\}$$

is lambda-definable.

Proof. $f_\lambda = h \lceil 0 \rceil$ where $h = \Theta(\lambda H.\lambda\, y\, x_1 \ldots x_k.\mathbf{zero}(g_\lambda\, y\, x_1 \ldots x_k)\, y\, (H\, (\mathbf{succ}\, y)\, x_1 \ldots x_k))$ □

The previous constructions show:

3.7.13. Theorem (Church). The class of lambda-definable functions is exactly the class of partial recursive functions.

This indicates that questions on lambda terms are hard to answer in general. For instance it is undecidable whether two lambda term have equal normal forms. Otherwise we could decide whether two program texts encode the same function.

3.8 Combinatory Logic

3.8.1. Definition. Combinatory logic with the base combinators $\{C_1, \ldots, C_n\}$, denoted by

$$\mathbf{CL}\,(C_1, \ldots, C_n)$$

is a TRS on the signature $\{\cdot, C_1, \ldots, C_n\}$, where $a\,(\cdot) = 2$ and $a\,(C_i) = 0$ and for each i there is exactly one rule $C_i x_1 \ldots x_k \to M_i$, where M_i is a *pure* term.

3.8.2. Definition. A *pure* term consists entirely of applications and variables, and contains no constants.

We follow the lambda calculus conventions on left-associativity of application, and on omitting the application symbol.
The following combinators were introduced by Curry, Schönfinkel et al.

$$\begin{array}{l}
\mathbf{I}\ x \to x; \mathbf{K}\ x\ y \to x; \mathbf{S}\ x\ y\ z \to x\ z\ \ (y\ z)\\
\mathbf{B}\ x\ y\ z \to x\ \ (y\ z)\,; \mathbf{C}\ x\ y\ z \to x\ z\ y; \mathbf{J}\ x\ y\ z\ w \to x\ y\ \ (x\ w\ z)\\
\mathbf{W}\ x\ y \to x\ y\ y; \mathbf{L}\ x\ y \to x\ \ (y\ y)
\end{array}$$

From the syntactic restrictions of the term algebras and the reduction rules it follows that combinatory logic systems are orthogonal, so they have the Church-Rosser property. Yet they can be quite complicated.
And they are of great entertainment value, see for instance the ornithological book by Smullyan [Smu85]. The appendix of Hindley's and Seldin's text [HS86] tells you how to care for your pet combinators.

3.9 Completeness

3.9.1. Definition. A combinatory TRS is called *combinatorially complete* iff for all $n > 0$ and for all pure terms M with $\mathrm{FVar}(M) \subseteq \{x_1, \ldots, x_n\}$, there is a ground term N such that $N x_1 \ldots x_n \twoheadrightarrow M$.

3.9.2. Theorem. $\mathbf{CL}\,(\mathbf{S}, \mathbf{K}, \mathbf{I})$ is combinatorially complete.

Proof. We define a function $\lfloor\cdot\rfloor_x$ such that $\lfloor M\rfloor_x\ x \twoheadrightarrow M$. The argument of $\lfloor\cdot\rfloor_x$ is any term, and the result is a term where x does not occur.

1. $\lfloor x\rfloor_x = \mathbf{I}$.
2. If $x \ntrianglelefteq M$, then $\lfloor M\rfloor_x = \mathbf{K}\ M$.
3. $\lfloor MN\rfloor_x = \mathbf{S}\ \lfloor M\rfloor_x\ \lfloor N\rfloor_x$.

Now we set $N = \lfloor \ldots\ \lfloor\lfloor M\rfloor_{x_1}\rfloor_{x_2}\ \ldots\rfloor_{x_n}$. Beware: the size of N compared to M might grow exponentially with n. □

Since $\mathbf{S}\ \mathbf{K}\ \mathbf{K}\ x \to \mathbf{K}\ x\ \ (\mathbf{K}\ x) \to x \leftarrow \mathbf{I}\ x$, we can simulate the combinator $\mathbf{I}$ by $\mathbf{S}$ and $\mathbf{K}$:

3.9.3. Corollary. $\mathbf{CL}\,(\mathbf{S}, \mathbf{K})$ is combinatorially complete.

3.10 Non-erasing Combinators

Similar to $\Lambda_{\mathbf{I}}$, there is a non-erasing version of $\mathbf{CL}$, called $\mathbf{CL_I}$, where only combinators with non-erasing reduction rules are allowed. Still there are complete systems in $\mathbf{CL_I}$.

As was the case in $\Lambda_{\mathbf{I}}$, the conservation theorem also holds for $\mathbf{CL_I}$. This can also be derived from proposition 2.6.5.

Although $\mathbf{CL_I}$ is contained in $\mathbf{CL}$, is powerful enough:

3.10.1. Definition. A combinatory TRS is called $\mathbf{CL_I}$-*complete* iff for all $n > 0$ and for all pure terms M with $\mathrm{FVar}(M) = \{x_1, \ldots, x_n\}$, there is a ground term N such that $N x_1 \ldots x_n \twoheadrightarrow M$.

Notice the subtle difference to definition 3.9.1.

3.10.2. Theorem. $\mathbf{CL_I}\,(\mathbf{S}, \mathbf{B}, \mathbf{C}, \mathbf{I})$ is $\mathbf{CL_I}$-complete.

Proof. The definition of $\lfloor\cdot\rfloor_x$ is slightly modified. We define $\lfloor M\rfloor_x$ only for the case that $x \trianglelefteq M$.

1. M is a variable. Then $M = x$, and $\lfloor x\rfloor_x = \mathbf{I}$. The case $M \neq x$ is impossible because x must occur in M.
2. M is an application PQ. Because of $x \trianglelefteq M$, only the following three cases are possible:
   (a) $x \trianglelefteq P$ and $x \ntrianglelefteq Q$. Then $\lfloor PQ\rfloor_x = \mathbf{C}\ \lfloor P\rfloor_x\ Q$.
   (b) $x \ntrianglelefteq P$ and $x \trianglelefteq Q$. Then $\lfloor PQ\rfloor_x = \mathbf{B}\ P\ \lfloor Q\rfloor_x$.
   (c) $x \trianglelefteq P$ and $x \trianglelefteq Q$. Then $\lfloor PQ\rfloor_x = \mathbf{S}\ \lfloor P\rfloor_x\ \lfloor Q\rfloor_x$.

□

3.11 Combinatory Logic and Lambda calculus

There are translations between Λ^0 and $\mathbf{CL}\,(\mathbf{S},\mathbf{K})$, as well as between $\Lambda^0_{\mathbf{I}}$ and $\mathbf{CL_I}\,(\mathbf{S},\mathbf{B},\mathbf{C},\mathbf{I})$.

3.11.1. Definition. The mapping $\langle\cdot\rangle_\Lambda : \mathbf{CL} \to \Lambda^0$ is given by

$$\begin{aligned}
&\langle\mathbf{S}\rangle_\Lambda = \lambda\, x\; y\; z\; .\; x\; z\; (y\; z)\,; \langle\mathbf{K}\rangle_\Lambda = \lambda\, x\; y\; .\; x; \langle\mathbf{I}\rangle_\Lambda = \lambda\, x\; .\; x\\
&\langle\mathbf{B}\rangle_\Lambda = \lambda\, x\; y\; z\; .\; x\; (y\; z)\,; \langle\mathbf{C}\rangle_\Lambda = \lambda\, x\; y\; z\; .\; x\; z\; y\\
&\langle M \,\hat{}\, N\rangle_\Lambda = \langle M\rangle_\Lambda\; \langle N\rangle_\Lambda
\end{aligned}$$

3.11.2. Proposition. If $M \to_{\mathbf{CL}} N$, then $\langle M\rangle_\Lambda \to^+_\beta \langle N\rangle_\Lambda$.

This translation is not very tight, however, since one reduction step in $\mathbf{CL}$ corresponds to one *or more* steps in Λ, and the intermediate steps in Λ don't correspond to terms in $\mathbf{CL}$. Moreover, normal forms, and normalization, are not preserved.

3.11.3. Definition. The mapping $\langle\cdot\rangle_{\mathbf{CL}} : \Lambda^0 \to \mathbf{CL}$ resp. $\Lambda^0_{\mathbf{I}} \to \mathbf{CL_I}$ is given by

$$\langle M\; N\rangle_{\mathbf{CL}} = \langle M\rangle_{\mathbf{CL}}\; \langle N\rangle_{\mathbf{CL}}; \langle\lambda\, x\; .\; M\rangle_{\mathbf{CL}} = \lfloor\langle M\rangle_{\mathbf{CL}}\rfloor_x$$

where $\lfloor\cdot\rfloor_x$ is defined in the proofs of theorems 3.9.2 or 3.10.2, respectively. $\langle\cdot\rangle_{\mathbf{CL}}$ is in fact a mapping from a lambda term M to $\mathbf{CL}$-terms with variables $\mathrm{FVar}(M)$.

3.11.4. Proposition. If $M \to_\beta N$, then $\langle M\rangle_{\mathbf{CL}} \to^+_{\mathbf{CL}} \langle N\rangle_{\mathbf{CL}}$.

This is even less tight, and not at all an "inverse" of $\langle\cdot\rangle_\Lambda$.
For a more detailed analysis of the relation between these two notions of reduction, see [Hin77].

Even if the translations do not faithfully represent single reduction steps, they show that Λ and $\mathbf{CL}$ have equivalent computational power. So from theorem 3.7.13 we derive

3.11.5. Theorem. The $\mathbf{CL}\,(\mathbf{S},\mathbf{K})$-definable functions are exactly the partial recursive functions.

This shows that $\mathbf{CL}\,(\mathbf{S},\mathbf{K})$ is hard as well. For instance, it is not decidable whether two terms have the same normal form.
As $\mathbf{CL}\,(\mathbf{S},\mathbf{K})$ is an instance of an orthogonal TRS, this question is not decidable for this class of TRS. There are even stronger results.

Returning, for a moment, to the topic of this thesis, it seems not clear what functions are $\mathbf{CL}(\mathbf{S})$-definable. There isn't a self-evident encoding of the naturals, even. Since $\mathbf{CL}(\mathbf{S})$ isn't combinatorially complete, severe restrictions are to be expected, and one could indeed hope that termination could be decidable.

Chapter 4

Finite Automata and Rational Languages

4.1 Introduction

The concept of formal languages was originally invented to eventually capture natural languages as well. Or, even more, to explore machine understanding of language. So the idea of automatons operating in formal languages was at hand.

A finite automaton reads words and performs state transitions. It can only read a word once from left to right, and it only has a finite number of states available.

An automata accepts a word if it starts reading from a prescribed state and finishes reading in a set of prescribed final states. The languages thus acceptable are called *regular*, or *rational*.

In Chomsky's hierarchy of word languages, rational languages are the most simple class. They have many nice properties. Especially, they are closed under Boolean operations.

Finite automata can be generalized so that they operate on arbitrary term algebras. The notions of acceptability and rationality carry over, as well as the fundamental results, including closure under Boolean operations.

The material presented in this chapter is taken from Gécseg's and Steinby's standard book [GS84], except for the section on relations between rational languages and term rewriting systems which is collected from recent publications.

Finite Automata in `RX`

This chapter also serves as a guide to the *implementation* of the `RX` system (whereas chapter A is a guide to the *usage* of that program). Most of the definitions and results given here are directly implemented in `RX`, and we provide pointers to the source code wherever possible.

4.2 Automata and Languages

Automata

The state transitions of an automaton can be seen as relations between terms whose nodes are labeled with the states. So it's no surprise that we use TRS to define finite automata.

We extend a given signature by new nullary symbols which correspond to the states of the automaton.

4.2.1. Definition. An non-deterministic automaton (R, S, S') on the signature (F, a) is a TRS R on the signature $(F \cup S, a)$ where $S = \{s_1, s_2, \ldots\}$ are new constants (with $a(s_i) = 0$), called *states* and a subset S' of S, called *final* or *accepting* states and all rules have a specific form:

The automaton is called *bottom-up* if all rules have the form

$$f(x_1, \ldots, x_n) \to y \quad \text{where} \quad x_i \in S, y \in S$$

The automaton is called *top-down* if all rules have the form

$$y \to f(x_1, \ldots, x_n) \quad \text{where} \quad x_i \in S, y \in S$$

An *epsilon move* is a rule

$$x \to y \quad \text{where} \quad x \in S, y \in S$$

Note that all rules are ground rules. So a finite automaton is a ground TRS.

4.2.2. Definition. The automaton is called *finite* iff S is finite. (The set of rules is then finite as well.)

4.2.3. Definition. A bottom-up automaton is called *deterministic* iff no two left hand sides of the rules coincide.

4.2.4. Definition. A bottom-up automaton is called *complete* iff all possible left hand sides really occur in the rule set.

Implementation

The file `FAtypes.hs` contains data declarations for automata:

1. `BDFA` : bottom-up deterministic
2. `BNFA` : bottom-up non-deterministic
3. `TNFA` : top-down non-deterministic
4. `ETNFA` : top-down non-deterministic with epsilon moves

4.2.5. Example. The following piece of code (taken from that file) declares a `BDFA` with state type `a` as a parameter.

```
data BNFA a = BNFA
        TCons           -- what algebra we're in
        (Set a)         -- all states
        (Set a)         -- accepting states
        (FiniteMap (STerm a) (Set a))   -- transition table
```

Rational Languages

4.2.6. Definition. The language accepted by a bottom-up automaton (R, S, S') on the signature F is

$$\{t \mid t \in T(F), \exists s \in S' : t \twoheadrightarrow_R s\}$$

4.2.7. Definition. The language accepted by a top-down automaton (R, S, S') on the signature F is

$$\{t \mid t \in T(F), \exists s \in S' : s \twoheadrightarrow_R t\}$$

4.2.8. Definition. A language is called *rational* iff it is accepted by some finite automaton.

4.2.9. Definition. Two automata are *equivalent* iff they accept the same language.

4.2.10. Definition. A language L in (F, a) defines an equivalence $\cong_L$ on terms from $T(F)$ by

$$t_1 \cong_L t_2 \quad \text{iff} \quad \forall t \in T(F), p \in \mathrm{Pos}(t) : t\,[p \leftarrow t_1] \in L \Leftrightarrow t\,[p \leftarrow t_2] \in L$$

4.2.11. Theorem. L is rational if $\cong_L$ has finite index.

Proof. Take the equivalence classes of $\cong_L$ as the states of an automaton that accepts L. □

4.3 Variants of Finite Automata

Extended Automata

Syntactic restrictions can be lifted somewhat. This makes writing down automata more convenient.

4.3.1. Definition. An extended automaton on the signature (F, a) is a TRS on the signature $(F \cup S \cup \{u\}, a)$ where S is the set of states as above, and u is a new binary symbol (to be read as "union").
All rules have the form

$$y \to z \quad \text{where} \quad y \in S, z \in T(F \cup S \cup \{u\})$$

4.3.2. Proposition. Each extended automaton has an equivalent (non-extended) automaton.

Proof. Nested constructor applications are be removed by introducing new states. Example:

$$y \to f(\ldots, g(\ldots), \ldots) \quad \text{is replaced by} \quad y \to f(\ldots, z, \ldots); \quad z \to g(\ldots)$$

The u symbol is resolved by introducing two epsilon moves.

$$y \to u(a, b) \quad \text{is replaced by} \quad y \to a \quad \text{and} \quad y \to b$$

□

The file `Gram2FA.hs` contains the function `gram` that converts an expression that describes an extended rational automaton into an `ETNFA`, employing the algorithm just given.

Relations between different automata

4.3.3. Proposition. For each non-deterministic bottom-up (resp. top-down) automaton with epsilon moves there is an equivalent non-deterministic bottom-up (resp. top-down) automaton without epsilon moves.

Proof. Replace the left (right) hand side of each epsilon rule by its epsilon closure, i. e. all states that are reachable by epsilon moves alone. □

4.3.4. Proposition. For each non-deterministic bottom-up automaton there is an equivalent non-deterministic top-down automaton, and vice versa.

Proof. Just reverse the direction of the rules of the automaton's TRS. Note that determinism is lost in such a transformation. □

These conversions are implemented in the file `FAconv.hs`.

4.4 Canonical forms of automata

Trim automata

For this section we adopt the view that automata *generate* (rather that *accept*) a language, i. e. they produce a set of ground terms.

An automaton may contain useless states. A state is called *useless* if it does not take part in the creation of terms.

4.4.1. Definition. In a bottom-up automaton (R, S, S'), the set $P \subseteq S$ of *reachable* states is the smallest set P with the property

$$\text{if} \quad f(q_1, \ldots, q_n) \to p \in R \quad \text{and} \quad \forall i : q_i \in P \quad \text{then} \quad p \in P$$

So P contains all states that produce a non-empty language. Especially, all states that produce constants (with $n = 0$) are elements of P.

Note that the set S' of accepting states doesn't play any role in the definition of P.

4.4.2. Definition. In a bottom-up automaton (R, S, S'), the set $Q \subseteq S$ of *connected* states is the smallest set Q with the properties

$$S' \subset Q$$
$$\text{if} \quad f(q_1, \ldots, q_n) \to p \in R \quad \text{and} \quad p \in Q \quad \text{then} \quad \forall i : q_i \in Q$$

So Q contains the accepting states, and all states that are connected to them.

4.4.3. Definition. An automaton is called *trim* if all states are reachable and connected.

4.4.4. Proposition. For each bottom-up automaton, there is an equivalent trim automaton.

Proof. First delete all states that are not reachable. Then, in remaining automaton, delete all states that are not connected. (The order of these steps is important here.) □

These functions are implemented in the file `FAuseful.hs`.

Determinism

4.4.5. Theorem. For each non-deterministic bottom-up automaton without epsilon moves, there is an equivalent deterministic and complete bottom-up automaton without epsilon moves.

Proof. The set of states of the deterministic automaton is the power set of the states of the non-deterministic automaton. The set of accepting states consists of those subsets of S that contain an element from S'. □

Note that the number of states is likely to explode.

In general, top-down automata cannot be made deterministic.

The implementation is contained in the file `FAdet.hs`. To save time and space, we only generate those elements of the power set that occur as states in a trim deterministic automaton.

Minimal deterministic automata

4.4.6. Definition. A *homomorphism* from an automaton A_1 to A_2 is a mapping from the states of A_1 to the states of A_2 that respects the state transitions and the set of accepting states.

4.4.7. Theorem. For each language L, there is a unique minimal (w.r.t. homomorphism) deterministic bottom-up automaton that recognizes it.

Proof. It is the automaton whose states are the classes of $\cong_L$. □

Note that the deterministic automaton obtained from the power set construction need not be minimal. However:

4.4.8. Proposition. The minimal automaton is homoeomorphically embedded in the power set automaton. It is found by identifying states that are congruent.

Minimalization is implemented in the file `FAmin.hs`. We follow the procedure given in [GS84]. It approximates $\cong_L$, starting with just two classes, containing accepting and non-accepting states. This relation is successively refined by looking for incongruent states, and singling them out into new equivalence classes.

4.4.9. Remark. It would be convenient to have minimal non-deterministic automata, too. But they are not unique, and moreover hard to find. See the papers [ADN92] and [MP95] that describe the situation, and some constructions (for word automata).

They use the idea to construct the minimal deterministic automaton for the mirror image of the given word language. For term languages, the mirror image would correspond to a top-down automaton. However, such an automaton cannot be made deterministic. This shows the difficulty of the problem.

4.5 Operations on rational languages

4.5.1. Proposition. Rational languages are closed under term formation.

This means that for rational languages L_i, the language $L = \{f(x_1, \dots, x_n) \mid x_i \in L_i\}$ is rational.

Proof. The automaton that accepts L has rules $l \to f(l_1, \ldots, l_n)$ for all tuples of accepting states l_i of the automata for L_i. □

The automaton for L need not be deterministic even if all of the L_i were.

4.5.2. Proposition. Rational languages are closed under union.

Proof. Represent the languages L_1 and L_2 by automata with start sets S'_1 and S'_2. The automaton that accepts $L_1 \cup L_2$ has state set $\{s\} \cup S_1 \cup S_2$ and one accepting state s with epsilon moves to (or from) all elements of S'_1 and S'_2. □

This goes through for either bottom-up or top-down automata.
Determinism is lost.
The implementation is in the file `FAunion.hs`.

4.5.3. Proposition. Rational languages are closed under intersection.

Proof. Represent the languages L_1 and L_2 by automata without epsilon moves. Name their state sets S_1 and S_2, and their accepting states S'_1 and S'_2. The automaton that accepts $L_1 \cap L_2$ has state set $S_1 \times S_2$ and accepting states $S'_1 \times S'_2$ □

This goes through for either bottom-up or top-down automata.
If the input automata were deterministic, the result will be, too.
The implementation is in the file `FAintersect.hs`.

4.5.4. Proposition. Rational languages are closed under complement.

Proof. Do the power set construction, making the automaton deterministic. But then call those subsets accepting that don't contain elements from S'. □

The implementation is in the file `FAneg.hs`.

4.5.5. Proposition. Rational languages are closed under set difference.

Proof. Compute $L_1 \setminus L_2$ as the intersection of L_1 with the complement of L_2. □

The implementation is in the file `FAminus.hs`.

4.6 Term Rewriting Systems and Rational Languages

The relationship between TRS and rational languages is an area of ongoing research. This section gives a very brief account of basic results.

Linearization

One point of interest is the set of normal forms in a given TRS. With this respect, only the left hand sides of the rules of the TRS are looked at. The following is a simple observation.

4.6.1. Proposition. If a TRS R is left linear, then the set of R-normal forms is rational.

Proof. This is due to closure properties of rational languages. For any term algebra, the set of all terms is rational. For left linear TRS, the set of all redexes is rational. The set of all terms containing redexes is rational. The set of all terms not containing redexes is rational. This is exactly the set of normal forms. $\square$

See also the example in section A.7 that shows how such a computation is done in the system `RX`.

If the TRS is not linear, then in some cases it is still possible to describe the normal forms, as was shown by Kucherov and Tajine [KT92], Hofbauer and Huber [HH92], Vágvölgyi and Gilleron [VG92].

4.6.2. Theorem. For a TRS R it is decidable whether the set of all R-normal forms is rational. If it is, then a left linear TRS R' can be constructed such that R'-normal forms are R-normal forms.

As we will be looking at $\mathbf{CL}(\mathbf{S})$, which is left linear, we don't need linearization.

Preservation of Regularity

Ground normal forms only depend on left hand sides of rewrite rules, while their right hand sides are ignored. As soon as rewriting really comes into play, things get much more complicated. The set of successors of a rational language isn't necessarily rational, and it is undecidable whether a TRS preserves rationality.

4.6.3. Definition. Let $\to$ be a relation, and L a set of terms. Then

- successors: $\Delta^{\to}(L) = \{t' \mid t \in L, t \to t'\}$,
- predecessors: $\Delta^{\leftarrow}(L) = \{t' \mid t \in L, t \to t'\}$,
- forward closure: $\Delta^{\twoheadrightarrow}(L) = \{t' \mid t \in L, t \twoheadrightarrow t'\}$.
- backward closure: $\Delta^{\twoheadleftarrow}(L) = \{t' \mid t \in L, t \twoheadrightarrow t'\}$.
- congruence closure: $\Delta^{=}(L) = \{t' \mid t \in L, t \leftrightarrow^* t'\}$.

Gilleron and Tison prove [GT95]

4.6.4. Theorem. The following problem is undecidable:
input: TRS R, **question**: is $\Delta_R^{=}(L)$ rational for all rational languages L?

Rationality *is* preserved for some restricted classes of TRS.
Dauchet and Tison show (see [DT90])

4.6.5. Theorem. If R is a ground TRS, and L is rational, then $\Delta_R^{\twoheadrightarrow}(L)$ and $\Delta_R^{=}(L)$ are rational.

This is not a surprise since finite automata and ground rewrite systems are closely related.
Salomaa considers monadic TRS (see [Sal88])

4.6.6. Definition. A TRS is *monadic* iff for every rule $l \to r$, $\mathrm{depth}(l) \geq 1$ and $\mathrm{depth}(r) \leq 1$.

4.6.7. Theorem. If R is a right linear monadic TRS, and L rational, then $\Delta_R^{\twoheadrightarrow}(L)$ is rational.

Coquide et. al. generalize this to semi-monadic TRS (see [CDGV91])

4.6.8. Definition. A TRS is *semi-monadic* iff for every rule $l \to r$, either r is a variable or a term $f(t_1, \ldots, t_n)$ where each t_i is a variable or a ground term.

4.6.9. Theorem. If R is a linear semi-monadic TRS, and L rational, then $\Delta_R^{\twoheadrightarrow}(L)$ is rational.

We will soon be looking at **CL(S)**, which does not belong to any of these classes. (Neither does **CL(S)** with the arrow pointing backwards.) Therefore it is somehow surprising that rational languages are usable in describing certain properties of **CL(S)**.

Approximation of forward closure

Even if $\Delta^{\twoheadrightarrow}(L)$ for rational L is not rational, we might try to approximate it by a rational superset. We follow the presentation by Genet [Gen97].

Let R be the term rewriting system under consideration, and let L be produced by the automaton (A, S, S'). So A is a ground term rewriting system.

The idea is to look for possible R-redexes inside A. For a rule $l \to_R r \in R$, we look for substitutions σ with $l\sigma \twoheadrightarrow_A q$. That means that the state q of the automaton produces some redexes. We now apply the rewrite rule to the redex, and add to A a state transition from its contractum to q: $r\sigma \to_A q$. So both the redex and the contractum can now be produced from the state q.

In fact $r\sigma$ need not be monadic, so the rule cannot be directly added because it might be of an extended form. This has to be resolved as described in proposition 4.3.2. In doing so, some new states have to be introduced. Therefore, the process isn't guaranteed to terminate, as new states might incur new redex positions. So one needs a clever algorithm that tries to re-use already existing states of the automaton for the resolution of the $r\sigma \to q$ rules.

We call A *saturated* w.r.t. R if for all $l \to_R r \in R$, for all $q \in S$, for all $l\sigma \twoheadrightarrow_A q$ we also have $r\sigma \twoheadrightarrow_A q$. It is clear that a saturated automaton that originally contained L gives a rational approximation of $\Delta^{\twoheadrightarrow}(L)$ from above.

For the **CL(S)** case, fortunately, for some L saturated automata can be obtained. In fact we guess languages L that are already saturated, proving $L = \Delta^{\twoheadrightarrow}(L)$ or $L = \Delta^{\twoheadleftarrow}(L)$ (by reversing arrows in R).

Obviously, saturation of an automaton is a decidable property, and `RX` implements a test for saturation.