Languages and Compilers.

Continuation of the course in Programming Languages.

- New computation paradigms:
  - Logic: Prolog
  - Functional logic: Curry
- Formal description of declarative languages behavior:
  - functional programming: rewriting systems, lambda calculus, pattern matching
  - logic programming: unification, SLD-derivations
  - functional logic programming : narrowing
- Compilers
  - parsing
  - attribute grammars
  - type checking
  - intermediate code generation

24 hours on first semester + 48 hours on the second.

Languages and Compilers

Course web page ⇒ Marco Comini;
These slides available on my web page.

Exam:

- Similar to the exam for Programming Languages:
- three small projects, one at the end of first semester, during the January-February exam period, (one month time window);
- group projects (1-3 persons),
- first project: $\sim$ 2 days for a group of 3 persons.

Rewriting systems

Term rewriting systems

are the founding theory for:

- functional programming,
- logic programming,
- functional logic programming,
- concurrent systems.

Text book: TeReSe-Lite

Freely available on line.

The course only condisiders the material contained in the following sections:

- 0.0
- 1.0, 1.1, 1.2
- 2.0, 2.1, 2.2, 2.3, 2.7.0, 2.7.1
- 3.1, 3.2.0, 3.3.2,
- 4.1, 4.8.0

Here, *n*.0 indicates the first part of Chapter *n*, before Section *n*.1 starts. When subsections are indicated, as in 2.7.0, it is sufficient to read the material contained in the specified subsections, omitting the other parts of the section.

Rewriting as computation

In evaluating an expression, e.g. $(3 + 5 + 6) \cdot (1 + 2)$

- One moves from one expression to the following one.
- An finite set of rules are applied.
- Local rewriting.
- Non-determinism – confluence.
- Result as termination.

Several form of computation

can be formulated as rewriting systems:

- Turing machine;
- Post canonical systems;
- lambda calculus, functional programming;
- logic programming;
- ...

indeed

- computation proceeds by steps,
- each step of computation is local (modify just a local of the system).

Abstract reduction system

one abstracts on the nature of rewritten (reduced) objects.

One just considers the graph of reductions.

Definition (Abstract reduction system)

$$\mathcal{A} := (A, \to)$$

- A set
- $\to$ binary relation on A

- the set A represents the possible states of the computation;
- the relation $\to$ represents the single steps of reduction (rewriting, computation).

Abstract reduction system

The shape of the computation graph give information on the nature of computation.

- deterministic programs (non-deterministic);
- reversible computing;
- termination, non-termination,
- confluence.

Labeled reductions

In some areas (e.g, concurrence), one considers a set of reduction relations.

$$\mathcal{A} := (A, \{\rightarrow_\alpha \mid \alpha \in I\})$$

These structures are also called labeled transition systems (LTS)

Motivations:

- to distinguish among several kind of reduction rules,
- the reduction generates side-effects that need to be specified.

ARS, definitions:

When $a \rightarrow_\alpha b$ one says that:

- a reduces to b in a single step of (α-)reduction (computation),
- b is a (α-)reduct, in one step, of a ,
- a is an (α-)expansion, in one step, of b.

Definition (Reduction sequence)

is a, finite or infinite, sequence is the form:

$$a_0 \rightarrow a_1 \rightarrow \ldots \rightarrow a_n \rightarrow$$

One says that;

- a_0 reduces to a_n (in several steps);
- a_n is a reduct (by several steps) of a_0.

ARS, derived relations

One can look at an ARS as a directed graphs (possibly labeled).

$\rightarrow^+$ is the transitive closure of $\rightarrow$
$a \rightarrow^* b$ iff a reduces to b in n steps with $n > 0$
as a graph: there exists a non-empty path from a to b,

$\rightarrow^*$ is the reflexive and transitive closure of $\rightarrow$
for every a, $a \rightarrow^* a$.

$a = b$ is the symmetric, reflexive and transitive closure of $\rightarrow$
implicitly: if $a \rightarrow b$ then a and b are taken as equal,
as a graph: a and b belong to the same connected component.

Notation: The syntactic equality is denoted by $\equiv$.

Property on ARS: Confluence

Different reductions, starting from the same element, can re-converge to a common element.

Definition

An element $a \in A$ is called weakly confluent or weakly Church-Rosser (WCR) if
$\forall b, c.\ b \leftarrow a \rightarrow c$ we have that: $\exists d.\ b \rightarrow^* d\ {}^*\!\!\leftarrow c$.

An element $a \in A$ is called confluent or Church-Rosser (CR) if
$\forall b, c.\ b\ {}^*\!\!\leftarrow a \rightarrow^* c$ we have that: $\exists d.\ b \rightarrow^* d\ {}^*\!\!\leftarrow c$.

An ARS is (weakly) Church-Rosser if every element is (weakly) Church-Rosser.

CR ensures that different computations can re-converge on a common state.

WCR is easier to verify.

WCR is a strictly weaker property that CR. Counterexample: . . .

CR reduction system: examples an counterexamples

Examples:

- evaluation of arithmetic expressions ;
- pure functional languages;
- deterministic computation.

Counterexamples:

- definitions by pattern matching,
- logic programming,
- concurrent systems.

Several notion of confluence

There exists a plethora of different notions of confluence, in common they have the structure of the definitions, they differ for the used relations.

Definition

- Diamond property: $\forall b, c.\ b \leftarrow a \rightarrow c$ implies $\exists d.\ b \rightarrow d \leftarrow c$,
- $\rightarrow_\alpha$ weakly commutes with $\rightarrow_\beta$: $\forall b, c.\ b \leftarrow_\alpha a \rightarrow_\beta c$ implies $\exists d.$ $b \rightarrow^*_\beta d \leftarrow^*_\alpha c$,
- . . .

The most meaningful notions remain CR and WCR.

ARS properties: termination

Assumption: the final results of computation are the elements that cannot be further reduced.

Definition

- a is in normal form if $\nexists\, b.a \rightarrow b$.
- a is weakly normalizing (WN) if $\exists\, b$ in normal form s.t . $a \rightarrow^* b$.
- a is strong normalizing (SN) if there is no infinite reduction starting from a.
- an ARS is WN (SN) if every element in it is WN (SN).

ARS properties: confluence on normal forms

Definition

- an ARS has the normal form property (NF) if $\forall a, b\ .\ \ b$ normal form $\wedge\ a = b \ \Rightarrow\ a \rightarrow^* b$
- an ARS has the unique normal form property (UN) it $\forall a, b\ .\ \ a, b,$ normal forms $\wedge\ a = b \ \Rightarrow\ a \equiv b$

Theorem

$NF \Rightarrow UN$

Show that the reverse implication does not hold.

Logical implication among properties

Theorem

- $SN \wedge WCR \Rightarrow CR$
- $WN \wedge UN \Rightarrow CR$
- $CR \Rightarrow NF$

Term rewriting system (TRS)

ARS defines an abstract notion.
TRS special cases of ARS where:

- the elements (computation objects) are first order terms on a given algebra;
- reduction given by a set of rules:
  - defined parametrically (using variables and pattern matching)
  - rules applicable everywhere (in any context).

The relation $\rightarrow$ is called rewriting (instead of reduction).

Motivating example

Natural numbers arithmetic

Terms generated signature:
$\langle 0 : 0, S : 1, add, mult : 2\rangle$.

Rewriting rules:

- $add(x, 0) \rightarrow x$
- $add(x, S(y)) \rightarrow S(add(x, y))$
- $mult(x, 0) \rightarrow 0$
- $mult(x, S(y)) \rightarrow add(mult(x, y), x)$

Problems:

- What are the (closed) terms in normal form.
- Reduce $mult(S(S(0)), S(S(0)))$.
- Show that $mult(S(S(0)), S(S(0)))$ è CR.
- Define the rewriting rules for subtraction.

Terms defined by a syntax

Signature Σ: list of constants and functions organized by arity.
Example: $\langle 0 : 0, S : 1, add, mult : 2\rangle$.
Constants can be seen as functions with 0 arguments.
The syntax may contain also an extra set of variables V.

Definition

The set of terms on Σ, V is the set $T(\Sigma, V)$ generated by the following rules:

- $\forall x \in V \qquad x \in T(\Sigma, V)$
- $\forall f \in \Sigma, \quad arity(f) = n, \quad t_1, \dots t_n \in T(\Sigma, V)$
  $f(t_1, \dots t_n) \in T(\Sigma, V)$

According to definition a constant c should be written as $c()$, but we use the usual writing.
Only prefix notation, no infix notation.
Exercise. Define the syntax for Boolean algebra.

Alternative definition

The set of terms generated by the grammar:

$$\begin{aligned} T := \; & x_1 \mid x_2 \mid \ldots \\ & c_1() \mid c_2() \mid \ldots \\ & u_1(T) \mid u_2(T) \mid \ldots \\ & f_1(T,T) \mid f_2(T,T) \mid \ldots \\ & g_1(T,T,T) \mid g_2(T,T,T) \mid \ldots \end{aligned}$$

Counting

Definition

- Ground terms: closed, not containing variables, belonging to $T(\Sigma, \emptyset)$
- Linear term: each variable occurs at most once.
- $Var(t)$ denotes the set of variable occurring in t:
  t ground $\Leftrightarrow$ $Var(t) = \emptyset$.
- The length of a term t, (denoted by $|t|$) is the number of functions and variable symbols appearing in t:
  - $|x| = 1$
  - $|f(t_1, \ldots, t_n)| = 1 + |t_1| + \ldots + |t_n|$

  The depth of a term t in the maximal number of consecutive application:
  - $depth(x) = 1$
  - $depth(f(t_1, \ldots, t_n)) = 1 + \max(depth(t_1), \ldots, depth(t_n))$

Context

Rewriting rules can be applied to subterms,
that is terms inside a context.
Context: what is remain of a terms having removed a subterm.
Formally:

Definition

A context Σ is a term generated by a signature $\Sigma : \langle [\,] : 0 \rangle$,

Symbol $[\,]$, called hole, represent a missing subterm.
We just consider in *one hole* context.
Notation. $C[\,]$.
Examples: $f(c_0, [\,], g(c_1, c_2))$, $(3 + 8) \times ([\,] + 5)$

Basic operation: substitute the hole in the context $C[\,]$ by a term t; $C[t]$.

Term tree

It is useful to thought to terms as ordered trees.

- variables an constants are the leaves.
- functions are internal nodes with having as children the arguments.

Positions

In a term tree every subterm is uniquely identify by the path root-subterm. Such a path is describe the sequence of natural numbers, telling for each node which subtree to choose.
Notation

- $\langle n_1, \dots, n_i \rangle$ sequence;
- $\langle n_1, \dots, n_i \rangle \cdot \langle m_1, \dots, m_j \rangle$ concatenation of sequences;
- $t\,|_{\langle n_0, \dots, n_i \rangle}$ subterm of t identify by $\langle n_0, \dots, n_i \rangle$;
- $t[\;]_{\langle n_0, \dots, n_i \rangle}$ context obtained by removing from t the subterm $t\,|_{\langle n_0, \dots, n_i \rangle}$

Examples:

Substitution

The operation of substituting, in a terms, some variables with other terms.

Definition

- A substitution is a function $\sigma : V \to T(\Sigma, V)$.
- A substitution can be extended to the whole set of terms, $\sigma : T(\Sigma, V) \to T(\Sigma, V)$, by syntax induction: $\sigma(f(t_1, \dots, t_n)) = f(\sigma(t_1), \dots, \sigma(t_n))$.

$\sigma(t)$ is usually written as $t\sigma$ or as t^{σ}.

Domain and codomain

Definition

The domain of σ is the set of variable modify by σ

$$Dom(\sigma) = \{x \in V \mid \sigma(x) \neq x\}$$

The codomain is the set of variables contained in the image of the domain.

$$Cod(\sigma) = \bigcup_{x \in Dom(\sigma)} Var(\sigma(x))$$

Note that this definition is different from the standard definition of domain and codomain of a functions.

Finite domain substitutions

We are mainly interested in substitutions having a finite domain, they can be represented by finite set (assignments) pairs:

$$\{x_1/s_1, \dots, x_n/s_n\}$$

The set contains the variable in the domain of the substitution.

Possible alternative notations, in the literature: $\{x_1 \mapsto s_1, \dots, x_n \mapsto s_n\}$ or $\{s_1/x_1, \dots, s_n/x_n\}$

Substitution restriction:

$$\sigma|_V = \begin{cases} \sigma(x) & \text{if } x \in V \\ x & \text{otherwise} \end{cases}$$

Unions of substitution with disjoint domain:

$$\sigma_1 \cup \sigma_2$$

Substitutions composition

As functions on terms, substitutions compose:

$$(\sigma \circ \tau)(t) = \sigma(\tau(t))$$

We use the notation:

$$t^{\tau\sigma} = (t^{\tau})^{\sigma}$$

On variables:

$$\tau\sigma(x) = (\tau(x))^{\sigma}$$

Composition is associative (as every function composition).
Is it commutative?

Substitutions composition

As functions on terms, substitutions compose:

$$(\sigma \circ \tau)(t) = \sigma(\tau(t))$$

We use the notation:

$$t^{\tau\sigma} = (t^{\tau})^{\sigma}$$

On variables:

$$\tau\sigma(x) = (\tau(x))^{\sigma}$$

Composition is associative (as every function composition).
Is it commutative? Find a counterexample.

Composition of finite domain substitutions

Given two substitutions with finite domain:

$$\sigma = \{x_1/s_1, \dots, x_m/s_m\}$$
$$\tau = \{y_1/t_1, \dots, y_n/t_n\}$$

from the formula of $\sigma\tau$ on variable, one obtains that $\sigma\tau$ is given by the set:

$$\{x_1/s_1^{\tau}, \dots, x_m/s_m^{\tau},\ y_1/t_1, \dots, y_n/t_n\}$$

from the above set one needs to remove the pairs y_i/t_i with
$y_i \in \{x_1, \dots, x_m\} = Dom(\sigma)$ e
(and for compactness) the pairs x_i/x_i, (generated by those x_i such that
$x_i = x_i^{\sigma\tau}$.

Matching

Problem

Given s and t build, if it exists, built a substitution σ such that $s \equiv t^{\sigma}$.

Example

- C(x, C (y, z)) / C(1, C(2,T))
- N(x, N(y,y,w), z) / N(S, N (T, T, z), y)
- N(x, N(y,w,y), z) / N(S, N (T, T, z), T)

In Haskell syntax:

- (C x (C y z)) / (C 1 (C 2 T))
- (N x (N y y w) z) / (N S (N T T z) y)
- (N x (N y w y) z) / (N S (N T T z) T)

Matching

To present the matching algorithm, it is convenient to generalize the problem to the matching of two sequences of terms, by a single substitution.

$$\{t_1/s_1, \ldots, t_n/s_n\}$$

Matching Algorithm

Givem by a set of rewriting rules:

$$\begin{array}{rcl} \{f(t_1,\ldots,t_n)/f(s_1,\ldots,s_n)\} \cup S & \Longrightarrow & \{t_1/s_1,\ldots,t_n/s_n\} \cup S \\ \{f(t_1,\ldots,t_n)/g(s_1,\ldots,s_m)\} \cup S & \Longrightarrow & \textbf{fail}, \text{ if } f \not\equiv g \\ \{f(t_1,\ldots,t_n)/x\} \cup S & \Longrightarrow & \textbf{fail} \\ \{x/s_1\} \cup S & \Longrightarrow & \textbf{fail} \text{ if } x/s_2 \in S \ \wedge \ s_1 \not\equiv s_2 \end{array}$$

When no rule can be applied, we have the matching substitution.

Unification

Problem

Given s and t build, if it exists, built a substitution σ such that $s^\sigma \equiv t^\sigma$.

Example

N(x, T, z) = N(T, y, y)
N(x, T, z) = N(T, y, w)

- More than one substitution solves the problem.
  Instead, matching has a unique solution.
- There exists a *canonical* solution: the most general substitution.
- We need to define a generality order on terms, and on substitutions.

Renaming

Definition

A renaming is a substitution σ:

- replacing variable by variable,
- and not identifying two different variables.

Prove that: a finite substiturion σ is a renaming iff is a permutations of variables.

Definition

A substitution σ is said to be a renaming for a term t is σ

- substitutes each variable in t by a variable and
- does not identify distinct variables in t).

TeReSe givem the following definition:
σ is renaming for a term t if $\sigma|_{Var(t)}$ is a renaming.

Why the two definitions do not coincide?

Subsumption order on terms

Definition

If $s \equiv t^\sigma$:

- s is an instance of t
- t subsumes s, t is more general than s.
- t is matched with its instance s by the substitution σ.
- $s \preceq t$

The relation $\preceq$ is a quasi-order
i.e. a reflexive and transitive relation,
not necessarily antisymmetric. Proof?

If $s \preceq t$ and $t \preceq s$, t is called a variant of s, $s \cong t$

$s \cong t$ iff there exists a renaming σ such that $s \equiv t^\sigma$

We write $s \prec t$ if $s \preceq t$ and $s \ncong t$, (s is a proper instance of t).

Property of subsumption quasi-order

Theorem

- *$<$ does not have any strictly increasing chain, (chain of more and more general terms)*
- *any two terms s and t have a supremum (sup) r, that is $s \preceq r$, $t \preceq r$ e $\forall q.(s \preceq q) \wedge (t \preceq q) \;\Rightarrow\; r \preceq q$,*

Corollary

Two terms s and t do not necessary have a common instance, but if they have a common instance they also have most general comon instance (mgci).

Subsumption order on substitutions

Definition

The substitution τ is more general that σ:

$$\sigma \preceq \tau$$

if

$$\exists \rho.\, \sigma = \tau\rho$$

Property:
$\sigma \preceq \tau \quad \Rightarrow \forall t.\ t\sigma \preceq t\tau.$

Does the other implication hold?
Almost always.
(Very difficult) Exercise: find a counterexample.

Unification

Given two terms s and t, can they be equated by a common substitution?

Definizione

[Unifier, MGU]

- If $s\sigma \equiv t\sigma$ then σ is a unifier of s and t.
- The greater unifier of two terms s and t, with respect to the subsumption order, $\preceq$, is called the most general unifier MGU.

Theorem

If two terms s, t admit a unifier they also have a MGU (unique up to renaming).

Unification algorithm (Martelli, Montanari, 1982)

Given by a set of rewriting rules:

$$\begin{array}{rcl}
\{f(t_1,\ldots,t_n) = f(s_1,\ldots,s_n)\} \cup E & \Longrightarrow & \{t_1 = s_1,\ldots,t_n = s_n\} \cup E \\
\{f(t_1,\ldots,t_n) = g(s_1,\ldots,s_m)\} \cup E & \Longrightarrow & \textbf{fail}\text{, se } f \not\equiv g \\
\{x = x\} \cup E & \Longrightarrow & E \\
\{f(t_1,\ldots,t_n) = x\} \cup E & \Longrightarrow & \{x = f(t_1,\ldots,t_n)\} \cup E \\
\{x = t\} \cup E & \Longrightarrow & \textbf{fail} \text{ se } x \in Var(t) \text{ e } x \neq t \\
\{x = t\} \cup E & \Longrightarrow & \{x = t\} \cup E\{x/t\} \\
 & & \text{se } x \notin Var(t), x \in Var(E)
\end{array}$$

When no rules can be applied, the equation gives the MGU.

Observations

- Martelli-Montanari algorithm is an efficient version of Robinson unification algorithm (65).
- The algorithm is non deterministic, several rule can be applied to the same set.
- Correctness: each single rules is correct, does not change the set of unifiers.
- Termination not completely obvious, the last rules can increase the length of the terms, a well-founded order is needed.

Reduction rules on Σ-terms

Definition (Reduction Rule)

A reduction (rewriting) rule (on Σ) is given by a pair of terms $l \to r$ in $T(\Sigma, V)$ such that:

1) l is not variable;
2) $Var(r) \subseteq Var(l)$

$l \to r$ represents a rule scheme: the whole rules that can be obtained by instantiating variables and inserting in contexts.

$$\{C[l^\sigma] \to C[r^\sigma] \mid \text{for any substitution} \sigma \text{and context } C[\,]\}$$

- l^σ redex
- r^σ contractum

Term Rewriting System (TRS)

Definition (Term Rewriting System)

- A TRS is a pair $\mathcal{R} := \langle \Sigma, R \rangle$ formed by a signature Σ and a set of rules R on Σ.
- A TRS induced a reduction relation, $\to_R$, on $T(\Sigma, V)$, the union of l'unione delle riduzioni definite dalle singole regole.

- If there is no ambiguity, one can remove the subscript in $\to$.
- A TRS is an instance of an Abstract Reduction System.
- All definition given on ARS can be applied to TRS. That is: reduction sequence, $\to^*$, $=$, WCR, CR, normal form, WN, SN, NF, UN.
- If reduction rules are named: $\rho : l \to r$, one can write $t \to_\rho s$ to indicate that t reduces to s by an application of the rule ρ.

Examples — Algebraic data types

- Arithmetic: $\langle 0 : 0, S : 1, add, mult : 2\rangle$
  - $add(x, 0) \to x$
  - $add(x, S(y)) \to S(add(x, y))$
  - $mult(x, 0) \to 0$
  - $mult(x, S(y)) \to add(mult(x, y), x)$
- Lists and append, merge functions.
- Binary trees and linearize function.
- Binary numbers and addition

Examples – Combinatory Logic (Moses Schönfinkel, Haskell Curry)

Combinatory Logic signature: $\langle S, K : 0, app : 2\rangle$
Usually *app* is written with the infix notation:
$app(x, y)$ is written as $x \cdot y$

- $(K \cdot x) \cdot y \ \rightarrow \ x$
- $((S \cdot x) \cdot y) \cdot z \ \rightarrow \ (x \cdot z) \cdot (y \cdot z)$.

Reduce: $((S \cdot K) \cdot K) \cdot x$.

As for product, the symbol $\cdot$ can be omitted.
Application is left-associative. Es. $SKK = ((SK)K)$.

Defined I as SKK, reduce $SII(SII)$.

Defined B as $S(KS)$, reduce $Bxyz$.

In combinatory one can encode natural numbers and define computable function.

Condition on rules

Some properties on rules implies properties of the TRS.

A rule $\rho : l \rightarrow r$ è:

- left-linear if l is linear.
  The cost of verifying if a rule is applicable to a given subterms, is constant.
- non-duplicating if no variable has more occurrences r than l.
  Duplicating otherwise: the rule duplicate part of the redex.
- non-erasing is $Var(l) = Var(r)$.
  Erasing otherwise: in the redex all the subterms that match with a given variable are erased.
- non-colapsing if r is not a variable. Collapsing otherwise.

A TRS is called left-linear (non-duplicating, non-erasing, non-collapsing if all its rules are so.

Exercise: apply these definition to the TRS for arithmetic.

Conditions on rules for confluence, (CR)

Example of a non-confluent TRS

$$\begin{array}{lll} \rho_1 : & f(g(x), y) & \rightarrow \ x \\ \rho_2 : & g(a) & \rightarrow \ b \end{array}$$

The term $f(g(a), b)$ is not confluent

The rules ρ_1 e ρ_2 can both be applied but the application of one rule destroy the possibility of applying the other.

An analysis of this possibility leads to the following definition.

Overlapping rules

Definition

Two rules $\rho_0 : l_0 \rightarrow r_0$ and $\rho_1 : l_1 \rightarrow r_1$ are overlapping if l_0 and a subterm, different from a variable, of l_1 have a common instance, or the other way round.

Examples of overlapping rules

Overlapping can involve the whole left sides of both rules l_0, l_1, consider the rules:

$$\begin{array}{lll} \rho_1 : & f(g(x), y) & \rightarrow & x \\ \rho_2 : & f(x, b) & \rightarrow & b \end{array}$$

and the term:

$$f(g(a), b)$$

One rule can overlap with itself, consider the rule:

$$\rho : \quad f(f(x)) \quad \rightarrow \quad a$$

and the term:

$$f(f(f(b)))$$

Equivalent condition

Two rules $\rho_0 : l_0 \rightarrow r_0$ e $\rho_1 : l_1 \rightarrow r_1$ are overlapping if, after renaming the variable in such a way that $Var(l_0) \cup Var(l_1) = \emptyset$, the term l_0 unify with a subterm, different from a variable, of l_1.

Example of non-overlapping rules

Non overlapping rules can generate redex, one inside the other, but in this case the application of one redex tdoes not destroy the other:

Compare:

$$\begin{array}{lll} \rho_1 : & f(x, y) & \rightarrow & x \\ \rho_2 : & g(a) & \rightarrow & b \end{array}$$

with

$$\begin{array}{lll} \rho_1 : & f(g(x), y) & \rightarrow & x \\ \rho_2 : & g(a) & \rightarrow & b \end{array}$$

on the term:

$$f(g(a), g(a))$$

Result

Theorem

A TRS without overlapping rules is weakly confluent, WCR (weak Church-Rosser).

There exists a stronger version of this theorem admitting “good” overlapping rules.

A WCR is not necessary confluent CR, if the system is not strong normalizing
($WCR \wedge SN \;\Rightarrow\; CR$).

Counterexample

Consider the TRS given by the rules:

$$\begin{array}{llll} \rho_1: & f(x,x) & \rightarrow & b \\ \rho_2: & g(x) & \rightarrow & f(x,g(x)) \\ \rho_3: & a & \rightarrow & g(a) \end{array}$$

Observe that:

- the rules are not overlapping
- the TRS is WCR
- $g(a) \rightarrow^* b$ and $g(a) \rightarrow^* g(b)$,
- b and $g(b)$ do not reduce to a comon term.

Orthogonality

Definition

A TRS $\mathcal{R}$ is orthogonal if it is left-linear and does not contain overlapping rules.

Theorem

Every orthogonal TRS is confluent (CR).

Constructor Based TRS

Quite often on defining a signature Σ, one assume a partition on the function symbols in Σ,

- $\mathcal{C}$, constructors, they generate the elements in a given type
- $\mathcal{D}$, defined functions, the functions defined on given type

Examples:

- on the signature for arithmetic, constructors are $0, S$, defined functions are add, mul
- on the signature for lists, constructors are $nil, cons$, defined functions are $append, concatenate$

Definition

A Constructor Based TRS, is a TRS containing only rules in the form:

$$f(t_1, \ldots, t_n) \rightarrow s \text{ with } f \in \mathcal{D} \wedge \forall i.\, t_i \in T(\mathcal{C}, V)$$

Haskell

Haskell, when functions are defined by pattern matching is a Constructor Based TRS.

Lemma

In a CBTRS, two rules, $\rho_0 : l_0 \rightarrow r_0$ and $\rho_1 : l_1 \rightarrow r_1$ are overlapping iff l_0 and l_1 have a common instance.

With some modification, the orthogonality theorem can be applied to Haskell, in fact:

- pattern mathing rules, in Haskell are left linear,
- there can be overlapping rules, but in this case,
  they are part of the definition of the same function,
  since the rule are apply in the written order,
  the rewriting system is confluent.

Reduction strutegies

In a TRS $\mathcal{R}$, in order to apply a reduction $t \to t'$ on a term t, one needs:

- to select a position p on t,
- and a rewriting rule $\rho : l \to r$ in $\mathcal{R}$,
- such that there exists $\sigma = \text{match}\{l/t|_p\}$ ($l^\sigma = t|_p$),
- in this case, $t' = t[r^\sigma]_p$ and the subterm $t|_p$ is the redex.

Given a term t, there can be many possible reductions $t \to t'$.

A reduction strategy for any term t select on possible reduction on t.

Reduction strategies

Definition (Extentional)

A one step reduction strategy $\mathbb{F}$ on a TRS $\langle \Sigma, R \rangle$ is a function $\mathbb{F} : T(\Sigma, V) \to T(\Sigma, V) \cup \{*\}$ such that:

- $\mathbb{F}(t) \equiv *$ if $t \not\to$
- $t \to \mathbb{F}(t)$ otherwise.

A multy step reduction strategy strategia require that $t \to^+ \mathbb{F}(t)$.

A more intensional definition, associate to any term t, and the position p and rule ρ defining a rewriting on t.

In an orthogonal TRS, for any term and any position there exists at most one applicable rule. One can define strategies, just by indicating where the rules have to be applied.

Strategie di riduzione classiche

leftmost-innermost (ad un passo) scelgo il redex più a sinistra tra quelli più interni.
In termini di posizioni: scelgo il redex con posizione p minima nell'ordine lessicografico tra i redex aventi una posizione p' massimale in lunghezza.

parallel-innermost (a molti passi) riduco tutti i redex più interni (con posizioni massimali in lunghezza). I redex sono disgiunti.

leftmost-outtermost (a un passo) scelgo il redex più a sinistra tra quelli più esterni.

parallel-outtermost (a molti passi) riduco tutti i redex più esterni (con posizioni minimali in lunghezza). I redex sono disgiunti.

full-substitution (a molti passi) riduco tutti i redex più ovunque. I redex non sono disgiunti ma in un TRS ortogonali una riduzione non disturba l'applicabilità di un altra.

Normalization

Definition

Normalizing strategy A reduction strategy is $\mathbb{F}$ is normalizing if leads to the normal form when it exists:
for every term t having a normal form, there exists n such that $\mathbb{F}^n(t) = *$.

Proposition

In orthogonal TRS, satisfying an extra condition: left-normality, the leftmost-outtermost strategy is normalizing.

The need for left-normality , is explained by the following counterexample:

consider the term $f(c, a)$, and the reduction rules:

- $f(x, b) \to d$
- $a \to b$
- $c \to c$

Normalization

Proposition

In an orthogonal TRS the parallel-outtermost is normalizing.

Outtermost, innermost

- Innermost: strategia di riduzione eager, si valutano gli argomenti prima di passarli alla funzione, ML.
- Outtermost: strategie di riduzione lazy, valuto un argomento solo se espressamente richiesto, Haskell .

Strategie eager non sono normalizzanti: valuto un argomento che diverge anche se questo non viene usato nella funzione.

Strategie lazy, con regole di riduzioni duplicanti, portano a valutare un argomento più volte, a meno che non rappresenti i termini con DAG (Direct Acyclic Graph), come in Haskell.

Haskell and TRS

In Haskell one can define constructor based TRS:

- in type definition one defines a set of constructors for each type;
- functions defined by pattern matching induce the reduction rules;
- lazyness and the order of pattern matching rules defines the reduction strategies;
- TRS theorem shows that Haskell computation, for this CBTRS is confluent and, in some cases, normalizing.

Can one describe every Haskell computation via a TRS?

Operational Semantics for Haskell

- Haskell programs are terms (p).
- A subset of terms defined the possible results of a computation (n).
- A set of rules defined a reductions relation $\rightarrow$ is defined on terms.
- Such that $p \rightarrow^* n$ iff the program p gives the result n.

TRS and Operational Semantics for Haskell

TRS are not sufficient for the operational semantics, several problems

- Terms are non typed, there is only one kind of terms,
  one cannot distinguish between booleans and integers, ...
  This problem can be solved by multi-sort TRS.
- Haskell is functional:
  can define and manipulated functions,
  this cannot be done in TRS, one needs higher-order TRS.

Lambda abstraction

In Haskell it is possible to defines name less fuctions
`\x -> x + 2`
`\f -> \x -> f (f x)`
In other formalism these expression are written:
$\lambda x\,.\,x + 2$, $\lambda f\,.\,\lambda x\,.\,f(fx)$

Functions are first class citizen.

One need a rule for application, the β-rule:

$$\texttt{(\textbackslash x -> f) m} \quad \rightarrow \quad \texttt{f \{x/m\}}$$

An example:

$$\texttt{(\textbackslash x -> x + x) n} \quad \rightarrow \quad \texttt{n + n}$$

Lambda-abstraction

Lambda-abstraction is binding operator:
`\x ->`
Transform `x` in a closed variable, the parameter of the functions.

Binding operators cannot be defined in TRS.

α-equality

The name of the parameter is not important:

`\x -> x + 2`

is perfectly equivalent to:

`\y -> y + 2`

One need to define a rule establishing this equivalence the α-rules:

$$\texttt{\textbackslash x -> t} = \texttt{\textbackslash y -> t \{x/y\}}$$

Variable (parameter) renaming

Necessary while applying the β-rule.

Without variable renaming
`(\f -> \x -> f x) (\y -> x + y)`
β reduces to
`\x -> (\y -> x + y) x`
`\x -> x + x`

Not correct, the free variable `x` is captured.

The correct reduction is:
```
(\f -> \x -> f x) (\y -> x + y)
(\f -> \z -> f z) (\y -> x + y)
\z -> (\y -> x + y) z
\z -> x + z
```

β-reduction

The previous example shows that β reduction cannot be defined naively, some α-conversion (variable renaming) can be necessary during the reduction,
this make more complex:

- the formal definition of rewriting system
- the implementation of functional programming languages

The pure lambda calculus (untyped)

A quite expressive rewriting system
where term are build using

- variables: `x`,
- application, a binary operator
  if `s` and `t` are terms then `(s t)` is a term,
- λ-abstraction
  if `x` is variable and `t` is term then `\x -> t` is a term,

having as only reduction rule the β-rules,
that can be applied in any context,
not definable as a rule of TRS.

Terms are not typed, one term can be applied to itself, like in
`(\x -> x x)(\x -> x x)`

Church numerals

One can encode natural numbers by:

- 0 ≡ `(\f x -> x)`
- 1 ≡ `(\f x -> f x)`
- 2 ≡ `(\f x -> f (f x))`
- $n \equiv$ `(\f x -> f`^{n}`x)`

The standard arithmetic function are defined by:

- Successor: `\m -> \f x -> f (m f x)`
- Addition: `\m n -> \f x -> m f ( n f x)`
- Product: `\m n -> \f x -> m ( n f) x`
- Exponentiation: `\m n -> \f x -> n m f x`

A difficult exercise: to define predecessor.

Recursion

In the λ-calculus one can define a fixed-point operator:
the Y combinator, discovered by Haskell B. Curry:

```
 Y  ≡\f -> (\x -> f (x x)) (\x -> f (x x))
```

Beta reduction of this gives,

```
 Y g = ( \x -> g ( x x ) ) ( \x -> g ( x x ) )
= g (( \x -> g ( x x ) ) ( \x -> g ( x x ) ))
= g ( Y g ) (by first equality)
```

By repeatedly applying this equality we get,

```
 Y g = g ( Y g ) = g ( g ( Y g ) )
 = g ( ...  g ( Y g ) ...  )
```

`Y` allows to define function recursively.

Confluency

There can be several possible reductions for λ-terms,

```
(\f -> f ( f a ) ) ((\x y -> g x y )b )
```

But we have

Teorema (Church-Rosser)

The reduction relation on λ-calculus is confluent (is Church-Rosser).

Untyped λ-calculus inside Haskell

One can define a data for the untyped λ-terms.

```
data Untyped = Abst [Untyped -> Untyped]
```

One need a translation:
untyped application and abstraction cannot translate to typed ones:

- `(t s)`
  becomes: `app t s`
  where: `app =  t -> case t of Abst f | f`
- `\x -> t`
  becomes: `Abst (\x -> t)`

λ-calculus with constants

Pure untyped λ-calculus is unsuitable for real programming.

One needs to introduce a set of constants:

- to build elements of data types,
- to define the basic functions on data types.

Typed language are safer.

A typed λ-calculus with constants is the:
Core Language di Haskell.

Core Language

Core languages are useful in defining the behavior, or the implementation, of a programming languages.

- A core language is a simple language, containing a subset of the features of a main programming language;
- the main language can be (easily) translated in the core language.
- Core language allow a modular approach to semantics and implemtetation; in fact is sufficient:
    - to give semantics (implement) the core language,
    - define the translation of mail language in the core language.

Core languages emphasize:

- fundamental mechanisms of computation,
- similarities and differences between languages.

Core Language for Haskell

Tiny, a typed lambda calculus:

- application and λ-abstraction,
- recursive types,
- a set of basic constant.

Recursive data type: constructor and destructor

In Haskell constants are defined via data type declarations.
Each data type declaration introduces a given number of constructors and a single destructor.

```
data PNat = Z | S PNat
```

The above definition generates two constructors:

```
Z :: PNat
S :: PNat -> PNat
```

and a destructor:

```
case_PNat ::  a -> ( PNat -> a) -> PNat -> a
```

Destructor

The destructor `case_PNat` has the following reduction rules:

```
case_PNat a f Z → a
case_PNat a f (S n) → f n
```

The Haskell expression:

```
case m of Z -> a |
        S n -> f n
```

is a syntactic sugar for (stands for):

```
case_PNat a f m
```

Example

The patter matching definition of addition:

```
add 0 y = 0
add (S x) y = S (add x y)
```

is equivalent to the definition:

```
add = \ x y -> case x of 0 -> y |
                           S x1 -> S (add x1 y)
```

that, in the core language, becomes:

```
add = Y (\ f x y -> case_PNat y (\ x1 -> f x1 y) x)
```

Core language

- The only computation mechanism is function application.
- Definition by pattern matching are reduced to definition by `case` that can be reduced to application of data destructor.

Reduction strategies

One core language the reduction relation is defined by a set of rules in natural deduction style,
the rules implicitly define the reduction strategy, for example:

Eager evaluation:

$$\frac{t_1 \to^* \lambda x.t_1' \quad t_2 \to^* v_2 \quad t_1'[v_2/x] \to^* c}{(t_1 t_2) \ \to^* c}$$

Lazy evaluation:

$$\frac{t_1 \to^* \lambda x.t_1' \quad t_1'[t_2/x] \to^* c}{(t_1 t_2) \ \to^* c}$$