Some ideas for the introduction of reasoning techniques in Welder

July-August 2017

Rodrigo Raya Castellano

Contents

1 Contributions to Welder

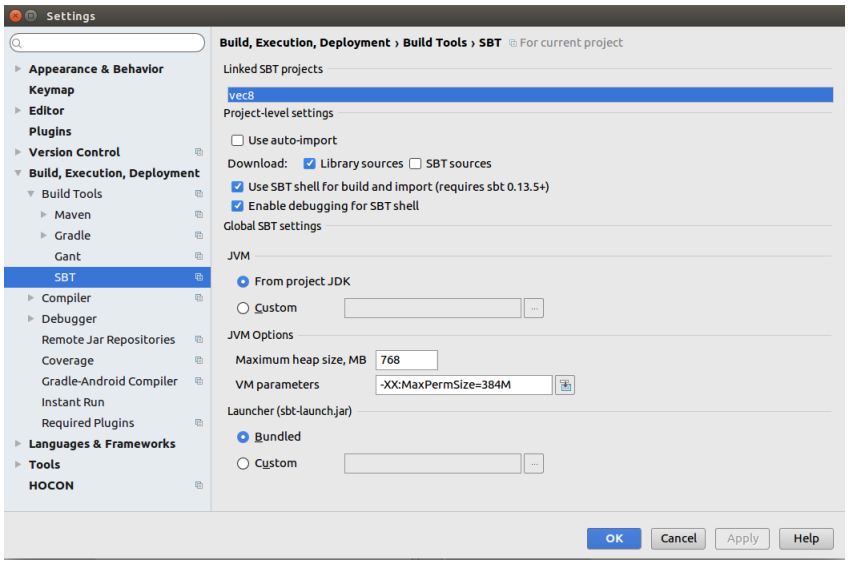
1.1 Setup to use Welder in Intellij Idea

One of the main difficulties to prove theorems in Welder when we first approached the assistant was the fact that errors where really difficult to deal with. Errors could come not only from Welder, but from Inox (we actually uncover some bugs in the least used parts of the system) or just because what we wanted to prove was not true.

We believe that having a proper way to debug proofs could not only reduce the learning curve of new users but also facilitate that new users acquire a better understanding of the SMT solver → Inox → Welder stack. The following explains a possible setup for Welder to run on Intellij Idea IDE.

On Intellij Idea 2017.2.1 we create a new Scala SBT-based project. In the file build.sbt we write the following:

Next, we go to File → Settings → Build,Execution,Deployment → Build Tools → SBT and mark the option *Use SBT shell for build and import (requires sbt 0.13.5+)* and press OK.



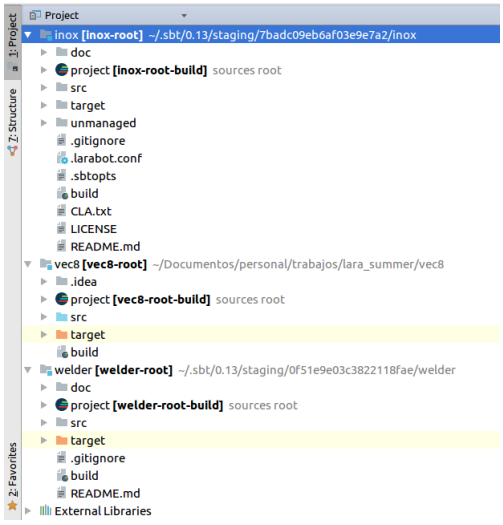
Finally, we select the option Refresh project that should have appeared after updating the build.sbt file. We accept, without unmarking the removal of the modules that Intellij Idea suggests.

Once we are done, we obtain in the Project Panel three projects modules. Observe that the RootProject command in the build.sbt is highlighted. We can get rid of this by renaming the projects and build

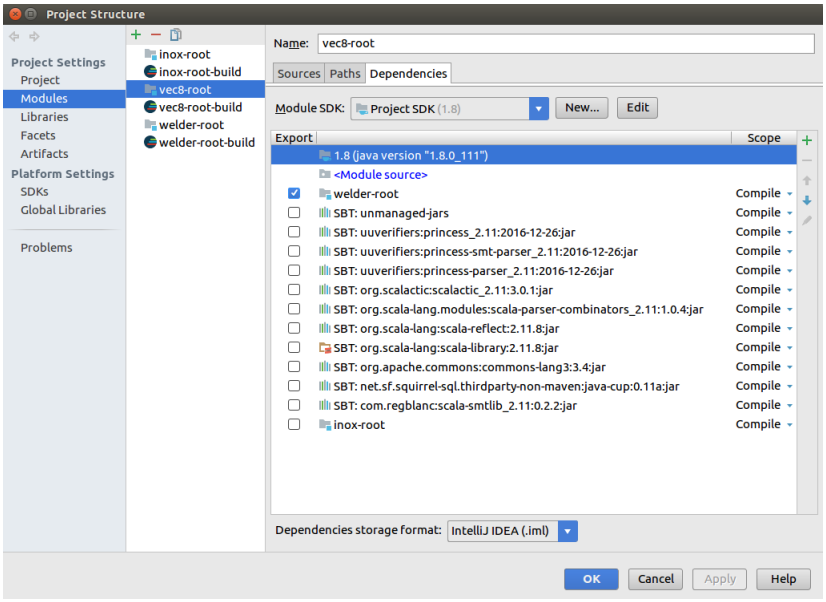
files to the following:

1. inox-root and inox-root-build
2. myproject-root and myproject-root-build
3. welder-root and welder-root-build

To do so right click and the modules and then Refactor → Rename → Rename Module. Before building our project, we mark the src file in our project as source file in File → Project-Structure → Project name → Sources → src and selecting Sources icon. Then we put our Main.scala file in the corresponding directory.



We also need to tell Intellij Idea that our project depends on Inox. We do so in File → Project-Structure → Project name → + → Module dependency... → inox-root.



We set the running configuration in Run → Edit configuration → Application mode and selecting the module and the main program as usual. Before running this, we go to the SBT shell and compile the program. Finally, we run the Application as usual in Intellij Idea.

The above configuration process can be found in references [1]-[7].

1.2 Adding the first "naive" tactics

During the course, we learnt that much of the effort in proving theorems happens while finding the appropiate inductive hypothesis. In fact, many direct goals are not solved by the underlying SMT solver.

On the other hand, most of the literature that we are aware of works at low level on the SMT solver such as in [8] or [9]. We carried out experiments to use some of the strategies described in these papers but the results did not change.

Therefore, we concluded that in order to provide a first example of tactic in Welder it should be as simple as a method that allows the user to perform automatic induction by providing automatically the inductive hypothesis on the sub-trees of the algebraic data types.

Here is an example of how this new syntax would look like:

```
lazy val lemma: Theorem = {
 def property(l: Expr) = {
  forall("i" :: IntegerType, "j" :: IntegerType){ case (i,j) =>
   (i >= 0 && j >= 0) ==> (drop(l,i+j) === drop(drop(l,j),i))
  }
 }
 induct(property _, "l" :: T(list))
}
```

Listing 1: Proposed induct syntax

The above example would succeed in proving $\forall l : List, i : Nat, j : Nat.drop(l, i+j) === drop(drop(l, j), i)$. In any case, we recommend to use the version of Welder used in the online reference of the project [10] for testing the examples.

Essentially the two summarized versions of the induction constructs we implemented are as follows:

```
def induct(property: Expr => Expr, valDef: ValDef): Attempt[Theorem]
def natInduct(property: Expr => Expr, base: Expr,
              baseCase: Goal => Attempt[Witness]): Attempt[Theorem]
```

Listing 2: Proposed induct signatures to be found in Tactics.scala

which perform structural and natural induction respectively.

It was not easy to find other easy examples of implementable tactics which motivates the second part of our work in which we tried to investigate Term Rewriting techniques in order to solve automatically theorems that we proved in the elliptic curves work carried out during the semester.

2 Term rewriting

2.1 Motivation

After the practical work that we carried out to improve the way in which the developer relates with Welder, it turned out that more interesting strategies were necessary to turn Welder into a real proof assistant. We found a possible solution in [11] and used [12] as further support.

As a motivating example, we consider the problem of determining if a given equation holds in group theory. This is not far from our elliptic curve project, since in several occasions we had to manually deduce easy properties of fields that only require basic reasoning. Given the axioms that define a group:

$$\begin{aligned}(G1)\ (xy)z &\approx x(yz)\\ (G2)\ ex &\approx x\\ (G3)\ x^{-1}x &\approx e\end{aligned}$$

where as usual e denotes the neutral element of the group and x^{-1} denotes the inverse. We can for instance try to decide if equation $e = xx^{-1}$ holds. This is the so-called word problem:

Definition 2.1 (The word problem informally)**.**
Given a set of identities E and two terms s, t, determine if it is possible to derive t from s using E as a set of rewriting equations in both directions.

Using $E = \{G_1, G_2, G_3\}$ we can derive term $t = xx^{-1}$ from term $s = e$:

$$e \overset{G3}{\approx} (xx^{-1})^{-1}(xx^{-1}) \overset{G2}{\approx} (xx^{-1})^{-1}(x(ex^{-1})) \overset{G3}{\approx} (xx^{-1})^{-1}(x((x^{-1}x)x^{-1})) \overset{G1}{\approx} (xx^{-1})^{-1}((x(x^{-1}x))x^{-1})$$

$$\overset{G1}{\approx} (xx^{-1})^{-1}(((xx^{-1})x)x^{-1}) \overset{G1}{\approx} (xx^{-1})^{-1}((xx^{-1})(xx^{-1})) \overset{G1}{\approx} ((xx^{-1})^{-1}(xx^{-1}))(xx^{-1}) \overset{G3}{\approx} e(xx^{-1}) \overset{G2}{\approx} xx^{-1}$$

It is clear that the amount of rewriting operations is unacceptable. Term rewriting proposes another strategy. One considers the identities in E as uni-directional rewrite rules:

$$\begin{aligned}(RG1)\ (xy)z &\to x(yz)\\ (RG2)\ ex &\to x\\ (RG3)\ x^{-1}x &\to e\end{aligned}$$

Here the idea is that identities will be only be applied in the direction that simplifies a given term. The strategy is to look for terms such that no more rules apply, so-called normal forms. Given two terms s, t we could:

1. Reduce them to normal forms $\hat{s}, \hat{t}$.
2. Check whether $\hat{s}, \hat{t}$ are syntactically equal.

There are two problems with this approach:

1. Equivalent terms can have distinct normal forms. For instance, xx^{-1} and e are normal forms with respect to (RG1) and (RG2) and we have shown they are equivalent. The above method would fail because they are syntactically distinct. The property needed to have uniqueness of normal forms is called confluence.

2. Normal forms may not exist. the process of reducing a term can lead to an infinite chain of rule applications. The property that ensures the existence of normal forms is called termination.

   If the rewrite system is not confluent we can use a technique called completion to extend it to a confluent one. In the case of groups, there exists a confluent and terminating extension.

2.2 Abstract reduction systems

Definition 2.2 (Abstract reduction system)**.**
Given a set A and a binary relation $\rightarrow$ in A.

An abstract reduction system is a pair $(A, \rightarrow)$. The binary relation $\rightarrow$ can be seen to represent computation steps.

We note $(a, b) \in \rightarrow$ as $a \rightarrow b$.

We want to determine if two terms are equivalent, that is, if $a \overset{*}{\leftrightarrow} b$. Our strategy is the one that we pointed out in the examples about groups: to reduce to a certain normal form that has a uniqueness property. The advantage of this approach is that an undirected search on $\rightarrow$ and $\leftarrow$ would be too expensive. First we need some notation for operations that are based on the composition of relations:

Given $R \subseteq A \times B \wedge S \subseteq B \times C$ two relations, its composition is:

$$R \circ S = \{(x, z) \in A \times C : \exists y \in B.(x, y) \in R \wedge (y, z) \in S\}$$

	$\overset{0}{\rightarrow} = \{(x, x) : x \in A\}$		$\overset{i+1}{\rightarrow} = \overset{i}{\rightarrow} \circ \rightarrow$
Transitive closure	$\overset{+}{\rightarrow} = \cup_{i>0} \overset{i}{\rightarrow}$	Reflexive transitive closure	$\overset{*}{\rightarrow} = \overset{+}{\rightarrow} \cup \overset{0}{\rightarrow}$
Reflexive closure	$\overset{=}{\rightarrow} = \rightarrow \cup \overset{0}{\rightarrow}$	Inverse	$\overset{-1}{\rightarrow} = \{(y, x) : x \rightarrow y\}$ and $\leftarrow = \overset{-1}{\rightarrow}$
Symmetric closure	$\leftrightarrow = \rightarrow \cup \leftarrow$	Transitive symmetric closure	$\overset{+}{\rightarrow} = (\leftrightarrow)^+$
Reflexive, transitive, symmetric closure	$\overset{*}{\rightarrow} = (\leftrightarrow)^*$		

Table 1: Notation for operations on relations

We give some terminology for abstract reduction systems:

x is reducible	$\exists y.x \rightarrow y$	x is irreducible or in normal form	$!\exists y.x \rightarrow y$
y is a normal form of x	$x \overset{*}{\rightarrow} y \wedge y$ is in normal form	unique normal form of x (if exists)	$x \downarrow$
y is direct successor of x	$x \rightarrow y$	y is successor of x	$x \overset{+}{\rightarrow} y$
x and y are joinable	$\exists z \in x \overset{*}{\rightarrow} z \overset{*}{\leftarrow} y$	x,y joinable	$x \downarrow y$

Table 2: Terminology for the elements of the reduction system

And some terminology for reductions:

Church-Rosser	$x \overset{*}{\leftrightarrow} y \implies x \downarrow y$	Semi-confluent	$y_1 \leftarrow x \overset{*}{\rightarrow} y_2 \implies y_1 \downarrow y_2$
Confluent	$y_1 \overset{*}{\leftarrow} x \overset{*}{\rightarrow} y_2 \implies y_1 \downarrow y_2$	Terminating	there is no infinite descending chain $a_0 \rightarrow a_1 \rightarrow \cdots$
Normalizing	every element has a normal form	Convergent	confluent and terminating

Table 3: Terminology for reductions in reductions systems

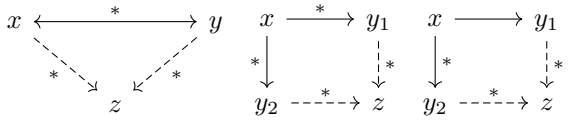
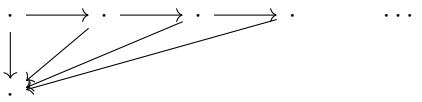


Figure 1: Church-Rosser, confluence and semi-confluence diagrams

Proposition 2.1 (Relation between the properties of an abstract reduction system)**.**
The following propositions relate the properties of abstract reduction systems:

1. *Any terminating relation is normalizing. The converse is not true.*
2. *Church-Rosser* $\iff$ *Confluence* $\iff$ *Semi-confluence.*
3. *Given a confluent relation* $\rightarrow$ *and* $x \overset{*}{\leftrightarrow} y$*:*
   1. *y is in normal form* $\implies x \overset{*}{\rightarrow} y$
   2. *x,y are in normal form* $\implies x = y$.

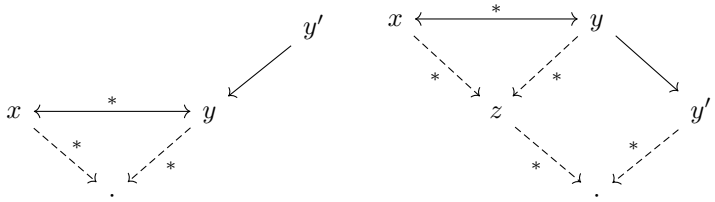
Proof. 1. Clearly, every terminating relation arrives to a normal form from any initial term. For the converse, consider the following acyclic, confluent and non-terminating relation:



2. 1) $\implies$ 2). Assume $y_1 \overset{*}{\leftarrow} x \overset{*}{\rightarrow} y_2$ then $x \overset{*}{\leftrightarrow} y$ and by Church-Rosser $\exists y_1 \downarrow y_2$.

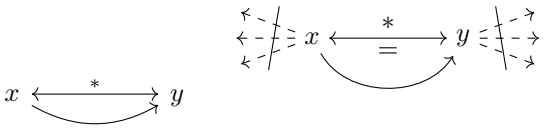
   2) $\implies$ 3). Trivial.

   3) $\implies$ 1). Assume $x \overset{*}{\leftrightarrow} y$. By induction, if $x = y$ then $x \downarrow y$. Assuming $x \overset{*}{\leftrightarrow} y \leftrightarrow y'$ and that $x \downarrow y$ we need to show $x \downarrow y'$. There are two cases:



   For the first case, we perform induction on the lower triangle and in the second, induction on the upper triangle and then semi-confluence for the quadrilateral.

3. We take advantage of the fact that confluence is equivalent to Church Rosser and consider the following diagrams:



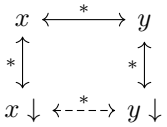
□

Proposition 2.2 (Results on the reduction strategy)**.**
The following results give us hints on what conditions should we look for to apply the strategy of our motivation:

1. *Condition for well-definition of* $x\downarrow$*:*
   (a) $\rightarrow$ *is confluent* $\implies$ *every element has at most one normal form.*
   (b) $\rightarrow$ *is normalizing and confluent* $\implies$ *every element has a unique normal form.*
2. *Equivalence test with normal forms:*
   If $\rightarrow$ *is confluent and normalizing then* $x \overset{*}{\leftrightarrow} y \iff x\downarrow = y\downarrow$
3. *Equivalence test with normal forms:*
   If $\rightarrow$ *is convergent then* $x \overset{*}{\leftrightarrow} y \iff x\downarrow = y\downarrow$

Proof. 1. (a) If x has two normal forms n_1^x, n_2^x by confluence we can join them. But since they're normal the only way to join them is that they are equal.
(b) Use 1 to proof that the number of normal forms is ≤ 1 and normality to proof that it is ≥ 1.

2. $x \overset{*}{\leftrightarrow} y \implies x\downarrow \overset{*}{\leftrightarrow} y \implies x\downarrow \implies x\downarrow = y\downarrow$.
   $x\downarrow = y\downarrow \implies x \overset{*}{\leftrightarrow} y$



3. This a weaker result than the previous but note that in the search for normal forms, the previous one requires a breadth-first search which may be more costly in some situations. Therefore, literature focus on the convergent case.

□

So now the question is, how do we prove confluence and termination?

2.2.1 Termination

Well-founded induction is a first tool to prove termination. It is a generalization of strong induction for natural numbers:

Definition 2.3 (Strong induction for natural numbers)**.**
Given property P on natural numbers.

$\forall n \in \mathbb{N}.P(n) \iff \forall n \in \mathbb{N}.(\forall m \in \mathbb{N}.m < n \implies P(m)) \implies P(n)$.

Note that the base case for 0 leaves the right hand-side with proving $P(0)$ just what we do in normal induction proofs. Now we abstract this principle from $(\mathbb{N}, >)$ to any reduction system $(A, \to)$:

Definition 2.4 (Well-founded induction (WFI) rule)**.**
Given a property P on the elements of A:

$$\frac{\forall x \in A.(\forall y \in A.x \xrightarrow{+} y \implies P(y)) \implies P(x)}{\forall x \in A.P(x)}$$

In this case, the base case reduces to prove $P(x)$ for every x without successor. It is interesting that this property holds exactly for terminating relations.

Theorem 2.3 (Characterization of terminating relations)**.**
Given $(A, \to)$ an abstract reduction system.

WFI holds $\iff$ $\to$ *terminates* $\iff$ $\forall B \subseteq A.\exists b \in B.\nexists b' \in B.b \to b'$

Proof. $\Rightarrow$) Set $P(x)$ = there is no infinite chain starting from x. Given any x the premise of WFI says that no successor of x has an infinite chain. Therefore, x itself cannot have an infinite chain. Equivalently, $P(x)$ holds.

$\Leftarrow$) If WFI doesn't hold for $\to$ then there must exist $a_0 \in A$ such that:

$$[\forall y \in A.a_0 \xrightarrow{+} y \implies P(y)) \implies P(a_0)] \wedge \neg P(a_0)$$

Therefore there must exist $a_1 \in A$ such that:

$$a_0 \xrightarrow{+} a_1 \wedge \neg P(a_1)$$

This process repeats itself giving a chain:

$$a_0 \xrightarrow{+} a_1 \xrightarrow{+} \cdots$$

that does not terminate. □

Definition 2.5 (Properties related to termination)**.**
A relation $\to$ is called:

1. **finitely branching** if each element has only a finite number of direct successors.
2. **globally finite** if each element has only finitely many successors.
3. **acyclic** if there is no element a such that $a \xrightarrow{+} a$.

Proposition 2.4 (Relation between properties related to termination)**.**
1. $\to$ terminating and finitely branching $\implies$ globally finite.
2. $\to$ acyclic and globally finite $\implies$ terminating.
3. $\to$ acyclic and finitely branching $\implies$ (globally finite $\iff$ terminating)
4. König's lemma: a finitely branching tree is infinite $\iff$ it contains an infinite path.

Different techniques to prove termination are presented in [11]. They imply the use of many orders such as lexicographic or multi-set orders which do not convey our main goal here of presenting general concepts towards completion algorithms. There is also an interesting comment on the characterization of termination with monotonic embeddings in $(\mathbb{N}, >)$ for finitely branching relations.

2.2.2 Confluence

We now look at techniques to prove confluence.

Definition 2.6 (Other notions of confluence)**.**
$\to$ is locally confluent $\iff \forall y_1, y_2 \in A. y_1 \leftarrow x \to y_2 \implies y_1 \downarrow y_2$

$\to$ is strongly confluent $\iff \forall y_1, y_2, x. y_1 \leftarrow x \to y_2 \implies \exists z. y_1 \overset{*}{\to} z \overset{=}{\leftarrow} y_2$

$\to$ has the diamond property $\iff \forall y_1, y_2, x. y_1 \leftarrow x \to y_2 \implies \exists z. y_1 \to z \leftarrow y_2$

Note that local confluence does not imply confluence as in the following example:

$$a \longleftarrow 0 \rightleftarrows 1 \longrightarrow b$$

Note also that the diamond property is stronger than strong confluence.

It is clear also that $\to$ is confluent $\iff \overset{*}{\to}$ has the diamond property.

Proposition 2.5 (Sufficient conditions for confluence)**.**
The following are sufficient conditions to show confluence:

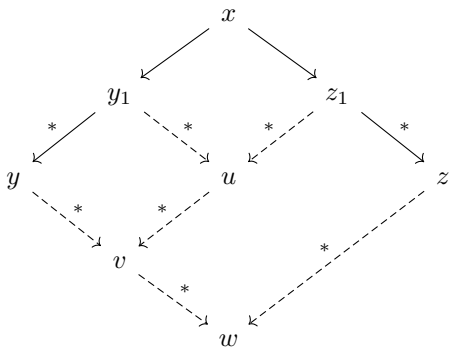
1. *Newman's lemma: $\to$ terminating and locally confluent $\implies$ confluent.*
2. *$\to$ strongly confluent $\implies$ confluent.*

Proof. 1. Well-founded induction on $P(x) = \forall y, z. y \overset{*}{\leftarrow} x \overset{*}{\to} z \implies y \downarrow z$. We analyse $y \overset{*}{\leftarrow} x \overset{*}{\to} z$:

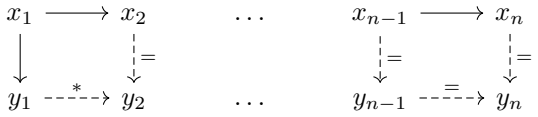
(a) If $x = y \vee x = z$ then $y \downarrow z$.

(b) In other case, $x \to y_1 \overset{*}{y} \wedge x \to z_1 \overset{*}{\to} z \overset{\text{local confluence}}{\implies} \exists u = y_1 \downarrow z_1$.

Since $x \overset{+}{\to} y_1 \wedge x \overset{+}{z_1} \overset{\text{induction}}{\implies} \exists v = y \downarrow u \wedge w = v \downarrow z \implies \exists w = y_1 \downarrow z_1$



2. We prove semi-confluence with $y_1 \leftarrow x_1 \overset{n}{\to} x_n \implies \exists y_n. y_1 \overset{*}{\to} y_n \overset{=}{\leftarrow} x_n$ and we apply induction over n.



□

To use strong confluence property one defines given a relation $\rightarrow$, a stronly confluent relation $\rightarrow_s$ such that $\overset{*}{\rightarrow}=\overset{*}{\rightarrow}_s$ and by the theorem above we have that $\rightarrow$ is confluent ($\overset{*}{\rightarrow}=\overset{*}{\rightarrow}_s \implies \rightarrow_1$ confluent $\iff \rightarrow_2$ confluent). A sufficient condition for $\overset{*}{\rightarrow}_1=\overset{*}{\rightarrow}_2$ is $\rightarrow_1 \subseteq \rightarrow_2 \subseteq \overset{*}{\rightarrow}_1$. Therefore we have the following:

Corollary 2.6 (Proving confluence by strong confluence)**.**
If $\rightarrow_1 \subseteq \rightarrow_2 \subseteq \overset{*}{\rightarrow}_1$ and $\rightarrow_2$ is strongly confluent then $\rightarrow_1$ is confluent.

Another strategy may be to split $\rightarrow = \rightarrow_1 \cup \rightarrow_2$ and prove that the confluence of each part gives the confluence of the total. The following concept is useful to prove this implication:

Definition 2.7 (Commutation)**.**
Two relations $\rightarrow_1, \rightarrow_2$:

1. commute: $\forall y_1, y_2, x. y_1 \overset{*}{\leftarrow}_1 x \overset{*}{\rightarrow}_2 y_2 \implies \exists z. y_1 \overset{*}{\rightarrow}_2 z \overset{*}{\leftarrow}_1 y_2 \iff \overset{*}{\leftarrow}_1 \circ \overset{*}{\rightarrow}_2 \subseteq \overset{*}{\rightarrow}_2 \circ \overset{*}{\leftarrow}_1$
2. strongly conmute: $\forall y_1, y_2, x. y_1 \leftarrow_1 x \rightarrow_2 y_2 \implies \exists z. y_1 \overset{=}{\rightarrow}_2 z \overset{*}{\leftarrow}_1 y_2 \iff \leftarrow_1 \circ \rightarrow_2 \subseteq \overset{=}{\rightarrow}_2 \circ \overset{*}{\leftarrow}_1$

Proposition 2.7 (Commutation properties)**.**
1. Commutation Lemma: If $\rightarrow_1, \rightarrow_2$ commute strongly then they commute.
2. Commutative Union Lemma: If $\rightarrow_1, \rightarrow_2$ are confluent and commute then $\rightarrow_1 \cup \rightarrow_2$ is confluent.

2.3 Universal algebra

2.3.1 General notions

A way to formalize the notion of computation is to introduce multi-typed algebraic structures:

Definition 2.8 (Signature and algebra)**.**
A **signature** is given by:

1. A set $S \neq \emptyset$ of types or sorts.
2. An indexed family in $S^\star \times S$:
$$\{\Sigma_{w,s} : w \in S^\star, s \in S\}$$
where $c \in \Sigma_{\epsilon,s}$ is a constant of type s and $\sigma \in \Sigma_{w=s(1)\ldots s(n),s}$ is an operation of type (w, s) and arity n.

Normally, one denotes a signature as $\Sigma = (S, \{\Sigma_{w,s} : w \in S^\star, s \in S\})$.

A **Σ-algebra** consists of:

1. A family $\{A_s : s \in S\}$ of data types.
2. A family $\{\Sigma^A_{w,s} : w \in S^\star, s \in S\}$ such that:
   (a) $\forall s \in S. \Sigma^A_{\epsilon,s} = \{c_A : c \in \Sigma_{\epsilon,s}\}$ where $c_A \in A_s$ is a constant of type s that instantiates the constant $c \in \Sigma_{\epsilon,s}$ for this algebra.
   (b) $\forall w \in S^+, s \in S. \Sigma^A_{w,s} = \{\sigma_A : \sigma \in \Sigma_{w,s}\}$ where $\sigma_A \in \Sigma^A_{w=s(1)\ldots s(n),s}$ is a function:
   $$\sigma_A : A_{s(1)} \times \ldots \times A_{s(n)} \mapsto A_s$$
   that interprets the function symbol σ for this algebra.

In other words, a Σ-algebra is an instantiation of the signature Σ for particular data-types. It is denoted $(\{A_s : s \in S\}, \{\Sigma^A_{w,s} : w \in S^\star, s \in S\})$.

EXAMPLE 2.1: In the following examples, triples $(a; b; c)$ denote the set of data-types, constants and operations in this order. Note that the signature is implicit in most of the examples.

1. Let $B = \{\top, \bot\}$ be the set of truth values. Then $(B; \top, \bot; \neg, \wedge, \vee, \oplus, \implies, \iff)$ is an algebra.
2. Let $\mathbb{N}$ denote the set of natural numbers and $Succ$ be the successor function $Succ(n) = n + 1$. Then $(\mathbb{N}; 0; Succ)$ is an algebra.
3. Algebraic structures such as rings or fields are clearly examples of algebras.
4. The most important example for us is the algebra of terms. In this case we explicitly name the underlying signature.

   The signature Σ will be given by a unique data type $S = \{s\}$ a set of constants denoted F_0 and a family of functions denoted with $F = \cup_{n\in\mathbb{N}} F_n$ where F_n gathers all the functions with arity n.

   The instantiation data-type is the set of terms:

   $$T(\Sigma, X) = F_0 \cup X \cup \{f(t_1, \ldots, t_n) : t_i \in T(\Sigma, X) \wedge f \in F_n \wedge n \in \mathbb{N}\}$$

   Here X is a set of variable symbols of type s such that $X \cap F_0 = \emptyset$.

   Finally, the algebra of terms is given by $T(\Sigma, X)$ as data-type, F_0 as a set of constants and a set of operations built from the signature: for each $f \in F_n$ one gives $F : T(\Sigma, X)^n \to T(\Sigma, X)$ such that $(t_1, \ldots, t_n) \mapsto f(t_1, \ldots, t_n)$.

   The algebra of terms is usually denoted by $T(\Sigma, X)$ (same notation as the underlying datatype).

As usual in mathematics, it is interesting to introduce the notion of substructure and homomorphism between structures:

Definition 2.9 (Σ-sub-algebra)**.**
Let A, B be Σ-algebras indexed in S. B is a sub-algebra of A, which we write, $B \leq A$ if:

1. $\forall s \in S. B_s \subseteq A_s$
2. $\forall s \in S, c \in \Sigma_{\epsilon,s}. c_B = c_A$.
3. $\forall s \in S, w \in S^+, \sigma \in \Sigma_{w,s}, (b_1, \ldots, b_n) \in B^w. \sigma_B(b_1, \ldots, b_n) = \sigma_A(b_1, \ldots, b_n)$

Definition 2.10 (Homomorphism of algebras)**.**
Let A, B be two Σ-algebras indexed in S.

A Σ-homomorphism $\phi : A \to B$ is an S-indexed family of mappings $\phi = \langle \phi_s : A_s \to B_s : s \in S\rangle$ such that:

1. $\forall s \in S, c \in \Sigma_{\epsilon,s}. c_B = \phi_s(c_A)$
2. $\forall w = s(1) \ldots s(n) \in S^+, s \in S, \sigma \in \Sigma_{w,s}, (a_1, \ldots, a_n) \in A^w. \phi_s(\sigma_A(a_1, \ldots, a_n)) = \sigma_B(\phi_{s(1)}(a_1), \ldots, \phi_{s(n)}(a_n))$

Of course, the notation is greatly simplified if we consider only one underlying data-type $S = \{s\}$.

Our goal now is to show how the algebra of terms can be formally studied to make useful deductions.

2.3.2 Syntactic characterization of $\overset{*}{\leftrightarrow}$

Definition 2.11 (Positions and size of a term)**.**
Given terms $s \in T(\Sigma, X)$ its **set of positions** is defined inductively as:

1. $s = x \in X \implies Pos(s) = \{\epsilon\}$
2. $s = f(s_1, \cdots, s_n) \implies Pos(s) = \{\epsilon\} \bigcup \cup_{i=1}^{n} \{ip : p \in Pos(s_i)\}$

We call the position ϵ root position and the function or variable symbol at this position is called the root symbol.

The **size** of $s \in T(\Sigma, X)$ is $|s| = |Pos(s)|$.

Definition 2.12 (Order on positions)**.**
The prefix order is a partial order on the set of positions:

$p \leq q \iff \exists p'.pp' = q$

With respect to $\leq$, we say p, q are:

1. parallel, $p||q$, if they are incomparable.
2. p is above q, if $p \leq q$.
3. p is strictly above q, if $p < q$.

Definition 2.13 (Term operations)**.**
Given $p \in Pos(s)$ the **sub-term of s at position p**, $s|_p$, is defined by induction on the length of p:

1. $s|_\epsilon = s$
2. $f(s_1, \cdots, s_n)|_{iq} = s_i|_q$.

The term obtained from s by **replacing the sub-term at position p by t**, $s[t]_p$ is:

1. $s[t]_\epsilon = t$
2. $f(s_1, \cdots, s_n)[t]_{iq} = f(s_1, \cdots, s_i[t]_q, \cdots, s_n)$

Finally, $Var(s) = \{x \in X : \exists p \in Pos(s).s|_p = x\}$ denotes the **set of variables occurring in s** and the corresponding positions are called variable positions.

Definition 2.14 (Ground terms)**.**
$s \in T(\Sigma, X)$ is a **ground** term if $Var(s) = \emptyset$. $T(\Sigma)$ denotes the set of all ground terms over Σ.

Definition 2.15 (Substitutions)**.**
Suppose that V is a countably infinite set of variables.

A $T(\Sigma, V)$**-substitution** is a function $\sigma : V \to T(\Sigma, V)$ such that $\sigma(x) \neq x$ for only finitely many variables. Associated with a substitution we have the following notions:

1. The **domain** of σ is $Dom(\sigma) = \{x \in V : \sigma(x) \neq x\}$.
2. The **range** of σ is the set $Ran(\sigma) = \{\sigma(x) : x \in Dom(\sigma)\}$.
3. The **variable range** of σ is $VRan(\sigma) = \cup_{x \in Dom(\sigma)} Var(\sigma(x))$.

$Sub(T(\Sigma, V))$ denotes the set of all $T(\Sigma, V)$-substitutions.

We **extend** the application of a substitution to terms as follows:

Given $\sigma \in Sub$, define $\hat{\sigma} : T(\Sigma, V) \to T(\Sigma, V)$ such that $\hat{\sigma}(x) = \begin{cases} \sigma(x) & \text{if } x \in V \\ f(\hat{\sigma}(s_1), \cdots, \hat{\sigma}(s_n)) & \text{if } s = f(s_1, \cdots, s_n) \end{cases}$

Then one can define the composition of σ and τ as $\sigma\tau(x) = \hat{\sigma}(\tau(x))$. This has the following properties:

1. $\sigma\tau$ is again a substitution.
2. The composition of substitutions is associative.
3. $\hat{\sigma\tau} = \hat{\sigma}\hat{\tau}$

An instance of term s is a term t such that $\exists\sigma$ substitution such that $\sigma(s) = t$. We note it as $t \gtrsim s$ and $t > s$ if the opposite relation does not hold.

Definition 2.16 (Identities and reduction relation)**.**
An **Σ-identity** is a pair $(s, t) \in T(\Sigma, V) \times T(\Sigma, V)$. We write $s \approx t$ and call s left-hand side and t right-hand side.

Let E be a set of Σ-identities. The **reduction relation** $\to_E \subseteq T(\Sigma, V) \times T(\Sigma, V)$ is defined as

$$s \to_E t \iff \exists (l, r) \in E, p \in Pos(s), \sigma \in Sub.s|_p = \sigma(l) \wedge t = s[\sigma(r)]_p$$

This relation is quite intuitive. We take the equation (l, r) as a template, whose left part matches up to a transformation of a sub-term of the source term s. Then, we apply the equivalence (l, r) and substitute the right term modified properly. This should give us all the derived terms.

Definition 2.17 (Properties of binary relations on $T(\Sigma, V)$)**.**
Let $\equiv$ be a binary relation on $T(\Sigma, V)$. $\equiv$ is:

1. **closed under substitutions:**

   $\forall s, t \in T(\Sigma, V), \sigma \in Sub.s \equiv t \implies \sigma(s) \equiv \sigma(t)$

2. **closed under Σ-operations:**

   $\forall f \in \Sigma^{ar(f)}, s_i, t_j \in T(\Sigma, V).(\forall i.s_i \equiv t_i) \implies f(s_1, \ldots, s_{ar(f)}) \equiv f(t_1, \ldots, t_{ar(f)})$

3. **compatible with Σ-operations:**

   $\forall f \in \Sigma^{arg(f)}, s_j, s, t \in T(\Sigma, V).s \equiv t \implies f(s_1, \ldots, s_{i-1}, s, s_{i+1}, \ldots, s_{ar(f)}) \equiv f(s_1, \ldots, s_{i-1}, t, s_{i+1}, \ldots, s_{ar(f)})$

4. **compatible with Σ-contexts:**

   $\forall t \in T(\Sigma, V), p \in Pos(t).s \equiv s' \implies t[s]_p \equiv t[s']_p$

Theorem 2.8 (Characterization of $\overset{*}{\leftrightarrow}_E$)**.**
1. Let E be a set of Σ-identities. The reduction relation $\to_E$ is closed under substitutions and compatible with Σ-operations.
2. Given a binary relation $\equiv$ on $T(\Sigma, V)$:

- *$\equiv$ is compatible with Σ-operations $\iff$ it is compatible with Σ-contexts.*
- *If $\equiv$ is reflexive and transitive then it is compatible with Σ-operations $\iff$ it is closed under Σ-operations.*

3. Let E be a set of Σ-identities. The relation $\overset{*}{\leftrightarrow}_E$ is the smallest equivalence relation on $T(\Sigma, V)$ that contains E and is closed under substitutions and Σ-operations.

Proof. 1. Follows from definition. 2. Definition and induction. 3. Clearly, $\overset{*}{\leftrightarrow}$ contains E. By definition, $\overset{*}{\leftrightarrow}$ is an equivalence relation. Using parts 1,2 one shows that $\overset{*}{\leftrightarrow}$ is closed under substitutions and Σ-operations.

Finally, one assumes that $\equiv$ is another equivalence relation containing E and closed under substitutions and Σ-operations. Then one shows that $s \rightarrow_E t \implies s \equiv t$. Since by definition, $\overset{*}{\leftrightarrow}$ is the smallest equivalence relation containing $\rightarrow_E$, we deduce that $\overset{*}{\leftrightarrow} \subseteq \equiv$. □

The last property, tells us that $\overset{*}{\leftrightarrow}_E$ is obtained from E, closing it under reflexivity, symmetry, transitivity, substitutions and Σ-operations. This process can be described with inferences rules and leads to **equational logic**:

$\dfrac{(s \approx t) \in E}{E \vdash s \approx t}$
$\dfrac{}{E \vdash t \approx t}$
$\dfrac{E \vdash s \approx t}{E \vdash t \approx s}$
$\dfrac{E \vdash s \approx t \quad t \approx u}{E \vdash s \approx u}$
$\dfrac{E \vdash s \approx t}{E \vdash \sigma(s) \approx \sigma(t)}$
$\dfrac{E \vdash s_1 \approx t_1 \quad \dots \quad s_n \approx t_n}{E \vdash f(s_1, \dots, s_n) \approx f(t_1, \dots, t_n)}$

Table 4: Inference rules of equational logic

We note $E \vdash s \approx t$ to specify that $s \approx t$ can be obtained from E applying the above rules. We read it as "$s \approx t$ is a syntactic consequence of E". The first rule receives the special name of **assumption rule**. One can also build **proof trees** by composing inference rules in the usual manner. There are however differences to note between both approaches:

1. The rewriting approach given by the computation of $\overset{*}{\leftrightarrow}$ allows the replacement of a sub-term at an arbitrary position in a single step (choosing $l \approx r, \sigma, p$). The inference rule approach needs to simulate this by many small steps of closure under single operations.

2. Closure under operations in the inference rule approach allows the simultaneous replacement in each argument of an operation. The rewriting approach needs to simulate this by a number of replacements steps in sequence.

2.3.3 Semantic characterization of $\overset{*}{\leftrightarrow}$

Definition 2.18 (Models for a set of identities)**.**
Let $\mathcal{A}$ be a Σ-algebra and $s \approx t$ a Σ-identity:

$s \approx t$ **holds** in $\mathcal{A} \iff \forall \phi : T(\Sigma, V) \rightarrow A$ homomorphism.$\phi(s) = \phi(t)$.

In that case, we denote it as $\mathcal{A} \models s \approx t$.

Let E be a set of Σ-identities.

$\mathcal{A}$ is a **model** of E $\iff \forall (s \approx t) \in E . \mathcal{A} \models s \approx t$.

In that case, we denote it as $\mathcal{A} \models E$ and $\nu(E)$ the class of all models or Σ**-variety** defined by E.

Here we stress the use of word "identity" for a pair of term $s \approx t$ to express that this equality is assumed to hold in an algebra. The word "equation" is used to express that the equality must be solved in an algebra. This reduces to a universally or existentially quantified goal.

Definition 2.19 (Equational theory)**.**
$s \approx t$ is a **semantic consequence** of E $\iff \forall \mathcal{A} \in \nu(E).\mathcal{A} \models s \approx t$.

In that case, we denote it as $E \models s \approx t$.

The relation $\approx_E = \{(s,t) \in T(\Sigma, V) \times T(\Sigma, V) : E \models s \approx t\}$ is the **equational theory** induced by E.

Theorem 2.9 (Birkhoff's theorem)**.**
Let E be a set of identities.

The syntactic consequence relation $\overset{*}{\leftrightarrow}$ *coincides with the semantic consequence relation* $\approx_E$*.*

Alternatively, we say that $\vdash$ *and* $\models$ *coincide.*

The meaning of these theorem is that the syntactic procedure always leads to valid equations and that any valid equations can be obtained through the syntactic procedure.

2.4 A basic completion algorithm

Given a set of identities E we want to find a convergent term rewriting system R equivalent to E, that is, $\approx_E = \approx_R$. In this section we limit ourselves to give a basic completion algorithm trying to precise the meaning of the concepts implied.

As a further remark, we should note that this algorithm would give the convergent reduction system for the word problem of our motivation on groups. [12] deduces formally from page 45-49 this particular case obtaining the completed system:

$e \cdot x \to x$
$x^{-1}x \to e$
$(x \cdot y) \cdot z \to x \cdot (y \cdot z)$
$x^{-1}(xz) \to z$
$ye \to y$
$(y^{-1})^{-1} \to y$
$e^{-1} \to e$
$y \cdot y^{-1} \to e$
$y \cdot (y^{-1} \cdot x) \to x$
$(x \cdot y)^{-1} \to y^{-1}x^{-1}$

Table 5: Completed system for the word problem of groups

Before the statement of the completion algorithm we need to define some of the notions involved.

2.4.1 Term rewriting systems

Theorem 2.10 (Equational theory of convergent system is decidable)**.**
If E is finite and $\rightarrow_E$ is convergent then $\approx_E$ is decidable.

Proof. By the equivalence test with normal forms, $\overset{*}{\leftrightarrow}_E t \iff s\downarrow_E = t\downarrow_E$. We need to show that operator $\downarrow_E$ is decidable. This reduces to determine whether a term u is in normal form or not. To do so, we pick $(l \approx r) \in E$, $p \in Pos(u)$ and check if there is a substitution σ such that $u|_p = \sigma(l)$. If there isn't such a substitution then we have a normal form. If there is such a substitution then one iterates the process with $u[\sigma(r)]|_p$. This iteration must terminate because $\rightarrow_E$ is terminating. And determining the existence of such a substitution can be done in linear time (exercise 4.24 of the book). □

Definition 2.20 (Word problem)**.**
The **word problem** for E is the problem of deciding $s \approx_E t$ for $s, t \in T(\Sigma, V)$. The **ground word problem** for E is the word problem restricted to ground term $s, t \in T(\Sigma, \emptyset)$.

Definition 2.21 (Term rewriting systems)**.**
A **rewrite rule** is an identity $l \approx r$ such that l is not a variable and $Var(l) \supseteq Var(r)$.

A **term rewriting system (TRS)** is a set of rewrite rules. The default notation is R.

In the context of TRS, a **reducible expression (redex)** is an instance of the lhs of a rewrite rule and **contracting** the redex means replacing it with the corresponding instance of the rhs of the rule.

Given that any TRS R is a particular set of identities, we will use notation $\rightarrow_R$ for the rewrite relation induced by R and we will say that R is terminating, confluent, convergent, etc. if $\rightarrow_R$ has the corresponding properties. In this context, the previous theorem reformulates as follows (any terminating system is a TRS):

Corollary 2.11 (Equational theory of convergent term rewriting systems is decidable)**.**
If R is a finite convergent TRS, $\approx_R$ is decidable: $s \approx_R t \iff s\downarrow_R = t\downarrow_R$.

Definition 2.22 (Reduction order)**.**
Let Σ be a signature and V be a countably infinite set of variables.

A strict order $>$ (irreflexive, asymmetric, transitive) on $T(\Sigma, V)$ is called a **rewrite order** if and only if:

1. compatible with Σ-operations
2. closed under substitutions

A **reduction order** is a well-founded rewrite order.

Theorem 2.12 (Motivation for reduction orders)**.**
Let R be a term rewriting system.

R terminates $\iff$ $\exists$ a reduction order $>$ such that $\forall(l \rightarrow r) \in R. l > r$.

Proof. $\Rightarrow$) Choose $\overset{+}{\rightarrow}_R$ as reduction order.

$\Leftarrow$) Use definitions to show that hypothesis imply that $\forall s_1, s_2 \in T(\Sigma, V). s_1 \rightarrow_R s_2 \implies s_1 > s_2$. Since $>$ is well-founded, an infinite chain $s_1 \rightarrow_R s_2 \rightarrow_R \ldots$ gives an infinite chain $s_1 > s_2 > \ldots$ which gives a contradiction. □

2.4.2 Critical pairs

The last theorem of the previous section gives us the tool that will guarantee the construction of a terminating system in the completion algorithm. We know deal with the notion that will guarantee the construction of a confluent system. We profit to remark that deciding if a TRS is terminating or confluent is undecidable in the general case.

Definition 2.23 (Most general substitutions and renamings)**.**
A substitution σ is **more general** than a substitution σ' if there is a substitution δ such that $\sigma' = \delta\sigma$. In this case we write $\sigma \lesssim \sigma'$. We also say that σ' is an **instance** of σ.

We write $\sigma \sim \sigma' \iff \sigma \lesssim \sigma' \wedge \sigma \gtrsim \sigma'$.

A **renaming** is a substitution that acts as a bijection on the set of variables V.

Definition 2.24 (Syntactic unification problem)**.**
Given E a set of identities and two terms s,t, find a substitution σ such that $\sigma s \approx_E \sigma t$.

It is an undecidable problem.

We focus in the case $E = \emptyset$. In this case, $\sigma s \approx_\emptyset \sigma t \equiv \sigma s = \sigma t$ and such a σ is called a unifier of s and t or a solution of equation $s =^? t$.

A **unification problem** is a finite set of equations $S = \{s_i =^? t_i : i \in \{1, \ldots, n\}\}$.

A **unifier or solution** of S is a substitution such that $\forall i \in \{1, \ldots, n\}.\sigma s_i = \sigma t_i$.

Let $\mathcal{U}(S)$ be the set of all unifiers of S, so that S is unifiable if $\mathcal{U}(S) \neq \emptyset$.

A **most general unifier (mgu)** of S is a least element of $\mathcal{U}(S)$, that is:

$\sigma \in Sub$ such that $\sigma \in \mathcal{U}(S) \wedge \forall \sigma' \in \mathcal{U}(S).\sigma \lesssim \sigma'$.

Definition 2.25 (Critical pair)**.**
Let $l_i \to r_i$ with $i = 1, 2$ be two rules whose variables have been renamed in a way that $Var(l_1, r_1) \cap Var(l_2, r_2) = \emptyset$.

Let $p \in Pos(l_1)$ be such that $l_1|_p$ is not a variable and let θ be a mgu of $l_1|_p =^? l_2$. This determines a **critical pair** $\langle \theta r_1, (\theta l_1)[\theta r_2]_p \rangle$.

The set of all critical pairs is denoted by $CP(R)$.

In particular, one can see that critical pairs are formed by equational consequences of R using the syntactic characterization and Birkhoff's theorem.

Proposition 2.13 (Properties of critical pairs)**.**
We state the necessary elements to make critical pairs useful for the theory:

1. *Critical Pair Lemma:*

   If $s \to_R t_1, t_2$ then $t_1 \downarrow_R t_2 \vee \exists u_1, u_2.t_i = s[u_i]_p$ and $\langle u_1, u_2 \rangle$ or $\langle u_2, u_1 \rangle$ are critical pairs of R.

2. *Critical Pair Theorem: A TRS is locally confluent $\iff$ all its critical pairs are joinable.*

3. *A terminating TRS is confluent $\iff$ all its critical pairs are joinable.*

4. *Confluence of a finite and terminating TRS R is decidable.*

2.4.3 The basic algorithm

The basic completion procedure works as follows. In a first initialization step it removes the trivial identities $s \approx s$ and tries to orient the remaining non-trivial identities according to the reduction order.

Then, it computes all critical pairs of the rewrite system obtained. The terms in each pair are reduced to its normal forms. If identical the pairs are joinable and we are done. In other case, one tries to orient the normal terms into a rewrite rule whose termination can be shown using $>$ and adds the rewrite rule to the current rewrite system.

The process is iterated until failure or until the current rewrite system remains unchanged in an iteration which would mean that the system does not have non-joinable critical pairs, that is, it is confluent.

Algorithm Basic completion procedure

Input: A finite set E of Σ-identities and a reduction order $>$ on $T(\Sigma, V)$.

Output: A finite convergent term rewriting system R that is equivalent to E, if the procedure terminates successfully. Otherwise, it returns "Fail".

Initialization: If there exists $(s \approx t) \in E$ such that $s \neq t, s \not> t, t \not> s$ (the rule cannot be oriented) then terminate with output "Fail". Otherwise $i = 0$, $R_0 = \{l \to r : (l \approx r) \in E \cup E^{-1} \wedge l > r\}$

repeat
  $R_{i+1} = R_i$
  for all $\langle s, t\rangle \in CP(R_i)$ **do** do
    Reduce s, t to some R_i-normal forms $\hat{s}, \hat{t}$.
    If $\hat{s} \neq \hat{t}$ and $!(\hat{s} > \hat{t} \vee \hat{t} > \hat{s})$ then terminate with output "Fail".
    If $\hat{s} > \hat{t}$ then $R_{i+1} \cup \{\hat{s} \to \hat{t}\}$.
    If $\hat{t} > \hat{s}$ then $R_{i+1} = R_{i+1} \cup \{\hat{t} \to \hat{s}\}$
  end for
  $i = i + 1$
until $R_i = R_{i-1}$
output R_i

The algorithm can therefore yield three different behaviours (see details in [11]):

1. End with failure. One can try a different reduction order.
2. End successfully. We obtain a finite convergent TRS equivalent to E which gives a decision procedure for the word problem for $\approx_E$.
3. Loop forever. In this case the union of all the partial TRS is an infinite convergent TRS equivalent to E which gives a semi-decision procedure for $\approx_E$.

We leave for the future to investigate and implement the practical variations of this algorithm.

3 Conclusion and future work

There are several directions that we could not follow during our summer stay, either because we did not have the time for it or because we were not aware of them.

Firstly, we should note that the integration of Welder with Intellij Idea greatly improved our perception of the proving process with Welder. It may seem a minor aspect but from our experience debugging proofs (probably in aspects not related with the proof itself) was the most time-consuming activity for users of the tool.

More importantly, now we are more aware of what are the capabilities of the SMT $\rightarrow$ Inox $\rightarrow$ Welder stack. In particular, we were interested in the concept of tactic and our surprise was that our experiments did not show the alledged advantages of using them in the theorem prover. It may be the case that Inox's unrolling procedure is doing part of the work here or maybe we were not building properly the tactics. In any case, now we have the tools for inspecting closely the proving process and find out what is happening. Regarding theorem proving it may be also worth to read through "Designing a theorem prover" by L. Paulson in [12].

We have given some remarks on term rewriting based on the notes that we took while working through [11]. We have put in context certain aspects specially using [12]. However, we were not aware of the existence of any serious implementation of the concepts explained and our task focused on obtaining a quick way towards the completion procedures which are the real strength of the theory. Having done a careful study of the main concepts of term rewriting theory one should look at what existing solutions there are in the field. We found that Haskell has already such a library. See [13] and [14]. The question that remains to be answered is how to properly integrate term rewriting in Welder. We hope to answer this in the future.

Another topic that has attracted our attention and may be interesting for the future are Gröbner basis. The usual development of the topic [15] for mathematicians does not present it as a term rewriting strategy as [11] does. Gröbner basis could be applied for theorem proving in geometry [16].

References

[1] *Debugging the code of an external library in Intellij Idea*. 2017. URL: https://stackoverflow.com/questions/43396002/debugging-the-code-of-an-external-library-in-intellij-idea (visited on 04/08/2017).

[2] *Online dependence not found in .ivy2 file and not found by Intellij Idea*. 2017. URL: https://stackoverflow.com/questions/45463978/online-dependence-not-found-in-ivy2-file-and-not-found-by-intellij-idea (visited on 04/08/2017).

[3] *Problems with SBT dependencies in Intellij Idea*. 2017. URL: https://stackoverflow.com/questions/45471836/problems-with-sbt-dependencies-in-intellij-idea (visited on 04/08/2017).

[4] *NoClassDefFoundError using sbt for builds and imports in Intellij Idea*. 2017. URL: https://stackoverflow.com/questions/45486294/noclassdeffounderror-using-sbt-for-builds-and-imports-in-intellij-idea (visited on 08/04/2017).

[5] *Integrating Intellij Idea and SBT*. 2017. URL: https://intellij-support.jetbrains.com/hc/en-us/community/posts/115000475984-Integrating-Intellij-Idea-and-SBT (visited on 08/04/2017).

[6] *Integrating Intellij Idea and SBT II*. 2017. URL: https://intellij-support.jetbrains.com/hc/en-us/community/posts/115000493770-Integrating-Intellij-Idea-and-SBT-II?page=1#community_comment_115000407610 (visited on 04/08/2017).

[7] Josh Suereth and Matthew Farwell. *SBT in Action*. Manning Publications, 2015.

[8] Rustan Leino. "Automating induction with an SMT solver". In: ().

[9] Andrew Reynolds and Viktor Kuncak. "Automating induction with an SMT solver". In: ().

[10] *Automatic theorem proving tools for Welder*. 2017. URL: https://github.com/rjraya/WelderToolbox (visited on 01/09/2017).

[11] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1999.

[12] Abramsky et alii. *Handbook of Logic in Computer Science*. Oxford University Press, 1992.

[13] *The term-rewriting package*. 2017. URL: https://hackage.haskell.org/package/term-rewriting (visited on 01/09/2017).

[14] Bertram Felgenhauer et alii. "A Haskell Library for Term Rewriting". In: ().

[15] David Cox et alii. *Ideals, Varieties and Algorithms*. Springer, 1997.

[16] D. Petrovic. "Automated Proving in Geometry using Gröbner Bases in Isabelle/HOL". In: ().