Chapter 2

Data Type Specification

Specification of data types using many sorted equational logic is the most classical level of algebraic specification. This constitutes the topic of this chapter which is structured as follows.

1. We introduce the basic concepts of signature, axioms, algebra, and satisfaction both as specification concepts and as mathematical ingredients of equational logic. In fact, conceptually these two sides are identical, the only difference being in the notation used. Here we also develop the foundational result of existence of initial algebras for conditional equational specifications.
2. In the next section we discuss how to deal with some forms of partiality of the operations, namely those handled by subsorting and by sort constraints.
3. Next we discuss equational deduction as a finitary proof system for the semantic consequence relation in equational logic. We introduce the concepts of proof rule and entailment systems at a rather general level in order to make them available for other deduction systems later in our lecture notes. The important results developed here are the soundness and the completeness of equational deduction.
4. In order to mechanize equational deduction and make it computational, we introduce the rewriting technique. Rewriting is the basic verification mechanism and the main execution engine for equational specifications. We show that while as a logical deduction system rewriting is sound, it is complete only under some special conditions. We develop these results in a more abstract form than traditional term rewriting, a fraemwork which covers also the important case of rewriting modulo axioms.
5. An important aspect of the formal verifications of data type specifications are the induction proofs, meaning the proof of logical properties (equations in our case) holding in the initial algebras of the respective specifications. These ‘inductive’ properties admit only infinitary complete proof systems. For dealing with this we develop the finitary proof method of ‘structural induction’.

6. The final section of the chapter illustrates the material developed so far through a dedicated example, namely that of a specification of a simple compiler. This is based upon the so-called ‘initial algebra semantics’ method. Its correctness appears as an inductive equational property, for which we give a rather simple proof score.

2.1 Basic Notions

Data type specification consists of axiomatic descriptions of sets of elements together with certain relevant functions on these sets. The word ‘basic’ means that we do not consider any structuring mechanism for the specifications. Structuring of specifications is the topic of Chap. 6.

Basic Specification. Consider the following simple specification of natural numbers with addition, written in CafeOBJ notation.

```
mod! SIMPLE-NAT {
  [ Nat ]
  op 0 : -> Nat
  op s_ : Nat -> Nat
  op _+_ : Nat Nat -> Nat
  vars M N : Nat
  eq N + (s M) = s(N + M) .
  eq N + 0 = N .
}
```

This specification consists of several distinct parts:

1. Header (i.e., the part before {...}) giving the name of the specification (`SIMPLE-NAT`) and the kind of denotation (`mod!`).
2. The sort (type) declaration (`[Nat]`), giving a name to the set of elements of the specified sort (or ”type”) (in this case the set of the natural numbers); while the name ”sort” is common in algebraic specification, the name ”type” is more used in programming.
3. The operations declarations, starting with `op`, denoting functions on the set(s) of elements. In our case these are `0`, `s_` and `_+_`, respectively, as expected denoting, the zero element, the succesor function, and the addition of naturals, respectively.
4. The variables declarations, starting with `vars`, (such as `M` and `N`). (CafeOBJ supports the declarations of the variables on the spot, for example `M:Nat`.)
5. Axioms (the statements starting with `eq`) defining the equality between elements.

Signatures. The sort and the operation declarations form the *signature* of the specification. The *operation declarations* consist of:

- The **name** of the operation. Some specification languages, including CafeOBJ, support the so-called *mix-fix* syntax for the name of the operations, showing the position of the arguments by "_" when writing an application of the operation. The mix-fix syntax enhaces greatly the readability of the specifications since it brings them closer to the common mathematical notations with whom we are familiar from school or from our programming practice.

- The **arity** of the operation, which is a string of (already declared) sorts that correspond to the sorts of the arguments for the operation. An arity of an operation may thus consist of

  - an empty string (like in the case of `0`); such operations are called *constants*,

  - only one sort (like '`Nat`' in the case of `s_`), or

  - several sorts (like '`Nat Nat`' in the case of `_+_`); in general these sorts may also be different.

- The **sort** of the operation, that is an already declared sort symbol. Note that all three operations of our specification have the same sort, namely `Nat`.

  The following is the mathematical definition of the concept of signature.

Definition 2.1 (Signatures)**.** *We let S^* denote the set of all finite sequences of elements from S, with $[]$ the empty sequence. A(n S*-sorted*)* signature (S,F) *is an $S^* \times S$-indexed set $F = \{F_{w \to s} \mid w \in S^*,\ s \in S\}$ of* operation symbols.

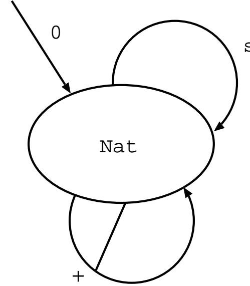
The sets $F_{w\to s}$ in the definition above stand for the sets of symbols with arity w and sort s. Note that this definition permits *overloading*, in that the sets $F_{w\to s}$ need *not* be disjoint. We may denote $F_{[]\to s}$ simply as $F_{\to s}$.

Graphical representation of the signature can be very useful. The graphical notation for signatures that we are using here was first introduced by the ADJ group as an extension of the classical set theory graphical representation of sets and functions. In this notation we represent

- sorts (types) by disks, and

- operations by multi-source arrows.

For example, the signature `SIMPLE-NAT` can be graphically represented as follows:



Formal variables. Given a signature (S,F), a *variable declaration* introduces a new constant symbol of a declared sort. The formal treatment of variables is not straightforward because it needs to avoid various clashes. The following definition formalizes the actual treatment of variables in specification languages, such as CafeOBJ.

Definition 2.2 (Variable)**.** *Let* (S,F) *be a signature. A* variable for (S,F) *is a triple* $(x,s,(S,F))$ *where x is the* name of the variable*,* $s \in S$ *is the* sort of the variable*, and* (S,F) its signature.

The sort and the signature are essential qualifications for variables, very much used in the treatment of the variables by various implementations of actual specification languages.. However, when these are clear, we may simply refer to a variable by its name only. For example, if X is a set of variables for (S,F), then $(x,s,(S,F)) \in X$ may be denoted simply $x \in X$. For this to make sense, but also in order to avoid other kinds of clashes, we make the basic assumption valid all over our material, that when considering sets of variables, *any two different variables have different names*.

Notation 2.3. *For any signature* (S,F) *and any set X of variables for* (S,F)*, the signature* $(S,F\cup X)$ *denotes the extension of* (S,F) *with X as (new) constants that respects the sorts of the variables. This means* $(F\cup X)_{w\to s} = F_{w\to s}$ *when w is not empty and* $(F\cup X)_{\to s} = F_{\to s} \cup \{(x,s,(S,F)) \mid (x,s,(S,F)) \in X\}$.

Due to set theoretic arguments (that we omit here) the latter union is always a disjoint one.

Terms. Terms are syntactic constructs which can be defined recursively as operations applied to arguments which are either complex terms, or primitive terms which are constants. These constants can be either constants of the specified signature or declared variables. If a term contains variables, then it is considered to belong to the corresponding extended signature. The application of operations to the arguments have to respect the arity of the operation, i.e. the argument must have the sort indicated by the arity. This is captured mathematically by the definition below.

Definition 2.4 (Terms)**.** *An* (S,F)*-term t of sort* $s \in S$*, is a structure of the form* $\sigma(t_1,\dots,t_n)$*, where* $\sigma \in F_{s_1\dots s_n\to s}$ *and* $t_1,\dots,t_n$ *are* (S,F)*-terms of sorts* $s_1,\dots,s_n$*, respectively.*

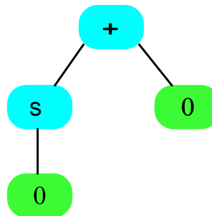
In CafeOBJ, as in other specification languages, by the command `parse` we can check the well-formedness of a term, i.e. that a certain expression is indeed a term, :

```
SIMPLE-NAT> parse s 0 0 .
[Error] no successful parse
```

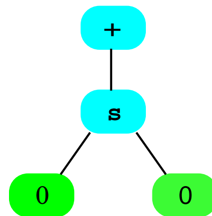
or

```
SIMPLE-NAT> parse s 0 + 0 .
((s 0) + 0) : Nat
```

(Here `SIMPLE-NAT>` is a system prompt.)
The first term is ill-formed, hence the parsing error. The second term is well-formed, and the system parses the term as



and tells the user that the sort of the term is `Nat`.
Note that another possible parsing for `s 0 + 0` is `s(0 + 0)`:



However, the system does not choose this possibility because, by internal convention, the operation `s_` has higher precedence than the operation `_+_`.

Equations. The axioms of equational specifications are called *equations*. They are formal equalities between (well formed) terms of the same sort and denote actual equalities between elements.

Definition 2.5 (Equations)**.** *Given a signature* (S,F)*, a* quantifier-free equation*, often called simply* equation*, is a symbolic equality* $t = t'$ *between* F*-terms* t *and* t' *of the same sort.*

The equations are the simplest sentences or axioms, often they are called *atomic* sentences. From equations we can form more complex sentences by application of logical connectives and quantifiers according to the following rules:

Definition 2.6 (Sentence)**.** *For any signature (S,F), the set of (S,F)-*sentences *is the least set such that:*

- *Each (S,F)-equation is an (S,F)-sentence.*
- *If ρ_1 and ρ_2 are (S,F)-sentences then $\rho_1 \wedge \rho_2$ (*conjunction*), $\rho_1 \vee \rho_2$ (*disjunction*), $\rho_1 \Rightarrow \rho_2$ (*implication*) and $\neg\rho_1$ (*negation*) are also (S,F)-sentences.*
- *If X is a set of variables for (S,F), then $(\forall X)\rho$ and $(\exists X)\rho$ are (S,F)-sentences whenever ρ is an $(S,F \cup X)$-sentence.*

The sentences that do not involve any quantifications are called *quantifier-free sentences*.

Definition 2.7 (Conditional equations)**.** *A* conditional equation *is a sentence of the form $(\forall X)H \Rightarrow C$ where H is a finite conjunction of (atomic) equations and C is a single (atomic) equation.*

When H is empty the respective conditional equation is usually written simply as $(\forall X)C$ and is called *unconditional equation*. The CafeOBJ notation, as other specification language notations, keeps '$(\forall X)$' in the notation of the conditional or the unconditional equations implicit by following the convention that each equation is universally quantified by exactly the variables that occur in its terms. For example the first axiom of `SIMPLE-NAT`, namely

```
eq N + (s M) = s(N + M) .
```

is just $(\forall$`N,M`$)$ `N + (s M)` $=$ `s(N + M)`.

The equational specification paradigm, as its name suggests, considers only conditional or unconditional equations as axioms for its specifications. This is also true for CafeOBJ. The main reason for such a restriction is the existence of the so-called 'initial' models or algebras for equational specifications. This property is important because when specifying data types, one often likes to specify a particular singular model or implementation, and the initiality property can characterize models uniquely (up to isomorphisms). For example this is the case of the specification `SIMPLE-NAT` when one wants to specify the standard model of the natural numbers data type. Another reason in favour of equational specifications are the existence of a relatively simple proof system with good computational properties. We will see the precise meaning of all these things later in the chapter.

Denotations and algebras. Specifications are formal descriptions of certain class of possible implementations. In algebraic specification, "implementation" is captured by the concept of *model* and "possible implementations" by the concept of *denotation*. The models of data type specifications are called *algebras*. Algebras are ideal mathematical entities interpreting the syntactic constituents of signatures of the specifications as ideal semantics entities. Thus algebras interpret:

- sorts as sets, and
- operations as functions on these sets,

such that the interpretation of the operations is compatible with the interpretation of the sorts. Software and even hardware implementations of specifications can be mathematically regarded as algebras.

Definition 2.8 (Algebras)**.** *Given a set of sort symbols S, an S-*indexed *(or* sorted*) set A is a family $\{A_s\}_{s\in S}$ of sets indexed by the elements of S; in this context, $a \in A$ means that $a \in A_s$ for some $s \in S$. Given an S-indexed set A and $w = s_1 \ldots s_n \in S^*$, we let $A_w = A_{s_1} \times \cdots \times A_{s_n}$; in particular, we let $A_{[]} = \{\star\}$, some one point set.*

*Given a signature (S,F), a (S,F)-*algebra *A consists of*

- *an S-indexed set A (the set A_s is called the* carrier *of A of sort s), and*
- *a function $A_{\sigma:w\to s} : \ A_w \to A_s$ for each $\sigma \in F_{w\to s}$.*

When there is no danger of ambiguity (because of overloading of σ) we may simplify the notation $A_{\sigma:w\to s}$ to A_σ. If $\sigma \in F_{\to s}$ then A_σ determines a point in A_s which may also be denoted A_σ.

For example, the natural numbers with succesor and addition functions are an algebra A for the signature of `SIMPLE-NAT` as follows:

- $A_{\texttt{Nat}} = \omega = \{0,1,2,\ldots\}$,
- $A_{\texttt{0}} = 0$,
- $A_{\texttt{s_}}(x) = x+1$ for each $x \in \omega$, and
- $A_{\texttt{_+_}}(x,y) = x+y$ for all $x,y \in \omega$.

But there are myriads of other models for the signature of `SIMPLE-NAT`, most of them very different from the intended model A above, such as B defined below:

- $B_{\texttt{Nat}} = \{0,1\}$,
- $B_{\texttt{0}} = 1$,
- $B_{\texttt{s_}}(x) = 1-x$ for each $x \in \omega$, and
- $B_{\texttt{_+_}}(x,y) = 1$ for all $x,y \in \omega$.

Although the algebra B may go against all our preconceptions regarding the interpretations of familiar symbols such `0` and `+`, its definition respects Dfn. 2.8 which allows us to interpret the syntactic symbols quite freely.

The satisfaction relation. The concept of satisfaction between models and sentences is the crucial link between the semantics and the syntax of formal specifications. This tells us when a certain axiom holds in a certain model. On this basis, from all models of the signature of a given specification we may isolate the models that actually verify the axioms of the specification.

The first step in defining the mathematical concept of satisfaction is to see how each term of a given signature gets evaluated as an element in any algebra of that signature. This can be defined recursively on the structure of the term.

Definition 2.9 (Interpretation of terms). *Let (S,F) be any signature. Any F-term $t = \sigma(t_1,\ldots,t_n)$, where $\sigma \in F_{w\to s}$ is an operation symbol and $t_1,\ldots,t_n$ are F-(sub)terms corresponding to the arity w, gets interpreted as an element $A_t \in A_s$ in a (S,F)-algebra A defined by*

$$A_t = A_\sigma(A_{t_1},\ldots,A_{t_n}).$$

The base case in the above definition is corresponds to the situations when $n = 0$, i.e, when σ is a constant symbol.

The satisfaction between (S,F)-algebras and (S,F)-sentences, denoted $\models_{(S,F)}$ or simply by $\models$ when there is no danger of confusion, is defined inductively on the structure of the sentences as follows.

Definition 2.10 (Satisfaction relation). *Given a fixed arbitrary signature (S,F) and a (S,F)-algebra A,*

- $A \models t = t'$ *if and only if* $A_t = A_{t'}$ *for equations,*
- $A \models \rho_1 \wedge \rho_2$ *if and only if* $A \models \rho_1$ *and* $A \models \rho_2$,
- $A \models \rho_1 \vee \rho_2$ *if and only if* $A \models \rho_1$ *or* $A \models \rho_2$,
- $A \models \rho_1 \Rightarrow \rho_2$ *if and only if* $A \not\models \rho_1$ *or* $A \models \rho_2$,
- $A \models \neg\rho_1$ *if and only if* $A \not\models \rho_1$,

for all (S,F)-sentences ρ_1 and ρ_2, and

- *for any set of variables X for the signature (S,F), and for any $(S,F\cup X)$-sentence ρ', $A \models_{(S,F)} (\forall X)\rho$ if and only if $A' \models_{(S,F\cup X)} \rho$ for each $(S,F\cup X)$-algebra A' such that $A'_s = A_s$ for each $s \in S$ and $A'_\sigma = A_\sigma$ for each operation symbol σ of F.*
- $A \models (\exists X)\rho$ *if and only if* $A \not\models (\forall X)\neg\rho$.

The $(S,F\cup X)$-algebra A' in the definition above is called the $(S,F\cup X)$-*expansion* of A and it is just A plus a function that assigns a element of A_s to each variable symbol of sort s in X. Then A is called the (S,F)-*reduct* of A'.

When $A \models \rho$ we say that A *satisfies* ρ or that ρ *holds* in A. While the algebra A introduced after Dfn. 2.8 satisfies both axioms of `SIMPLE-NAT`, the other algebra B does not satisfy any of them. We leave to the reader the task to check the validity of these two facts as applications of Dfn. 2.10.

The satisfaction relation between algebras and single sentences can be extended easily to a relation between algebras and *sets of* sentences: for any (S,F)-algebra A and any set E of (S,F)-sentences let $A \models_{(S,F)} E$ mean that $A \models_{(S,F)} \rho$ for all $\rho \in E$.

Operation attributes. Many algebraic specification languages, especially those that are directly executable, provide an alternative notation for some specific equations such as the commutativity or the associativity of operations. This notation is called 'operation attribute' and in CafeOBJ it looks like

```
op _+_ : Nat Nat -> Nat {comm}
```

which is an alternative to

```
eq M + N = N + M .
```

or

```
op _+_ : Nat Nat -> Nat {assoc}
```

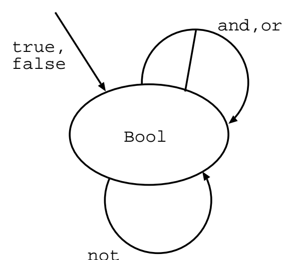
which is an alternative to

```
eq (M + N) + P = M + (N + P) .
```

The only reason for providing operation attributes notation as an alternative to the ordinary notation for equations has to do with the operational aspect of the language, and nothing to do with its specification aspect. From the semantic point of view both notations have exactly the same meaning, however the operation attributes give a signal to the system that the respective equations have to be used in a special way in the computations. This is necessary because the usual execution mechanism for equational specifications, namely rewriting, may run into an infinite loop because of commutativity equations or may get blocked because of associativity equations. Later on in the chapter we will learn more on this topic.

Booleans and predicates. There is a number of data types that are regularly used in specifications, such as numbers or truth values (also called 'Booleans'). In order to ease the task of the users, often such data types come incorporated in the specification systems as pre-defined, or system defined, types. Pre-defined types have at least two advantages: they save the users from the trouble to have to specify them, and more importantly, they may run more efficiently due to the fact that often their execution is performed by programs in some low level programming language used for the implementation of the specification system.

As mentioned before, one of the most used pre-defined data type is that of the Boolean values. In CafeOBJ it is named `BOOL` and unless specified it is automatically made available for any specification. The essential part of its signature can be represented as



where `true`, `false`, `and`, `or`, `not` have the standard well known meaning.

As an exercise let us now use the Boolean data type for defining the 'strictly less than' relation between the natural numbers. This is done in two steps. First we have to specify the symbol for the relation, which is treated as a Boolean valued function.

```
op _<_ : Nat Nat -> Bool
```

Next we write the equations that define the relation.

```
eq 0 < (s M) = true .
eq (s M) < (s N) = M < N .
```

However with only these two equations we may end up with some new values of sort `Bool`, such as `(s s 0) < (s 0)`. In order to collapse these to `false` we need one more equation.

```
eq M < 0 = false .
```

Sometimes, in order to get the notation closer to the familiar notation for relations, in CafeOBJ one may skip to write the sort `Bool` as the sort of the `Bool`-valued operation that simulates the respective relation. The short hand notation for this keeps `Bool` implicit and looks like

```
pred _<_ : Nat Nat
```

The keyword `pred` come from 'predicate' which in logic is another name for 'relation'. However, always remember that this is only a short hand alternative and that in CafeOBJ relations are treated as Boolean valued functions.

Conditions as Boolean terms. Some algebraic specification languages, especially those of the OBJ family, such as CafeOBJ and Maude, have a specific way to write the conditions of conditional equations, which is very good for running or executing the specifications. Recall that (according to Dfn. 2.7) the conditions of equations are just finite conjunctions of equations. In CafeOBJ and Maude these may be encoded as Boolean terms by encoding the conjunction $\wedge$ as the binary operation `and` of the (pre-defined) data type `BOOL` and by encoding equations $t = t'$ as terms t `==` t' of sort `Bool`. For this CafeOBJ provides implicitly for each declared sort s a Boolean valued operation

$$==_s \; : \; s\, s \; \text{->} \; \texttt{Bool}$$

For example in CafeOBJ we may define a maximum function on pairs of naturals as follows:

```
op max : Nat Nat -> Nat { comm }
ceq max(M:Nat, N:Nat) = M if (N < M) .
ceq max(M:Nat, N:Nat) = M if (M == N) .
```

Note that the two equations above have the same conclusion, and each of them corresponds to a different case for the condition. They can be written more compactly as one sentence by using the operation `or` on the sort `Bool` as follows.

```
ceq max(M:Nat, N:Nat) = M if (N < M) or (M == N) .
```

Note that this CafeOBJ code does not correspond anymore to a conditional equation since its condition is a Boolean term corresponding to a disjunction of equations. Because the pre-defined Boolean type `BOOL` has operations such as `or` and `not` also, means that we can write Boolean terms that correspond to any quantifier-free sentences. This goes considerably beyond conditions as conjunctions of equations, which means that using such Boolean terms as conditions places us beyond the logic of conditional equations. This may have a series of undesirable semantic consequences, such as inconsistency in the form of absence of any models for our specification. In short, it is advisable to write only conditions that correspond to finite conjunctions of equations. However, one exception still works well, namely using `or`. This is justified by the fact that any equation conditioned by a quantifier-free sentence envolving only conjunctions and disjunctions is equivalent semantically to a finite set or conditional equational. This means that what exactly should be avoided is the use of the negation (`not`) operation in conditions.

Exercises.

2.1. For any signature (S,F) and any sets X and Y of variables for (S,F)vsuch that $X \cap Y = \emptyset$ show that for any $(S, F \cup X \cup Y)$-sentence ρ, the sentences $(\forall X)(\forall Y)\rho$, $(\forall Y)(\forall X)\rho$, and $(\forall X \cup Y)\rho$ are *semantically equivalent*, meaning that they are satisfied by the same algebras. Hence we may not discriminate between these sentences.

2.2. Prove that any sentence of the form $(\forall X)H \Rightarrow C$ where H is a quantifier-free sentence formed from equations and $\wedge$, $\vee$, and C is an atom, is semantically equivalent to a set of conditional equations.

2.3. Write a CafeOBJ specification with equations conditioned by Boolean terms that does not have any models.

2.2 Initial Semantics

Tight versus loose denotations. There are two kinds of denotations for equational specifications, the so-called *tight*, or initial, and the so-called *loose* denotations. The difference between them corresponds to different intentions from the side of the specifier. Tight denotations are used when the intention is to specify a certain singular model, while the loose denotations are used for specifying a class of models. The choice of the kind of denotation is reflected by the CafeOBJ notation in the header of the specifications by the use of the keywords `mod!` and `mod*`, respectively.

We have already seen the example of `SIMPLE-NAT` which is specified with tight denotation (hence `mod!`) since in this case the intention is to specify the model of the natural numbers. The following example shows clearly the difference between the tight and the loose denotations. While the loose denotation specification `SEMIGROUP` specifies *all* semigroups with two designated constants, `a` and `b`, its tight denotation variant `STRG` specifies the semigroup consisting of all strings formed by two characters `a` and `b`.

```
mod! STRG {
  [ S ]
  ops a b : -> S
  op __ : S S -> S {assoc}
}
```

```
mod* SEMIGROUP {
  [ S ]
  ops a b : -> S
  op __ : S S -> S {assoc}
}
```

While in the case of SEMIGROUP the denotation may be obtained rather easily, as the class of all algebras of the signature satisfying the only specified equation, namely the associativity of the binary operation (specified as an operation attribute), to obtain the denotation of STRG requires a more complex process. In general, this process can be informally explained as consisting of two main steps:

1. We construct all well-formed terms from the signature of the specification, and
2. We identify the terms which are equal under the equations of the specification.

For our current example the first step constructs the terms such as
`a, b,`
`ab, ba, aa, bb,`
`a(ab), a(ba), b(ab), b(ba), (ab)a, (ab)b, (ba)a, (ba)b,`
`(ab)(ab), (ab)(ba), (ba)(ab), (ba)(ba), ...` etc.,
and at the second step one identifies terms under the associativity equation, which just get rids of the brackets. For example `(a(ab))b, (aa)(bb), a(a(bb)), a((ab)b), ((aa)b)b` are all identified as one element, which may be denoted as `aabb`. This model of the strings can be characterized among all other models of the signature satisfying the respective associativity equation by a special property, called *initiality*. The rest of this section is devoted to the mathematical explanation of the concept of initiality and of the process of constructing initial algebras for specifications.

Algebra homomorphisms and initial algebras. The initiality property is about how the respective algebra relates to other algebras. The mathematical concept that relates algebras between them is called of *homomorphism of algebras*.

Definition 2.11 (Homomorphism of algebras)**.** *An S-*indexed *(or* sorted*)* function $f : A \to B$ *is a family* $\{f_s : A_s \to B_s \mid s \in S\}$*. Also, for an S-sorted function $f : A \to B$, we let $f_w : A_w \to B_w$ denote the function product mapping a tuple of elements $(a_1,\dots,a_n)$ to the tuple $(f_{s_1}(a_1),\dots,f_{s_n}(a_n))$.*

An (S,F)-homomorphism *from one (S,F)-algebra A to another B is an S-indexed function $h : A \to B$ such that*

$$h_s(A_\sigma(a)) = B_\sigma(h_w(a))$$

for each $\sigma \in F_{w\to s}$ and $a \in A_w$.

Homomorphisms of algebras may also be called algebra homomorphisms, and as Dfn. 2.2 suggests they are (families of) functions that preserve the algebraic structure. When there is no danger of confusion we may simply write $h(a)$ instead of $h_s(a)$.

Like functions, homomorphisms of algebras compose.

Definition 2.12 (Composition of algebra homomorphisms)**.** *Given* (S,F)*-homomorphisms* $h: A \to B$ *and* $g: B \to C$*, their composition* $h;g$ *is the algebra homomorphism* $A \to C$ *defined by* $(h;g)_s = g_s \circ h_s$ *for each sort symbol* $s \in S$.[1]

The reader may check by herself the correctness of the definition of composition of algebra homomorphisms, namely that $h;g$ of Dfn. 2.12 is an (S,F)-homomorphism indeed.

The following special case of homomorphism captures the situation when algebras are essentially the same, called *isomorphic algebras*, in the sense that they differ only by a renaming of their elements.

Definition 2.13 (Isomorphism of algebras)**.** *A* (S,F)*-homomorphism* $h: A \to B$ *is a* (S,F)-isomorphism *when there exists another homomorphism* $h^{-1}: B \to A$ *such that* $h;h^{-1} = 1_A$ *and* $h^{-1};h = 1_B$*, where by* $1_A: A \to A$ *and* $1_B: B \to B$ *we denote the 'identity' homomorphisms that map each element to itself.*

Note that Dfn. 2.13 characterizes isomorphisms by a compositionality property rather than by a direct property of the respective homomorphism as given in the following.

Fact 2.14. *A* (S,F)*-homomorphism* $h: A \to B$ *is isomorphism if and only if each function* $h_s: A_s \to B_s$ *is bijective (i.e., one-to-one and onto, in an older terminology).*

The proof of Fact 2.14 is rather straightforward and is left as exercise to the reader.

Definition 2.15 (Initial algebras)**.** *Given any class* $\mathbb{C}$ *of* (S,F)*-algebras, an algebra* A *is* initial *for* $\mathbb{C}$ *when* $A \in \mathbb{C}$ *and for each algebra* $B \in \mathbb{C}$ *there exists an unique homomorphism* $A \to B$.

In our lecture notes we will be mainly interested in the classes $\mathbb{C}$ of the algebras satisfying certain fixed sets of conditional equations.

Initial algebras have the crucial property that they are unique up to isomorphisms, as explained by the following simple result.

Proposition 2.16. *If* A *and* A' *are both initial algebras for* $\mathbb{C}$*, then there exists an isomorphism* $A \to A'$.

Proof. Since A is initial and $A' \in \mathbb{C}$ there exists a homomorphism $h: A \to A'$. Since A' is initial and $A \in \mathbb{C}$ there exists a homomorphism $h': A' \to A$. Then $h;h'$ is a homomorphism $A \to A$. But the identity $1_A: A \to A$ is also a homomorphism, and since by initiality there exists an unique homomorphism $A \to A$, we have that $h;h' = 1_A$. Similarly $h';h = 1_{A'}$. □

Because initial algebras are isomorphic, which means that they are the same modulo renaming of elements, we usually say 'the initial *algebra*' of an equational specification instead of the more correct terminology 'the initial algebras'. This also explains why we usually refer to the unique model or algebra of tight denotations, when in fact tight denotations mean a class of (mutually isomorphic) models or algebras. For example, the denotation of `SIMPLE-NAT` consists of all representations of the natural numbers, but yet since all these representations are mutually isomorphic we speak about *a* standard model of the natural numbers. To summarize, the initiality property gives us a very general way to capture single standard models that we intend to specify.

[1] This means $(h;g)(a) = g(h(a))$ for each $a \in A_s$.

Congruences and quotients. The next goal for us is to show that each specification with conditional equations admits initial algebras. This shows that such specifications that are considered with the tight denotation are *consistent*, i.e. they have a model. Moreover, this also shows that loose denotation equational specifications are consistent, since their denotations would contain at least the initial algebra.

For showing that in general each specification with conditional equations admits initial algebras, we need to introduce the concepts of *congruence* and *quotient algebra*.

Definition 2.17 (Congruence). *A F-*congruence *on a (S,F)-algebra A is an S-sorted family of relations, $\equiv_s$ on A_s, each of which is an equivalence relation, and which also satisfy the* congruence property*, that given any $\sigma \in F_{w \to s}$ and any $a \in A_w$, then $A_\sigma(a) \equiv_s A_\sigma(a')$ whenever $a \equiv_w a'$.*[2]

For example the binary relation $\equiv$ on the model A of `SIMPLE-NAT` (introduced after Dfn. 2.8) defined by $a \equiv b$ if and only if the difference between a and b is an even number, is a congruence. Note that if we replaced 'even' by 'odd' in the definition of $\equiv$, it remains an equivalence relation but not a congruuence anymore since the congruence property fails for `+`.

Definition 2.18 (Quotient algebra). *Each congruence on an (S,F)-algebra A determines a* quotient *algebra $A/_\equiv$ such that*

- *$(A/_\equiv)_s = (A_s)/_{\equiv_s}$ for each sort $s \in S$, i.e. $(A_s)/_{\equiv_s} = \{a/_\equiv \mid a \in A_s\}$ is the set of the equivalence classes of $\equiv_s$, and*
- *$(A/_\equiv)_\sigma(a_1/_\equiv \ldots a_n/_\equiv) = A_\sigma(a_1,\ldots,a_n)/_\equiv$ for each operation symbol $\sigma \in F_{s_1 \ldots s_n \to s}$ and each $(a_1,\ldots,a_n) \in A_{s_1} \times \cdots \times A_{s_n}$.*

The canonical (S,F)-homomorphism $q : A \to A/_\equiv$ mapping each element a to its equivalence class $a/_\equiv$ is called the quotient homomorphism *associated to $\equiv$.*

The second item of Dfn. 2.18 makes sense because of the congruence property, in that if we chose other representatives $a'_1,\ldots,a'_n$ for the equivalence classes $a_1/_\equiv,\ldots,a_n/_\equiv$ it would be no trouble since $A_\sigma(a_1,\ldots,a_n) \equiv A_\sigma(a'_1,\ldots,a'_n)$.

The quotient algebra of the congruence $\equiv$ on the algebra A of `SIMPLE-NAT` discussed above consists of only two elements, the sets of the odd numbers (represented and denoted by 1) and of the even numbers (represented and denoted by 0), respectively, and $(A/_\equiv)_{\mathtt{0}} = 0$, $(A/_\equiv)_{\mathtt{s}}(0) = 1$, $(A/_\equiv)_{\mathtt{s}}(1) = 0$, $(A/_\equiv)_+(0,0) = (A/_\equiv)_+(1,1) = 0$, and $(A/_\equiv)_+(0,1) = (A/_\equiv)_+(1,0) = 1$.

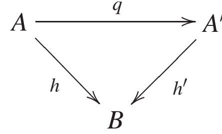
Definition 2.19 (Kernel of homomorphism). *For $h : A \to B$ any (S,F)-homomorphism let its* kernel *$=_h$ be the S-sorted family of binary relations defined by $a(=_h)_s b$ if and only if $h_s(a) = h_s(b)$.*

Fact 2.20. *For any (S,F)-homomorphism h, its kernel $=_h$ is a F-congruence.*

The following technical result will be used later in the process of proving the existence of initial algebras of equational specifications.

[2] Meaning $a_i \equiv_{s_i} a'_i$ for $i = 1,\ldots,n$, where $w = s_1 \ldots s_n$ and $a = (a_1,\ldots,a_n)$.

Proposition 2.21. *For any surjective (i.e. onto) (S,F)-homomorphism $q: A \to A'$ and any (S,F)-homomorphism $h': A \to B$, there exists an unique (S,F)-homomorphism $h': A' \to B$ such that $q;h' = h$ if and only if $=_q \subseteq =_h$.*



Proof. As the direct implication is almost trivial, we focus on the inverse implication. For each $a' \in A'$ let us define $h'(a') = h(a)$ where $a' = q(a)$. Such element a exists because q is surjective. The definition of h' is also correct, in the sense of being independent of the choice of a such that $a' = q(a)$, because $=_q \subseteq =_h$.

Now let us check the homomorphism condition for h'. Consider any operation symbol σ and any appropriate list of arguments $(a'_1,\dots,a'_n)$ for A'_σ. Then we have the following

$$\begin{array}{rcll} h'(A'_\sigma(a'_1,\dots,a'_n)) & = & (A'_\sigma(q(a_1),\dots,q(a_n)) & \text{for } a_k \in A \text{ such that } a'_k = q(a_k) \\ & = & h'(q(A_\sigma(a_1,\dots,a_n)) & \text{(because } q \text{ is homomorphism)} \\ & = & h(A_\sigma(a_1,\dots,a_n)) & \text{(because } h = q;h'\text{)} \\ & = & B_\sigma(h(a_1),\dots,h(a_n)) & \text{(because } h \text{ is homomorphism)} \\ & = & B_\sigma(h'(a'_1),\dots,h'(a'_n)) & \text{(by the definition of } h'\text{)} \end{array}$$

□

Free algebras. We are now ready to take the most important step towards our current goal, of showing the existence of initial algebras for equational specifications.

Definition 2.22 (Γ-congruence)**.** *For any conjunction $H = (t_1 = t'_1) \wedge (t_2 = t'_2) \wedge \cdots \wedge (t_n = t'_n)$ of (S,F)-equations and any (S,F)-algebra A, by A_H let us abbreviate the set $\{(A_{t_i}, A_{t'_i}) \mid 1 \le i \le n\}$.*

Given Γ a set of conditional equations in a signature (S,F), a congruence $\equiv$ on a (S,F)-algebra A is a Γ-congruence *if for each conditional equation $(\forall X)H \Rightarrow C$ in Γ and for any expansion A' of A to $(S,F \cup X)$, $A'_H \subseteq \equiv$ implies $A'_C \subseteq \equiv$.*

Proposition 2.23. *For each (S,F)-homomorphism $h: A \to B$ and each set Γ of conditional equations for (S,F), if $B \models \Gamma$ then $=_h$ is a Γ-congruence.*

Proof. Let $(\forall X)H \Rightarrow C$ in Γ and consider a $(S,F \cup X)$-expansion A' of A such that $A'_H \subseteq =_h$. Let us define a $(S,F \cup X)$-expansion B' of B by $B'_x = h(A'_x)$ for each $x \in X$. Note that h becomes a $(S,F \cup X)$-homomorphism $A' \to B'$. By induction on the structure of terms it is easy to see that $h(A'_t) = B'_t$ for each $(S,F \cup X)$-term t. By using this property we have that $A'_H \subseteq =_h$ is equivalent to $B' \models H$. Thus $B' \models H$. Because $B \models (\forall X)H \Rightarrow C$ it follows that $B' \models C$. But by the same property $h(A'_t) = B'_t$ above, $A'_C \subseteq =_h$ is equivalent to $B' \models C$. Hence $A'_C \subseteq =_h$. □

Proposition 2.24. *For each (S,F)-algebra A, any congruence $\equiv$ on A, and any set Γ of conditional equations in (S,F), $A/_{\equiv} \models \Gamma$ if and only if $\equiv$ is a Γ-congruence.*

Proof. The implication from the left to the right follows from Prop. 2.23 by considering the quotient homomorphism $A \to A/_{\equiv}$ in the role of h.

For showing the implication from the right to the left let us consider $(\forall X)H \Rightarrow C$ in Γ and any $(S,F\cup X)$-expansion A'' of $A/_{\equiv}$ such that $A'' \models H$. Let A' be an expansion of A such that for each $x \in X$ we have $A'_x \in A''_x$. By induction on the structure of any $(S,F\cup X)$-term t, it is easy to see that $(A'_t)/_{\equiv} = A''_t$. Therefore $A'' \models H$ means that $A'_H \subseteq \equiv$. Because $\equiv$ is a Γ-congruence we obtain $A'_C \in \equiv$ which shows in turn means $A'' \models C$, which shows that $A/_{\equiv} \models (\forall X)H \Rightarrow C$. □

Note that given any (S,F)-algebra A and any set Γ of conditional equations in (S,F), the intersection of any family of Γ-congruences is still a Γ-congruence. This implies that there exists the least Γ-congruence on A obtained as the intersection of *all* Γ-congruences on A.

Notation 2.25. *Let $=^A_\Gamma$ denote the least Γ-congruence on an algebra A and q^A_Γ denote its associated quotient homomorphism.*

Corollary 2.26 (Free algebras)**.** *For any (S,F)-algebra A, $A/_{=^A_\Gamma}$ is the* free algebra over *A satisfying Γ,*

$$\begin{array}{ccc} A & \xrightarrow{q^A_\Gamma} & A/_{=^A_\Gamma} \\ & {\scriptstyle\forall h}\searrow \quad \swarrow{\scriptstyle\exists! h_\Gamma} & \\ & B \models \Gamma & \end{array}$$

in the sense that for any other algebra B satisfying Γ and any (S,F)-homomorphism $h\colon A \to B$ there exists an unique (S,F)-homomorphism $h_\Gamma\colon A/_{=^A_\Gamma} \to B$ such that $q^A_\Gamma; h_\Gamma = h$. (This is called the universal property *of the quotient homomorphism q^A_Γ.)*

Proof. $A/_{=^A_\Gamma} \models \Gamma$ by Prop. 2.24. The universal property of the quotient homomorphism q^A_Γ follows directly from Prop. 2.21 by noting that $=_h$ is a Γ-congruence (cf. Prop. 2.23) and because $=^A_\Gamma$ being the least Γ-congruence is smaller than $=_h$. □

Term algebras. For each signature, the terms can be organized as an algebra which is initial.

Proposition 2.27 (Initial term algebras)**.** *For any signature (S,F), let $0_{(S,F)}$ be the* term algebra *defined as follows:*

- *$(0_{(S,F)})_s$ is the set $(T_{(S,F)})_s$ of all (S,F)-terms of sort s, and*
- *$(0_{(S,F)})_\sigma(t_1,\ldots,t_n) = \sigma(t_1,\ldots,t_n)$ for each operation symbol $\sigma \in F_{w\to s}$ and each list of terms $t_1,\ldots,t_n$ corresponding to w.*

Then $0_{(S,F)}$ *is initial in the class of all* (S,F)*-algebras.*

Proof. By induction on the structure of a term t we can show that for any (S,F)-algebra B the unique homomorphism $h\colon\ 0_{(S,F)} \to B$ can be defined only as $h(t) = B_t$. The base of this induction just means the homomorphism property of h for the constants of the signature. For the induction step let $t = \sigma(t_1,\dots,t_n)$ where σ is an operation symbol and $t_1,\dots,t_n$ are the immediate subterms of t. Then $h(t) = h(\sigma(t_1,\dots,t_n)) = h((0_{(S,F)})_\sigma(t_1,\dots,t_n)) = B_\sigma(h(t_1),\dots,h(t_n)) = B_\sigma(B_{t_1},\dots,B_{t_n})$ (by induction hypothesis) $= B_{\sigma(t_1,\dots,t_n)} = B_t$. □

Existence of initial algebras for conditional equational specifications. That the class of algebras satisfying any fixed set of conditional equations admits an initial algebra can be obtained as an instance of the existence of free algebras (Cor. 2.26).

Corollary 2.28. *Each set* Γ *of conditional equations admits an initial algebra denoted* 0_Γ *and which is* $0_{(S,F)}/_{=_\Gamma}$*, the free algebra over the term algebra* $0_{(S,F)}$ *satisfying* Γ.

Proof. Let B be any algebra satisfying Γ and according to Prop. 2.27 let h be the unique (S,F)-homomorphism $0_{(S,F)} \to B$. Then by Cor. 2.26 there exists an unique (S,F)-homomorphism $h'\colon\ 0_{(S,F)}/_{=_\Gamma} \to B$ such that $q_\Gamma;h' = h$.

$$\begin{array}{ccc} 0_{(S,F)} & \xrightarrow{q_\Gamma} & 0_{(S,F)}/_{=_\Gamma} \\ & {}_{h}\searrow \quad \swarrow_{h'} & \\ & B \models \Gamma & \end{array}$$

Moreover h' is unique simply as a homomorphism $0_{(S,F)}/_{=_\Gamma} \to B$ because for any other such homomorphism h'', because there exists only one homomorphism $0_{(S,F)} \to B$ we have that $q_\Gamma;h' = q_\Gamma;h'' = h$. By the uniqueness property of h' we obtain that $h' = h''$. □

Let us now reflect on the general process underlying the existence of initial algebras of conditional equational specifications we have just completed by looking back at the example of the initial algebra of `STRG`. The first step, that above was informally described as the construction of the terms of the signature corresponds the existence of the initial algebra in the class of all algebras of the signature. The second step, described as identifying the terms that are equal under the equations of the specification corresponds to the construction of the free algebra over the term algebra, satisfying the equations of the specification.

Exercises.

2.4. Do the proof of some facts about homomorphisms of algebras have been skipped in the text.

1. The composition of homomorphisms (Dfn. 2.12) is a homomorphism indeed.
2. The composition of homomorphisms is associative.

3. An (S,F)-homomorphism h is isomorphism if and only if for each sort symbol $s \in S$, h_s is bijective.

2.5. Let (S,F) be any signature and ρ a conditional equation for (S,F).

1. For any (S,F)-algebras A and B let $A \times B$ be their *direct product* defined by
   - $(A \times B)_s = A_s \times B_s = \{\langle a, b\rangle \mid a \in A_s, b \in B_s\}$ for each sort $s \in S$,
   - $(A \times B)_\sigma(\langle a_1, b_1\rangle, \ldots, \langle a_n, b_n\rangle) = \langle A_\sigma(a_1, \ldots, a_n), B_\sigma(b_1, \ldots, b_n)\rangle$ for each operation symbol $\sigma \in F_{w \to s}$.

   Prove that $A \times B \models \rho$ if $A \models \rho$ and $B \models \rho$.
2. Let $h: A \to B$ be an injective (S,F)-homomorphism. Prove that $A \models \rho$ if $B \models \rho$. Extend this result to the case when ρ is of the form $(\forall X)\rho_0$ where ρ_0 is a quantifier-free sentence.
3. Let $h: A \to B$ be a surjective homomorphism. Prove that $B \models \rho$ if $A \models \rho$ and ρ is an unconditional equation. Give a (counter)example showing that in general this property fails for conditional equations.

2.6. Consider a signature with one sort and constants `a`, `b`, `c`, and `d`. Show that the sentence

$$(\neg(\mathtt{a} = \mathtt{b})) \Rightarrow (\mathtt{c} = \mathtt{d})$$

does not have initial algebras.

2.7. Show that the algebra A introduced after Dfn. 2.8 and consisting of the natural numbers with the common interpretation of the constant `0`, of the successor function `s_`, and of the addition operation `_+_`, is a model of `SIMPLE-NAT`, i.e. is the initial algebra satisfying the axioms of `SIMPLE-NAT`.

2.8. Extend the specification `SIMPLE-NAT` with the specification of the multiplication function on naturals `_*_` and prove that the expansion of algebra A of Ex. 2.7 with the common interpretation of the multiplication symbol is indeed initial algebra for the resulting specification.

2.9. Define an algebra satisfying the axioms of `SIMPLE-NAT` that has strings of natural numbers as elements, `0` is interpreted as the empty list and `_+_` is interpreted as string concatenation.

2.10. Consider the following specification.

```
mod! BASIC-INT {
  [ Int ]
  op 0 : -> Int
  op s_ : Int -> Int
  op -_ : Int -> Int
  var X : Int
  eq - 0 = 0 .
  eq - - X = X .
  eq s(-(s X)) = - X .
}
```

1. Show that the set of the integer numbers together with the standard interpretation of `0` and of `-` as zero and unary minus, respectively, and of `s_` as addition with 1, is the model of `BASIC-INT`.

2. Prove that for any non-trivial congruence $\equiv$ on the algebra introduced at the item above there exists a natural number n such that for any integer numbers x and y we have that $x \equiv y$ if and only if there exists an integer number z such that $x - y = n * z$.

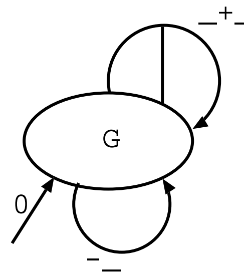
2.3 Equational Deduction

We are all familiar from school algebra with the basic principle of replacing equals by equals when manipulating algebraic expressions. The deduction system of ordinary or of general algebra is called 'equational deduction'. Since data type specification is based upon general algebra, its formal verification aspect is based upon equational deduction. This section is devoted to the formal introduction of the equational deduction and to its most important aspects:

- *soundness*, meaning the validity of the deductions, and
- *completeness*, meaning that the deduction system has the power to prove whatever is valid semantically.

Group theory. Groups are perhaps the most fundamental structure of ordinary algebra, thus many ordinary arithmetical and algebraic calculation being closely related to the deduction system of group theory. The class of groups can be specified as follows:

```
mod* GROUP {
  [ G ]
  op 0 : -> G
  op _+_ : G G -> G {assoc}
  op -_ : G -> G
  var X : G
  eq 0 + X = X .
  eq (- X) + X = 0 .
}
```



That the denotation of GROUP consists of all groups may not be obvious from the specification since the usual definition of groups contains two more equations:

```
eq X + 0 = X .
eq X + (- X) = 0 .
```

The point is that the above two equations can be *deduced* from the three axioms of the specification (two explicit equations plus the associativity attribute for _+_) just by ordinary syntactic manipulation of the respective expressions. In the following let us see the details of these deductions.

Proving `X + (- X) = 0`. The second equation can be deduced by the following sequence of replacements of subterms of expressions by corresponding terms that are 'equal' according to the axioms of GROUP:

(1) $(-(-a)) + (-a) = 0$ by the 2nd axiom for `X` substituted by $(-a)$,

(2) $(-(-a)) + (0 + (-a)) = 0$ by the 1st axiom for `X` substituted by $(-a)$,

(3) $(-(-a)) + (((-a) + a) + (-a)) = 0$ by the 2nd axiom for `X` substituted by a,

(4) $((-(-a)) + ((-a) + a)) + (-a) = 0$ by the associativity of $_ + _$,

(5) $(((-(-a)) + (-a)) + a) + (-a) = 0$ by the associativity of $_ + _$,

(6) $((-(-a)) + (-a)) + (a + (-a)) = 0$ by the associativity of $_ + _$,

(7) $0 + (a + (-a)) = 0$ by the 2nd axiom for `X` substituted by $(-a)$,

(8) $a + (-a) = 0$ by the 1st axiom for `X` substituted by $a + (-a)$.

Proving `X + 0 = X`. Similarly, the first equation can be deduced from the axioms of `GROUP` and from the second equation, that has been already deduced above.

(1) $0 + a = a$ by the 1st axiom for `X` substituted by a,

(2) $(a + (-a)) + a = a$ by the 2nd equation for `X` substituted by a,

(3) $a + ((-a) + a) = a$ by the associativity of $_ + _$, and

(4) $a + 0 = a$ by the 2nd axiom for `X` substituted by a.

Substitutions. The deduction steps in the proofs presented above of the two missing group theory equations consist of replacements in terms of an *instance* of a side of a specification axiom or of an already proved equation by the *corresponding instance* of the other side of the equation.

- An 'instance' of a term means the terms resulting from the replacement of some of its variables by terms of the same sort with the variables to be replaced. Such mapping of variables to terms is called *substitution* (of variables).
- By 'corresponding instance' we mean that both the replaced subterm and the replacement term use the same substitution of the variables.

  We now give the mathematical definition for the concept of substitution.

Definition 2.29 (Substitution). *Given sets X and Y of variables for a signature (S,F), an* (S,F)-substitution θ *from X to Y is is a function* $\theta : X \to T_{(S,F\cup Y)}$ *that respects sorts, i.e. if x has sort s then* $\theta(x) \in (T_{(S,F\cup Y)})_s$.

The existence of substitutions from X to Y requires that whenever there is a variable in X of sort s then $(T_{(S,F\cup Y)})_s$ is non-empty. In general, this condition can be met if we assume that the signatures contain at least one constant for each sort.

Any substitution $\theta : X \to T_{(S,F\cup Y)}$ extends to a function $\theta^\sharp : T_{(S,F\cup X)} \to T_{(S,F\cup Y)}$ defined by

$$\theta^\sharp(t) = \begin{cases} \theta(x) & \text{when } t = x \text{ for } x \in X \\ \sigma(\theta^\sharp(t_1),\dots,\theta^\sharp(t_n)) & \text{when } t = \sigma(t_1,\dots,t_n) \text{ with } \sigma \in F. \end{cases}$$

When there is no danger of notational confusion we may omit '$\sharp$' from the notation and write simply $\theta(t)$ instead of $\theta^\sharp(t)$.

The application of substitutions may be extended to $(S, F \cup X)$-sentences. Informally this is just the replacement of variables in the sentence by their corresponding terms. Formally this is defined as follows:

- $\theta(t_1 = t_2)$ stands for $\theta^\sharp(t_1) = \theta^\sharp(t_2)$,
- $\theta(\rho_1 \wedge \rho_2)$ stands for $\theta(\rho_1) \wedge \theta(\rho_1)$, and similarly for $\vee$, $\Rightarrow$, $\neg$, and
- $\theta((\forall Z)\rho)$ stands for $(\forall Z)\theta(\rho)$. In this case it is implicitly assumed that both X and Y are disjoint from Z.

Entailment systems. In order to formalize the deduction system for conditional equations we need to understand the general properties of deduction. These are captured by the mathematical concept of entailment system.

Definition 2.30 (Entailment system)**.** *Given a signature Σ, an* entailment relation *consists of a binary relation $\vdash_\Sigma$ between sets of Σ-sentences such that the following properties hold:*

1. union: *if $\Gamma \vdash_\Sigma \Gamma_1$ and $\Gamma \vdash_\Sigma \Gamma_2$ then $\Gamma \vdash_\Sigma \Gamma_1 \cup \Gamma_2$,*
2. monotonicity: *if $\Gamma' \supseteq \Gamma$ then $\Gamma' \vdash_\Sigma \Gamma$, and*
3. transitivity: *if $\Gamma \vdash_\Sigma \Gamma_1$ and $\Gamma_1 \vdash_\Sigma \Gamma_2$ then $\Gamma \vdash_\Sigma \Gamma_2$.*

An entailment system $\vdash$ *consists of an entailment relation $\vdash_\Sigma$ for each signature Σ.*

As a matter of terminology, $\Gamma \vdash \Gamma'$ reads Γ *entails* Γ' and each such pair of the entailment relation is simply called *an entailment*.

Note the high abstraction level of this definition which not only allows a multitude of entailment systems for conditional equations, considered as the sentences, but can also be applied to various other logical systems. The latter observation is based upon the fact that the properties of entailment relations do not depend upon the specific concepts of signature and sentence we have introduced, in fact they do not depend upon *any* specific concepts of signature and sentence.

The following is an important example of entailment system.

Proposition 2.31 (Semantic entailment)**.** *For any signature (S,F), the relation $\models_{(S,F)}$ between finite sets of conditional equations defined by*

$$E \models_{(S,F)} E' \text{ if and only if for any } (S,F)\text{-algebra } A,\ A \models_{(S,F)} E \text{ implies } A \models_{(S,F)} E'.$$

is an entailment relation.

Since the proof of this proposition is rather immediate we omit it here. The semantic entailment system is denoted by $\models$, the same symbol as for the satisfaction relation.

The role of the general properties of entailment systems becomes transparent when we try to make the proof of $(\forall$`X`$)$`X + 0 = X` more formal by expressing it as an entailment `GROUP` $\vdash \{(\forall$`X`$)$`X + 0 = X`$\}$, where here `GROUP` also means the set $\{A_1, A_2, A_3\}$

of the three axioms of the specification GROUP. For this purpose let us denote the equation $(\forall X) X + (- X) = 0$ by E_2 and write the fact that E_2 was proved from GROUP (proof informally presented above) by

$$\text{GROUP} \vdash \{E_2\} \tag{2.1}$$

Also let us denote the four succesive equations that occur in the proof of $(\forall X) X + 0 = X$ above by ρ_1, ρ_2, ρ_3 and ρ_4, respectively. Then the steps of the proof of $(\forall X) X + 0 = X$ (whose role is played by ρ_4 under the above convention; we shall clarify later how a proof of ρ_4 leads to a proof of $(\forall X) X + 0 = X$) correspond to the following sequence of four entailments.

$$\{A_2\} \vdash \{\rho_1\} \tag{2.2}$$

$$\{\rho_1, E_2\} \vdash \{\rho_2\} \tag{2.3}$$

$$\{\rho_2, A_1\} \vdash \{\rho_3\} \tag{2.4}$$

$$\{\rho_3, A_3\} \vdash \{\rho_4\} \tag{2.5}$$

From these four entailments plus the entailment lemma 2.1, by applying the general properties of entailment relations we obtain the following entailments:

$$\text{GROUP} \vdash \{A_2\} \quad \text{by 'monotonicity'} \tag{2.6}$$

$$\text{GROUP} \vdash \{\rho_1\} \quad \text{from 2.6 and 2.2 by 'transitivity'} \tag{2.7}$$

$$\text{GROUP} \vdash \{\rho_1, E_2\} \quad \text{from 2.1 and 2.7 by 'union'} \tag{2.8}$$

$$\text{GROUP} \vdash \{\rho_2\} \quad \text{from 2.8 and 2.3 by 'transitivity'} \tag{2.9}$$

$$\text{GROUP} \vdash \text{GROUP} \quad \text{by 'monotonicity'} \tag{2.10}$$

$$\text{GROUP} \vdash \text{GROUP} \cup \{\rho_2\} \quad \text{from 2.9 and 2.10 by 'union'} \tag{2.11}$$

$$\text{GROUP} \cup \{\rho_2\} \vdash \{\rho_2, A_1\} \quad \text{by 'monotonicity'} \tag{2.12}$$

$$\text{GROUP} \vdash \{\rho_3\} \quad \text{from 2.11, 2.12 and 2.4 by 'transitivity'} \tag{2.13}$$

$$\text{GROUP} \vdash \text{GROUP} \cup \{\rho_3\} \quad \text{from 2.13 and 2.10 by 'union'} \tag{2.14}$$

$$\text{GROUP} \cup \{\rho_3\} \vdash \{\rho_3, A_3\} \quad \text{by 'monotonicity'} \tag{2.15}$$

$$\text{GROUP} \vdash \{\rho_4\} \quad \text{from 2.14, 2.15 and 2.5 by 'transitivity'} \tag{2.16}$$

Proof rules. It is very useful to be able to describe or present a certain entailment system in a finitary way. Note for example that Prop. 2.31 introduces the semantic entailment system in an infinitary way because the respective definition relies on all algebras of the signature, which are not only infinite in number, they are so many that do not even constitute a set from the point of view of formal set theory.

A standard way to introduce entailment systems in a finitary way is to generate them by a system of *proof rules* that can be presented finitely.

Definition 2.32 (Proof rule)**.** *Given a signature* Σ*, a* proof rule *is pair* (E,e) *consisting of a finite set* E *of* Σ*-sentences and a* Σ*-sentence* e*.*

It is customary to denote proof rules (E,e) by $\frac{E}{e}$. Also note that the mathematical concept of proof rule lives at the same abstraction level as that of entailment relation or system.

Any collection of proof rules R for a fixed signature generates an entailment relation by considering the least entailment relation containing R. This is obtained as the intersection of all entailment relations containing R. The existence of this intersection is given by the following simple result.

Proposition 2.33. *The intersection of any family of entailment relations is an entailment relation.*

Proof. We have to check that the intersection $\vdash$ of any family $(\vdash^i)_{i\in I}$ of entailment relations has the properties of 'union', 'monotonicity', and 'transitivity'.

Let us consider 'union'. If $\Gamma \vdash \Gamma_1$ and $\Gamma \vdash \Gamma_2$ it means that $\Gamma \vdash^i \Gamma_1$ and $\Gamma \vdash^i \Gamma_2$ for each $i \in I$. By the 'union' property for each $\vdash^i$ we have that $\Gamma \vdash^i \Gamma_1 \cup \Gamma_2$ for each $i \in I$, which means that $(\Gamma, \Gamma_1 \cup \Gamma_2) \in \vdash^i$ for each $i \in I$. Hence $(\Gamma, \Gamma_1 \cup \Gamma_2) \in \bigcap_{i\in I} \vdash^i = \vdash$, which means $\Gamma \vdash \Gamma_1 \cup \Gamma_2$. This proof can be replicated for showing the 'monotonicity' and the 'transitivity' for $\vdash$ too. □

Soundness and completeness. These two concepts lies at the core of logical analysis since they express the most important aspects of the relationship between the semantic and the proof theoretic aspects of logical systems. On the one hand we have the semantic entailment system $\models$ which defines entailment between (sets of) sentences by means of the semantic level, that of the models (or algebras) and of the satisfaction relation between these and the sentences. This is the fundamental entailment system for any logic, corresponding to the deepest concept of truth given by the respective logic, that of semantic truth. The problem with the semantic entailment system is that it has an infinitary nature. On the other hand we may define entailment systems only syntactically, completely ignoring the semantics given by the models (or algebras), by generating them from systems of proof rules. Let us call these *proof theoretic entailment systems*. Their point is to approximate as exactly as possible the semantic entailment system. The really valuable proof theoretic entailment systems are those that are finitely generated. The word 'approximate' above means that we should not have proof theoretic entailments that are not semantic entailments too; this property is called *soundness*. Ideally the proof theoretic entailments should coincide with the semantic ones; this is called *completeness*. From these two properties, which in a sense are dual to each other, the crucial one is the soundness. In its absence the respective proof theoretic entailment system is completely useless, some of its entailments corresponding to incorrect deductions. Completeness is highly desirable especially when the proof theoretic entailment system is finitely generated, since this means a fully syntactic finite presentation of semantic entailment, a very good situation for mechanising deduction. However one can live with its absence. In other words it is crucial to perform correct deductions or proofs and only desirable to be able to prove everything that is true.

Definition 2.34 (Soundness and completeness)**.** *An entailment relation* $\vdash_\Sigma$ *for a signature* Σ

- *is* sound *when for any sets of* Σ*-sentences* Γ *and* Γ', $\Gamma \vdash_\Sigma \Gamma'$ *implies* $\Gamma \models_\Sigma \Gamma'$,[3] *and*
- *is* complete *when for any set of* Σ*-sentences* Γ *and for any* Σ*-sentence* ρ, $\Gamma \models_\Sigma \{\rho\}$ *implies* $\Gamma \vdash_\Sigma \{\rho\}$.

An entailment system is sound, respectively complete, when each of its entailment relations is sound, respectively complete.

In general soundness is much easier to establish than completeness, which is a rather fortunate situation if we take into account the fact that from these two properties the soundness is the crucial one. In the case of establishing the soundness property it is very useful if the entailment system is generated by proof rules; as we will see below, in this situation it is simply enough to check soundness only for the proof rules.

Definition 2.35 (Sound proof rule)**.** *The proof rule* $\frac{E}{e}$ *is* sound *if and only if* $E \models e$.

Proposition 2.36. *The entailment relation generated by a set of sound proof rules is sound too.*

Proof. By the hypothesis each proof rule $\frac{E}{e}$, considered as the pair $(E,\{e\})$, belongs to the semantic entailment relation $\models$. Because the entailment relation $\vdash$ generated by the set proof rules is the smallest one containing the respective set of proof rules, we have that $\vdash \subseteq \models$. This means $\vdash$ is sound. □

Proof rules for equational deduction. As an example let us check the soundness of the following proof rules for conditional equations.

Definition 2.37 (Equational proof rules)**.** *Given a signature* (S,F)*, the following are the* equational *proof rules for* (S,F)*:*

Reflexivity: $\dfrac{\emptyset}{t = t}$ *for all* (S,F)*-terms* t.

Symmetry: $\dfrac{\{t = t'\}}{t' = t}$ *for all* (S,F)*-terms* t *and* t' *of the same sort.*

Transitivity: $\dfrac{\{t = t',\ t' = t''\}}{t = t''}$ *for all* (S,F)*-terms* t, t' *and* t'' *of the same sort.*

Congruence: $\dfrac{\{t_i = t_i' \mid 1 \leq i \leq n\}}{\sigma(t_1,\dots,t_n) = \sigma(t_1',\dots,t_n')}$ *for each operation symbol* $\sigma \in F_{s_1\dots s_n \to s}$ *and any terms* t_i, t_i' *of sort* s_i *for* $1 \leq i \leq n$.

Substitutivity: $\dfrac{\{(\forall X)H \Rightarrow C\}}{\{\theta(H \Rightarrow C)\}}$ *for any conditional equation* $(\forall X)H \Rightarrow C$ *for* (S,F) *and for each substitution* $\theta : X \to T_{(S,F)}$.

[3] Recall that $\Gamma \models_\Sigma \Gamma'$ means that for any model/algebra A if $A \models \Gamma$ then $A \models \Gamma'$.

Each of the items above defines an infinite set of proof rules, called a *scheme* of proof rules. Since the equational proof rules consist of five schemes, we can say that the presentation of the equational proof rules is finite.

Proposition 2.38. *The equational proof rules given by Dfn. 2.37 are sound.*

Proof. We check the soundness of each of the proof rules of Dfn. 2.37. Let A be any (S,F)-algebra.

Reflexivity: For any term t, we have that $A_t = A_t$, hence $A \models t = t$.

Symmetry: For any terms t and t' of the same sort, if $A \models t = t'$ this means $A_t = A_{t'}$. By the symmetry of equality we have $A_{t'} = A_t$, hence $A \models t' = t$.

Transitivity: For any terms t, t' and t'' of the same sort, if $A \models t = t'$ and $A \models t' = t''$ this means $A_t = A_{t'}$ and $A_{t'} = A_{t''}$. By the transitivity of equality we have $A_t = A_{t''}$ hence $A \models t = t''$.

Congruence: Assume that $A \models t_i = t'_i$ for $1 \leq i \leq n$. This means $A_{t_i} = A_{t'_i}$ for $1 \leq i \leq n$ which implies $A_\sigma(A_{t_1}, \ldots, A_{t_n}) = A_\sigma(A_{t'_1}, \ldots, A_{t'_n})$ which by the definition of evaluation of terms means $A_{\sigma(t_1,\ldots,t_n)} = A_{\sigma(t'_1,\ldots,t'_n)}$. Hence $A \models \sigma(t_1, \ldots, t_n) = \sigma(t'_1, \ldots, t'_n)$.

Substitutivity: Let A be a (S,F)-algebra such that $A \models (\forall X)H \Rightarrow C$ for some conditional equation $(\forall X)H \Rightarrow C$ and for some substitution $\theta : X \to T_{(S,F)}$. We have to prove that $A \models \theta(H \Rightarrow C) = \theta(H) \Rightarrow \theta(C)$. Let A' be the $(S, F \cup X)$-expansion of A defined by $A'_x = A_{\theta(x)}$ for each variable $x \in X$. We use the following lemma.

Lemma 2.39. $A'_t = A_{\theta(t)}$ *for each* $(S, F \cup X)$*-term* t.

By Lemma 2.39 we have that $A_{\theta(H)} = A'_H$ and $A_{\theta(C)} = A'_C$. But $A \models_{(S,F)} \theta(H)$ means $A_{\theta(H)} \subseteq \{(a,a) \mid a \in A\}$ which implies $A'_H \subseteq \{(a,a) \mid a \in A\}$ which means $A' \models_{(S,F\cup X)} H$. Because A' is a $(S, F \cup X)$-expansion of A we have that $A' \models H \Rightarrow C$, hence $A' \models_{(S,F\cup X)} C$, which means $A'_C \subseteq \{(a,a) \mid a \in A\}$. Hence $A_{\theta(C)} \subseteq \{(a,a) \mid a \in A\}$ which means $A \models_{(S,F)} \theta(C)$.

We still owe the proof of Lemma 2.39, which completes the proof of the soundness of Substitutivity.

Proof of Lemma 2.39: By induction on the structure of t. Let $t = \sigma(t_1, \ldots, t_n)$ where $\sigma \in (F \cup X)_{w \to s}$ and $t_1, \ldots, t_n$ are the immediate subterms of t. We distinguish two cases:

1. If $\sigma \in X$ then $A_{\theta(t)} = A_{\theta(x)} = A'_x = A'_t$.

2. If $\sigma \in F$ then

$$\begin{aligned}
A_{\theta(t)} &= A_{\theta(\sigma(t_1,\ldots,t_n))} && = A_\sigma(A_{\theta(t_1)}, \ldots, A_{\theta(t_n)}) = \\
&= A_\sigma(A'_{t_1}, \ldots, A'_{t_n}) && \text{(by the induction hypothesis)} \\
&= A'_\sigma(A'_{t_1}, \ldots, A'_{t_n}) && \text{(since } A_\sigma = A'_\sigma\text{)} \\
&= A'_{\sigma(t_1,\ldots,t_n)} = A'_t.
\end{aligned}$$

□

In some situations, including the case of the proof theoretic entailment system for conditional equations studied here, it is not enough to generate the respective entailment system only from proof rules, another kind of rules being needed. These other kind of

rules are not proof rules in the sense of Dfn. 2.32, they are rather properties of the respective entailment system. Hence we call them 'meta-rules'. Our proof theoretic entailment system for conditional equations requires two such meta-rules which we discuss in the following.

The meta-rule of Implication. This meta-rule is better known in logic under the name of 'modus ponens'.

Definition 2.40 (Implication meta-rule)**.** *Given a signature (S,F), an entailment relation $\vdash_{(S,F)}$ for the conditional equations in (S,F) satisfies the* meta-rule of Implication *when for each set Γ of conditional equations in (S,F) and for each quantifier-free conditional equation $H \Rightarrow C$:*

$$\Gamma \vdash_{(S,F)} (H \Rightarrow C) \text{ if and only if } \Gamma \cup H \vdash_{(S,F)} C.$$

An entailment system satisfies the meta-rule of Implication *when each of its entailment relations satisfy it.*

Proposition 2.41. *The semantic entailment system $\models$ satisfies the* Implication *meta-rule.*

Proof. The Implication meta-rule for $\models$ means that for each signature (S,F)

$$\Gamma \models_{(S,F)} H \Rightarrow C \text{ if and only if } \Gamma \cup H \models_{(S,F)} C$$

for each set of conditional equations Γ and each quantifier-free conditional equation $H \Rightarrow C$ in (S,F). But $\Gamma \models_{(S,F)} H \Rightarrow C$ means that for any (S,F)-algebra M, if $M \models \Gamma$ then $M \models H \Rightarrow C$, which means that if $M \models \Gamma$ and $M \models H$ then $M \models C$, which means that if $M \models \Gamma \cup H$ then $M \models C$. □

The meta-rule of Universal Quantification. In logic one may find this meta-rule also under the name of the rule of 'generalization'.

Definition 2.42 (Universal Quantification meta-rule)**.** *An entailment system $\vdash$ for conditional equations satisfies the* meta-rule of Universal Quantification *when for each set Γ of conditional equations in a signature (S,F), for each set X of variables for (S,F), and for each conditional equation ρ in the signature $(S,F \cup X)$:*

$$\Gamma \vdash_{(S,F)} (\forall X)\rho \text{ if and only if } \Gamma \vdash_{(S,F\cup X)} \rho.$$

Proposition 2.43. *The semantic entailment system satisfies the* Universal Quantification *meta-rule.*

Proof. The *Universal Quantification* meta-rule for $\models$ means that for each signature (S,F) and for each set X of variables for (S,F)

$$\Gamma \models_{(S,F)} (\forall X)\rho \text{ if and only if } \Gamma \models_{(S,F\cup X)} \rho$$

for each set Γ of conditional equations for (S,F) and for each conditional equation ρ for $(S,F \cup X)$. Let us assume $\Gamma \models_{(S,F)} (\forall X)\rho$. For any $(S,F \cup X)$-algebra M' that satisfies Γ we consider its (S,F)-reduct M. We use the following lemma:

Lemma 2.44 (Satisfaction condition). *For each conditional equation* γ *for* (S,F) *and each* $(S,F\cup X)$*-algebra* M' *we have that*

$$M' \models_{(S,F\cup X)} \gamma \text{ if and only if } M \models_{(S,F)} \gamma$$

where M *denotes the* (S,F)*-reduct of* M'.

By Lemma 2.44 we have that $M \models_{(S,F)} \Gamma$, hence $M \models_{(S,F)} (\forall X)\rho$. Because M' is an $(S,F\cup X)$-expansion of M we have that $M' \models_{(S,F\cup X)} \rho$.
Now let us assume that $\Gamma \models_{(S,F\cup X)} \rho$ and prove that $\Gamma \models_{(S,F)} (\forall X)\rho$. Let M be any (S,F)-algebra such that $M \models_{(S,F)} \Gamma$ and let M' be any $(S,F\cup X)$-expansion of M. By Lemma 2.44 we have that $M' \models_{(S,F\cup X)} \rho$, hence $M \models (\forall X)\rho$ because M' was considered an arbitrary expansion of M. The following completes the proof of our proposition.

Proof of Lemma 2.44: Let γ be $(\forall Y)H \Rightarrow C$. In order to avoid an artificial clash of variables we may assume that X and Y are disjoint. $M' \models_{(S,F\cup X)} \gamma$ means that $M'_1 \models_{(S,F\cup X\cup Y)} H \Rightarrow C$ for any $(S,F\cup X\cup Y)$-expansion M'_1 of M' while $M \models_{(S,F)} \gamma$ means that $M_1 \models_{(S,F\cup Y)} H \Rightarrow C$ for any $(S,F\cup Y)$-expansion M_1 of M. Note that any M'_1 determines an M_1 by considering the $(S,F\cup Y)$-reduct and any M_1 determines an M'_1 by considering the $(S,F\cup X\cup Y)$-expansion defined by $(M'_1)_x = M'_x$ for each $x \in X$. The conclusion of our lemma follows from the relation

$$M'_1 \models_{(S,F\cup X\cup Y)} H \Rightarrow C \text{ if and only if } M_1 \models_{(S,F\cup Y)} H \Rightarrow C.$$

That the above relation equivalence holds can be seen easily from the fact that, because M'_1 is an $(S,F\cup X\cup Y)$-expansion of M_1, $(M'_1)_t = (M_1)_t$ for each $(S,F\cup Y)$-term t (which can be shown immediately by induction on the structure of t). □

Compactness. One of the most important properties of entailment systems is that any set of finite conclusions can be derived from a finite set of premises.

Definition 2.45 (Compact entailment). *An entailment system* $\vdash$ *is* compact *when for each signature* Σ*, for any* $E \vdash_\Sigma \Gamma$ *with* Γ *finite there exists* $E_0 \subseteq E$ *finite such that* $E_0 \vdash \Gamma$.

Proposition 2.46. *Let* $\vdash$ *be an entailment system for conditional equations that fulfills the meta-rules of Implication and Universal Quantification. Then the system of relations* $\vdash^c$ *on sets of sentences defined as follows*

$$\Gamma \vdash^c_\Sigma E \text{ if and only if for each finite } E_0 \subseteq E \text{ there exists a finite } \Gamma_0 \subseteq \Gamma \text{ such that } \Gamma_0 \vdash_\Sigma E_0$$

is an entailment system that fulfills the meta-rules of Implication *and* Universal Quantification *too.*

Proof. We have to prove five properties for $\vdash^c$ as follows.
Union: We assume $\Gamma \vdash^c E$ and $\Gamma \vdash^c E'$ and have to show that $\Gamma \vdash^c E \cup E'$. Any finite subset of $E \cup E'$ can be written as $E_0 \cup E'_0$ with $E_0 \subseteq E$ and $E'_0 \subseteq E'$ both finite. Then there exists Γ_0 and Γ'_0 both finite subsets of Γ such that $\Gamma_0 \vdash E_0$ and $\Gamma'_0 \vdash E'_0$. By 'Monotonicity' and 'Transitivity' of $\vdash$ we have that $\Gamma_0 \cup \Gamma'_0 \vdash E_0$ and $\Gamma_0 \cup \Gamma'_0 \vdash E'_0$ and by 'Union' of $\vdash$ we further deduce that $\Gamma_0 \cup \Gamma'_0 \vdash E_0 \cup E'_0$.

Monotonicity: If $\Gamma' \supseteq \Gamma \vdash^c E$ and $E_0 \subseteq E$ finite then let $\Gamma_0 \subseteq \Gamma$ finite such that $\Gamma_0 \vdash E_0$. Since $\Gamma_0 \subseteq \Gamma'$ we may deduce that $\Gamma' \vdash^c E$.
Transitivity: Assume $\Gamma \vdash^c E \vdash^c E'$. For any $E'_0 \subseteq E'$ finite there exists $E_0 \subseteq E$ finite such that $E_0 \vdash E'$. Now for E_0 there exists $\Gamma_0 \subseteq \Gamma$ finite such that $\Gamma_0 \vdash E_0$. By 'Transitivity' for $\vdash$ we have that $\Gamma_0 \vdash E'_0$.
Implication: Assume $\Gamma \vdash^c H \Rightarrow C$. Then let $\Gamma_0 \subseteq \Gamma$ finite such that $\Gamma_0 \vdash H \Rightarrow C$. By Implication of $\vdash$ we have that $\Gamma_0 \cup H \vdash C$. Since H is finite it follows that $\Gamma \cup H \vdash^c C$. Now we assume the opposite, that $\Gamma \cup H \vdash^c C$. Then let $\Gamma' \subseteq \Gamma \cup H$ finite such that $\Gamma' \vdash C$. Since H is finite we may assume with any loss of generality that $\Gamma' = \Gamma_0 \cup H$. By Implication for $\vdash$ we obtain that $\Gamma_0 \vdash H \Rightarrow C$, hence $\Gamma \vdash^c H \Rightarrow C$.
Universal Quantification: On the one hand we have that $\Gamma \vdash^c_{(S,F)} (\forall X)\rho$ if and only if there exists $\Gamma_0 \subseteq \Gamma$ finite such that $\Gamma_0 \vdash_{(S,F)} (\forall X)\rho$. On the other hand we have that $\Gamma \vdash_{(S,F\cup X)} \rho$ if and only if there exists $\Gamma_0 \subseteq \Gamma$ finite such that $\Gamma_0 \vdash_{(S,F\cup X)} \rho$. By Universal Quantification for $\vdash$ we deduce that $\Gamma \vdash^c_{(S,F)} (\forall X)\rho$ if and only if $\Gamma \vdash^c_{(S,F\cup X)} \rho$. □

The equational entailment system. Now we have everything necessary for defining a sound and complete proof theoretic entailment system for conditional equations.

Definition 2.47 (Proof theoretic equational entailment system)**.** *The* proof theoretic equational entailment system *(denoted $\vdash^{\mathrm{e}}$) is the least entailment system for conditional equations which contains the equational proof rules introduced by Dfn. 2.37 and which satisfies the meta-rules of* Implication *and of* Universal Quantification*.*

The proof theoretic equational entailment system is obtained by the intersection of all entailment systems containing the equational proof rules and satisfying the meta-rules of Implication and of Universal Quantification. That this intersection is an entailment system follows by Prop. 2.33. It also satisfies the above mentioned meta-rules because both of these are preserved under arbitrary intersection of entailment systems, a property which can be easily checked in the manner of the proof of Prop. 2.33 (we therefore omit here the details of this proof and leave it as an exercise). We have the following important consequence.

Corollary 2.48. *The proof theoretic equational entailment system $\vdash^{\mathrm{e}}$ is compact.*

Proof. Because the equational proof rules of Dfn. 2.37 are finitary, they are contained by $(\vdash^{\mathrm{e}})^c$ which according to Prop. 2.46 is an entailment system satisfying *Implication* and *Universal Quantification* that is less than $\vdash^{\mathrm{e}}$ (since by 'Transitivity' always $\Gamma \vdash^c E$ implies $\Gamma \vdash E$). Since $\vdash^{\mathrm{e}}$ is the least one with these properties, it follows that $\vdash^{\mathrm{e}} = (\vdash^{\mathrm{e}})^c$. □

A more constructive description of $\vdash^{\mathrm{e}}$ is given by adding iteratively to the proof rules the general properties of entailment relations plus the meta-rules of *Implication* and of *Universal Quantification* as follows. For each signature (S,F), each set of S-sorted variables X, any sets $\Gamma, \Gamma_1, \Gamma_2$ of conditional equations for (S,F), any conditional equation $H \Rightarrow C$ for (S,F), and any conditional equation ρ for $(S,F\cup X)$:

0. Let $\vdash^0_{(S,F)}$ be the set of pairs of sets of conditional equations for (S,F) consisting of the equational proof rules for (S,F) of Dfn. 2.37 and of all pairs (Γ,Γ') for which $\Gamma \subseteq \Gamma'$.

   For each natural number k we let:

1. $\vdash^{4k+1}_{(S,F)} = (\vdash^{4k}_{(S,F)}) \cup \{(\Gamma, \Gamma_1 \cup \Gamma_2) \mid \Gamma \vdash^{4k}_{(S,F)} \Gamma_1 \text{ and } \Gamma \vdash^{4k}_{(S,F)} \Gamma_2\}$.
2. $\vdash^{4k+2}_{(S,F)} = (\vdash^{4k+1}_{(S,F)}) \cup \{(\Gamma, \Gamma_2) \mid \Gamma \vdash^{4k+1}_{(S,F)} \Gamma_1 \text{ and } \Gamma_1 \vdash^{4k+1}_{(S,F)} \Gamma_2\}$.
3. $\vdash^{4k+3}_{(S,F)} = (\vdash^{4k+2}_{(S,F)}) \cup \{(\Gamma, \{H \Rightarrow C\}) \mid \Gamma \cup H \vdash^{4k+2}_{(S,F)} C\} \cup \{(\Gamma \cup H, \{C\} \mid \Gamma \vdash^{4k+2}_{(S,F)} H \Rightarrow C\}$.
4. $\vdash^{4k+4}_{(S,F)} = (\vdash^{4k+3}_{(S,F)}) \cup \{(\Gamma, \{(\forall X)\rho\}) \mid \Gamma \vdash^{4k+3}_{(S,F \cup X)} \rho\}$.
5. $\vdash^{4k+4}_{(S,F \cup X)} = (\vdash^{4k+3}_{(S,F \cup X)}) \cup \{(\Gamma, \{\rho\}) \mid \Gamma \vdash^{4k+3}_{(S,F)} (\forall X)\rho\}$.

Finally, we define $\vdash^{\mathrm{e}}_{(S,F)} = \bigcup_{n \in \omega} \vdash^n_{(S,F)}$. Note that $\vdash^{\mathrm{e}}$ thus defined is indeed an entailment system satisfying the two required meta-rules. Moreover, for any similar entailment system $\vdash$ it is easy to show by induction on $n \in \omega$ that $\vdash^n_{(S,F)} \subseteq \vdash_{(S,F)}$ hence we have that $\vdash^{\mathrm{e}} \subseteq \vdash$.

As an application of Dfn. 2.47 let us develop a fully formal proof of $(\forall \mathtt{X})\mathtt{X} + 0 = \mathtt{X}$ from GROUP. We have already gave a proof of this towards the beginning of the section that was presented in the manner of the usual mathematical proofs thus involving a high degree of informality. Later on, after Prop. 2.31, we have made it more formal by making explicit use of the general properties of entailment systems. Now Dfn. 2.47 puts us in the position to present this deduction in a fully formal way. As previously done, let us use the fact that

$$\mathtt{GROUP} \vdash^{\mathrm{e}} \{(\forall \mathtt{X})\mathtt{X} + (-\mathtt{X}) = 0\} \tag{2.17}$$

Let us denote by Σ the signature of GROUP and and let $\Sigma + a$ denote its extension with a constant a. Let us also recall our previous notations:

- A_1 for the associativity axiom $(\forall \mathtt{X},\mathtt{Y},\mathtt{Z})\mathtt{X} + (\mathtt{Y} + \mathtt{Z}) = (\mathtt{X} + \mathtt{Y}) + \mathtt{Z}$,
- A_2 for $(\forall \mathtt{X})0 + \mathtt{X} = \mathtt{X}$,
- A_3 for $(\forall \mathtt{X})(-\mathtt{X}) + \mathtt{X} = \mathtt{0}$, and
- E_2 for $(\forall \mathtt{X})\mathtt{X} + (-\mathtt{X}) = \mathtt{0}$.

Then our formal deduction goes as follows, first by applying the equational proof rules. By *Reflexivity*:

$$\emptyset \vdash^{\mathrm{e}}_{\Sigma+a} \{a = a\} \tag{2.18}$$

By *Symmetry*:

$$\{0 + a = a\} \vdash^{\mathrm{e}}_{\Sigma+a} \{a = 0 + a\} \tag{2.19}$$

$$\{a + (-a) = 0\} \vdash^{\mathrm{e}}_{\Sigma+a} \{0 = a + (-a)\} \tag{2.20}$$

$$\{a=(a+(-a))+a\} \vdash^{\mathrm{e}}_{\Sigma+a} \{(a+(-a))+a=a\} \quad (2.21)$$
$$\{a+((-a)+a)=a+0\} \vdash^{\mathrm{e}}_{\Sigma+a} \{a+0=a+((-a)+a)\} \quad (2.22)$$

By *Transitivity*:

$$\{a=0+a,\ 0+a=(a+(-a))+a\} \vdash^{\mathrm{e}}_{\Sigma+a} \{a=(a+(-a))+a\} \quad (2.23)$$
$$\{a+((-a)+a)=(a+(-a))+a,\ (a+(-a))+a=a\} \vdash^{\mathrm{e}}_{\Sigma+a} \{a+((-a)+a)=a\} \quad (2.24)$$
$$\{a+0=a+((-a)+a),\ a+((-a)+a))=a\} \vdash^{\mathrm{e}}_{\Sigma+a} \{a+0=a\} \quad (2.25)$$

By *Congruence*:

$$\{0=a+(-a),\ a=a\} \vdash^{\mathrm{e}}_{\Sigma+a} \{0+a=(a+(-a))+a\} \quad (2.26)$$
$$\{a=a,\ (-a)+a=0\} \vdash^{\mathrm{e}}_{\Sigma+a} \{a+((-a)+a)=a+0\} \quad (2.27)$$

By *Substitutivity* for $\mathtt{X} \mapsto a$, $\mathtt{Y} \mapsto (-a)$, $\mathtt{Z} \mapsto a$:

$$\{A_1\} \vdash^{\mathrm{e}}_{\Sigma+a} \{a+((-a)+a)=(a+(-a))+a\} \quad (2.28)$$
$$\{A_2\} \vdash^{\mathrm{e}}_{\Sigma+a} \{0+a=a\} \quad (2.29)$$
$$\{A_3\} \vdash^{\mathrm{e}}_{\Sigma+a} \{(-a)+a=0\} \quad (2.30)$$
$$\{E_2\} \vdash^{\mathrm{e}}_{\Sigma+a} \{a+(-a)=0\} \quad (2.31)$$

Now we apply the general properties of entailment systems.

$$\{A_2\} \vdash^{\mathrm{e}}_{\Sigma+a} \{a=0+a\} \quad \text{from 2.29 and 2.19 by 'transitivity'} \quad (2.32)$$
$$\{E_2\} \vdash^{\mathrm{e}}_{\Sigma+a} \{0=a+(-a)\} \quad \text{from 2.31 and 2.20 by 'transitivity'} \quad (2.33)$$
$$\{E_2\} \vdash^{\mathrm{e}}_{\Sigma+a} \{0+a=(a+(-a))+a\} \quad \text{from 2.33, 2.18 and 2.26} \quad (2.34)$$
$$\{A_2,E_2\} \vdash^{\mathrm{e}}_{\Sigma+a} \{a=(a+(-a))+a\} \quad \text{from 2.32, 2.34 and 2.23} \quad (2.35)$$
$$\{A_2,E_2\} \vdash^{\mathrm{e}}_{\Sigma+a} \{(a+(-a))+a=a\} \quad \text{from 2.35 and 2.21 by 'transitivity'} \quad (2.36)$$
$$\{A_1,A_2,E_2\} \vdash^{\mathrm{e}}_{\Sigma+a} \{a+((-a)+a)=a\} \quad \text{from 2.28, 2.36 and 2.24} \quad (2.37)$$
$$\{A_3\} \vdash^{\mathrm{e}}_{\Sigma+a} \{a+((-a)+a)=a+0\} \quad \text{from 2.18, 2.30 and 2.27} \quad (2.38)$$
$$\{A_3\} \vdash^{\mathrm{e}}_{\Sigma+a} \{a+0=a+((-a)+a)\} \quad \text{from 2.38 and 2.22 by 'transitivity'} \quad (2.39)$$
$$\{A_1,A_2,A_3,E_2\} \vdash^{\mathrm{e}}_{\Sigma+a} \{a+0=a\} \quad \text{from 2.39, 2.37 and 2.25} \quad (2.40)$$
$$\mathtt{GROUP}=\{A_1,A_2,A_3\} \vdash^{\mathrm{e}}_{\Sigma+a} \{a+0=a\} \quad \text{from 2.40 and 2.17} \quad (2.41)$$

By applying the meta-rule of *Universal Quantification* we further obtain that

$$\mathtt{GROUP} \vdash^{\mathrm{e}}_{\Sigma} \{(\forall a)a+0=a\} \quad (2.42)$$

By *Substitutivity* for $a \mapsto \mathtt{X}$:

$$\{(\forall a)a+0=a\} \vdash^{\mathrm{e}}_{\Sigma+\mathtt{X}} \{\mathtt{X}+0=\mathtt{X}\} \quad (2.43)$$

By 'transitivity' and *Universal Quantification* we finally obtain that

$$\mathtt{GROUP} \vdash^{\mathrm{e}}_{\Sigma} \{(\forall \mathtt{X})\mathtt{X}+0=\mathtt{X}\} \quad (2.44)$$

Note how the size of this proof has increased dramatically with the increase in the degree of formality.

Soundness of equational deduction. The precise mathematical formulation for the general correctness of equational deduction is that the proof theoretic entailment system of Dfn. 2.47 is sound, which at this stage is rather easy to establish.

Proposition 2.49 (Soundness of equational deduction)**.** *The proof theoretic equational entailment system is sound.*

Proof. If the semantic entailment system $\models$ satisfied the properties from Dfn. 2.47, then since $\vdash^{\mathrm{e}}$ is the *least* entailment system satisfying those properties we may conclude with the soundness property for each signature Σ, i.e. $\vdash^{\mathrm{e}}_{\Sigma} \subseteq \models_{\Sigma}$.

That $\models$ contains the equational proof rules means precisely the soundness of the latter, which has been established by Prop. 2.36. That $\models$ satisfies the Implication and the Universal Quantification meta-rules, respectively, has been established by Prop. 2.41 and 2.43, respectively. □

Completeness of equational deduction. Completeness of equational logic is originally due, for the single sorted case, to a famous result by Birkhoff [2]. This has been extended to the many sorted case in [19]. Recently the essence of the equational completeness phenomenon has been captured in [8] to a very general abstract setting based upon the so-called 'institution theory' of [17]. This has lead to a myriad of completeness results for various logical systems, many of these results being quite remote in form from the original equational completeness theorem.

Completeness of equational logic is a key to making rewriting, the main execution procedure for equational specifications, into a decision procedure.

Theorem 2.50 (Completeness of equational deduction)**.** *The proof theoretic equational entailment system is complete.*

Proof. We have to show that for any set Γ of conditional equations for a signature (S,F) and for any conditional equation ρ for (S,F), $\Gamma \models \{\rho\}$ implies $\Gamma \vdash \{\rho\}$.
Let us consider the binary relation $\equiv_{\Gamma}$ on the term algebra $0_{(S,F)}$ defined by:

$$t \equiv_{\Gamma} t' \text{ if and only if } \Gamma \vdash^{\mathrm{e}} \{t = t'\}.$$

The relation $\equiv_{\Gamma}$ is

- reflexive because for each (S,F)-term t we have succesively that $\emptyset \vdash^{\mathrm{e}} \{t = t\}$ by the proof rule of *Reflexivity*, $\Gamma \vdash^{\mathrm{e}} \emptyset$ by the 'monotonicity' of $\vdash^{\mathrm{e}}_{(S,F)}$, and $\Gamma \vdash^{\mathrm{e}} \{t = t\}$ from these two entailments above and by the 'transitivity' of $\vdash^{\mathrm{e}}_{(S,F)}$,
- symmetric because whenever $\Gamma \vdash^{\mathrm{e}} \{t = t'\}$ for any (S,F)-terms t and t' of the same sort, by the proof rule of *Symmetry* we have that $\{t = t'\} \vdash^{\mathrm{e}} \{t' = t\}$, which by the 'transitivity' property of $\vdash^{\mathrm{e}}_{(S,F)}$ implies that $\Gamma \vdash^{\mathrm{e}} \{t' = t\}$, and
- transitive because whenever $\Gamma \vdash^{\mathrm{e}} \{t = t'\}$ and $\Gamma \vdash^{\mathrm{e}} \{t' = t''\}$ for any (S,F)-terms t, t' and t'' of the same sort, by the 'union' property of $\vdash^{\mathrm{e}}_{(S,F)}$ we have $\Gamma \vdash^{\mathrm{e}} \{t = t',\ t' = t''\}$, by the proof rule of *Transitivity* we have that $\{t = t',\ t' = t''\} \vdash^{\mathrm{e}} \{t = t''\}$, and

from the last two entailments, by the 'transitivity' property of $\vdash^{\mathrm{e}}_{(S,F)}$, we obtain that $\Gamma \vdash^{\mathrm{e}} \{t = t''\}$.

Hence $\equiv_\Gamma$ is an equivalence. Moreover $\equiv_\Gamma$ is

– an F-congruence because if $\Gamma \vdash^{\mathrm{e}} \{t_i = t'_i\}$ for $1 \leq i \leq n$ and $\sigma \in F_{w \to s}$ then by the 'union' property of $\vdash^{\mathrm{e}}_{(S,F)}$ we have that $\Gamma \vdash^{\mathrm{e}} \{t_i = t'_i \mid 1 \leq i \leq n\}$, by the proof rule of *Congruence* we have that $\{t_i = t'_i \mid 1 \leq i \leq n\} \vdash^{\mathrm{e}} \{\sigma(t_1, \ldots, t_n) = \sigma(t'_1, \ldots, t'_n)\}$, and from the last two entailments, by the 'transitivity' property of $\vdash^{\mathrm{e}}_{(S,F)}$, we obtain that $\Gamma \vdash^{\mathrm{e}} \{\sigma(t_1, \ldots, t_n) = \sigma(t'_1, \ldots, t'_n)\}$.

Let us now prove that $\equiv_\Gamma$ is a Γ-congruence. For this we consider any conditional equation $(\forall X)H \Rightarrow C$ from Γ and any $(S, F \cup X)$-expansion A' of the term algebra $0_{(S,F)}$. We have to show that $A'_H \subseteq \equiv_\Gamma$ implies $A'_C \subseteq \equiv_\Gamma$. Let $\theta : X \to T_F$ be the substitution defined by $\theta(x) = A'_x$ for each $x \in X$. Then by induction on the structure of any $(S, F \cup X)$-term t it is easy to see that $\theta(t) = A'_t$. From this it follows that $A'_H \subseteq \equiv_\Gamma$ means

$$\Gamma \vdash^{\mathrm{e}} \theta(H) \tag{2.45}$$

We can establish the following sequence of entailments:

$$\Gamma \vdash^{\mathrm{e}} \{(\forall X)H \Rightarrow C\} \quad \text{by the 'monotonicity' property of } \vdash^{\mathrm{e}}_{(S,F)} \tag{2.46}$$

$$\{(\forall X)H \Rightarrow C\} \vdash^{\mathrm{e}} \{\theta(H) \Rightarrow \theta(C)\} \quad \text{by the proof rule of } \textit{Substitutivity} \tag{2.47}$$

$$\Gamma \vdash^{\mathrm{e}} \{\theta(H) \Rightarrow \theta(C)\} \quad \text{from 2.46 and 2.47 by the 'transitivity' of } \vdash^{\mathrm{e}}_{(S,F)} \tag{2.48}$$

$$\Gamma \cup \theta(H) \vdash^{\mathrm{e}} \theta(C) \quad \text{from 2.48 the } \textit{Implication} \text{ meta-rule for } \vdash^{\mathrm{e}}_{(S,F)} \tag{2.49}$$

$$\Gamma \vdash^{\mathrm{e}} \Gamma \quad \text{by the 'monotonicity' property of } \vdash^{\mathrm{e}}_{(S,F)} \tag{2.50}$$

$$\Gamma \vdash^{\mathrm{e}} \Gamma \cup \theta(H) \quad \text{from 2.50 and 2.45 by the 'union' property of } \vdash^{\mathrm{e}}_{(S,F)} \tag{2.51}$$

$$\Gamma \vdash^{\mathrm{e}} \theta(C) \quad \text{from 2.51 and 2.49 by the 'transitivity' property of } \vdash^{\mathrm{e}}_{(S,F)} \tag{2.52}$$

Since for each $(S, F \cup X)$-term t we have that $A'_t = \theta(t)$, the entailment 2.52 just means $A'_C \subseteq \equiv_\Gamma$. We have thus proved that $\equiv_\Gamma$ is a Γ-congruence on $0_{(S,F)}$. By Prop. 2.24 it follows that $0_{(S,F)}/_{\equiv_\Gamma} \models \Gamma$.

Consider an equation $t = t'$ such that $\Gamma \models t = t'$. Because $0_{(S,F)}/_{\equiv_\Gamma} \models \Gamma$ we have that $0_{(S,F)}/_{\equiv_\Gamma} \models t = t'$ which means $(0_{(S,F)}/_{\equiv_\Gamma})_t = (0_{(S,F)}/_{\equiv_\Gamma})_{t'}$ which means $t/_{\equiv_\Gamma} = t'/_{\equiv_\Gamma}$ which means $t \equiv_\Gamma t'$. Hence $\Gamma \vdash^{\mathrm{e}} t = t'$. We have thus obtained the completeness for the quantifier free equations. We now extend it to all conditional equations $(\forall X)H \Rightarrow C$.

Assume $\Gamma \models_{(S,F)} (\forall X)H \Rightarrow C$. Then by *Universal Quantification* for $\models$ (cf. Prop. 2.43) we obtain $\Gamma \models_{(S,F \cup X)} H \Rightarrow C$ and by *Implication* for $\models$ (cf. Prop. 2.41) we further obtain that $\Gamma \cup H \models_{(S,F \cup X)} C$. By the completeness for the quantifier free equations that has been established above in this proof, we have that $\Gamma \cup H \vdash^{\mathrm{e}}_{(S,F \cup X)} C$. And now we go the same way opposite direction for $\vdash^{\mathrm{e}}$ instead of $\models$. Because $\vdash^{\mathrm{e}}$ satisfies *Implication* we have that $\Gamma \vdash^{\mathrm{e}}_{(S,F \cup X)} H \Rightarrow C$ and because it satisfies *Universal Quantification* we finally obtain that $\Gamma \vdash^{\mathrm{e}}_{(S,F)} (\forall X)H \Rightarrow C$. □

Completeness of equational deduction has many important consequence, an immediate one being the compactness of the semantic entailment.

Corollary 2.51. *The semantic entailment $\models$ for conditional equations is compact.*

Proof. From the compactness of $\vdash^{e}$ (Prop. 2.48) and from the completeness result of Thm. 2.50. □

The key to the proof of the completeness of equational deduction is that the relation $\{(t,t') \mid \Gamma \vdash^{e} t = t'\}$ is a Γ-congruence on the initial (term) algebra. It can be easily shown that in fact it is $=_\Gamma$, the least Γ-congruence. The following result, also a consequence of the completeness Thm. 2.50, generalizes this important insight.

Definition 2.52 (Reachable algebra)**.** *An (S,F)-algebra A is reachable when for each element a of A there exists an (S,F)-term t such that $a = A_t$.*

Corollary 2.53. *For any set Γ of conditional (S,F)-equations, for any reachable (S,F)-algebra A and for any (S,F)-terms t and t' of the same sort*

$$E(A) \cup \Gamma \vdash^{e} t = t' \text{ if and only if } A_t =^A_\Gamma A_{t'}$$

where by $E(A)$ we denote $\{t = t' \mid A \models t = t'\}$ and where $=^A_\Gamma$ is the least Γ-congruence on A.

Proof. For the implication from the left to the right let us assume $E(A) \cup \Gamma \vdash^{e} t = t'$. By the soundness result of Prop. 2.49 it follows that $E(A) \cup \Gamma \models t = t'$. Since the quotient $A/_{=^A_\Gamma}$ satisfies both Γ (cf. Prop. 2.24) and $E(A)$ (because $(A/_{=^A_\Gamma})_t = (A_t)/_{=^A_\Gamma}$ for each term t) it follows that $A/_{=^A_\Gamma} \models t = t'$ which means $A_t =^A_\Gamma A_{t'}$.

For the implication from the right to the left let us assume that $A_t =^A_\Gamma A_{t'}$. Let $0_{E(A)\cup\Gamma}$ denote the initial algebra satisfying $E(A) \cup \Gamma$ (see Cor. 2.28). There exists a homomorphism $h\colon A \to 0_{E(A)\cup\Gamma}$ defined by $h(A_t) = (0_{E(A)\cup\Gamma})_t$ for each term t. This is defined on each element of A because A is reachable and it is defined correctly because if $A_{t1} = A_{t2}$ then $(t1 = t2) \in E(A)$ and thus $0_{E(A)\cup\Gamma} \models t1 = t2$. It is also straightforward to check that h is indeed a homomorphism. Since $A/_{=^A_\Gamma}$ is the free algebra over A satisfying Γ (see Cor. 2.26), there exists an unique homomorphism $h'\colon A/_{=^A_\Gamma} \to 0_{E(A)\cup\Gamma}$ such that $h = q^A_\Gamma; h'$. Thus for any term t we have that $h(A_t) = h'((A/_{=^A_\Gamma})_t) = (0_{E(A)\cup\Gamma})_t$, hence $A_t =^A_\Gamma A_{t'}$ implies $0_{E(A)\cup\Gamma} \models t = t'$. Now let us take any model M such that $M \models E(A) \cup \Gamma$. By considering the unique homomorphism $0_{E(A)\cup\Gamma} \to M$ we obtain that $M \models t = t'$, which proves that $E(A) \cup \Gamma \models t = t'$. Then $E(A) \cup \Gamma \vdash^{e} t = t'$ by the completeness of equational deduction (Thm. 2.50). □

Exercises.

2.11. Given substitutions $\theta\colon X \to T_{(S,F\cup Y)}$ and $\psi\colon Y \to T_{(S,F\cup Z)}$, their *composition* $\theta;\psi\colon X \to T_{(S,F\cup Z)}$ is defined by $(\theta;\psi)(x) = \psi^\sharp(\theta(x))$. Show that the composition of substitutions is associative.

2.12. Develop the detailed proof of the fact that the meta-rules of *Implication* and of *Universal Quantification* are preserved under arbitrary intersections of entailment systems.

2.13. Let X and Y be disjoint sets of variables for a signature (S,F). Prove the following more general form of *Substitutivity*, namely that for any $(S,F\cup X)$-sentence ρ and any substitution $\theta\colon X \to T_F(Y)$,

$$\{(\forall X)\rho\} \models_{(S,F)} \{(\forall Y)\theta(\rho)\}.$$

2.14. Show that the unconditional equations admit a sound and complete proof theoretic entailment system which is the least entailment system containing the equational proof rules of Dfn. 2.37 (with the *Substitutivity* rule in the unconditional form $\frac{(\forall X)C}{\theta(C)}$) and satisfying the meta-rule of Universal Quantification only.

2.15. Prove by equational deduction that any group satisfies the equation $(\forall$x$)$ `-(- x) = x`.

2.16. A group has *characteristic 2* if and only if it satisfies the equation $(\forall$x$)$ `x + x = 0`. Prove by equational deduction that any group of characteristic 2 is commutative, i.e. it satisfies the equation $(\forall$x,y$)$ `x + y = y + x`.

2.4 Rewriting

In its standard form that was presented in the section above, equational deduction is rather difficult to mechanize. This difficulty owes to the nature of the equational proof rules leading to a lot choices to make about what rules to use and how to use them. Without the human mind making these choices, this means an exponential explosion of the search space of the deduction process. In this section we present the well established method to overcome this problem in mechanizing equational deduction. This method is known as 'term rewriting' or just 'rewriting' in more general contexts.

The term rewriting entailment system. An important step for giving direction to the equational deduction process is to eliminate the rule of *Symmetry*, which means that the equations are used from the left to the right only.

Definition 2.54 (Rewriting entailment)**.** *The* (term) rewriting entailment system *(denoted* $\vdash^{\mathrm{r}}$*) is the least entailment system for conditional equations containing the proof rules of* Reflexivity*,* Transitivity*,* Congruence *and* Substitutivity *(of Dfn. 2.37) and which satisfies the meta-rules of* Implication *and of* Universal Quantification*.*

From this definition it is clear that the rewriting entailment system is less than the equational one, i.e. $\vdash^{\mathrm{r}}_{(S,F)} \subseteq \vdash^{\mathrm{e}}_{(S,F)}$ for each signature (S,F). An immediate consequence of this observation is that the soundness of the equational entailment system (Cor. 2.50) implies the soundness of the rewriting entailment system.

Proposition 2.55 (Soundness of rewriting)**.** *The rewriting entailment system is sound.*

Below in the section we will see that although in general the power of deduction without the rule of *Symmetry* is less than the full equational deduction, this difference can be

overcome by some other conditions which are fullfilled by a large spectrum of equational specifications.

Now we consider a further step in mechanizing equational deduction, namely that of the amalgamation of the rules of *Congruence* and *Substitutivity* as a single proof rule. For this we need the following concept.

Definition 2.56 (Rewriting contexts)**.** *Given a signature (S,F), an $(S,F\cup\{z\})$-term c over the signature extended with a new variable z is a* (rewriting) (S,F)-context *if*

- $c = z$, *or*
- $c = \sigma(c_1,\ldots,c_n)$ *such that* $\sigma \in F_{w\to s}$ *is an operation symbol and there exists exactly one* $k \in \{1,\ldots,n\}$ *such that* c_k *is context, with* c_i *being just* (S,F)*-terms for* $i \neq k$.

Then c_k *is called the* immediate sub-context *of c. A term* c' *is a* sub-context *of (a context) c if it is either the immediate sub-context of c or else it is a sub-context of the immediate sub-context of c.*

More informally, an (S,F)-context is an $(S,F\cup\{z\})$-term with exactly one occurence of z. Often, in order to emphasize the new variable z we may denote contexts by $c[z]$. In that case, if t is any term of the sort of z, by $c[t]$ we denote the term resulting from the replacement of z by t in $c[z]$.

Proposition 2.57. *The rewriting entailment system is the least entailment system containing the proof rules of* Reflexivity*,* Transitivity*, and*

Rewriting: $$\frac{\{(\forall X)H \Rightarrow (t=t')\} \;\cup\; \theta(H)}{c[\theta(t)] = c[\theta(t')]}$$ *for any substitution* $\theta\colon\; X \to T_{(S,F)}$ *and each context c.*

and which satisfies the meta-rules of Implication *and of* Universal Quantification*.*

Proof. Let us first show that the rules of *Congruence* and *Substitutivity* are contained by the entailment system defined in the statement of the proposition.

For *Substitutivity*, by taking the context c as just the variable z, by *Rewriting* we have that

$$\{(\forall X)H \Rightarrow (t=t')\} \cup \theta(H) \;\vdash\; \theta(t) = \theta(t')$$

which by *Implication* implies

$$\{(\forall X)H \Rightarrow (t=t')\} \;\vdash\; \theta(H) \Rightarrow (\theta(t) = \theta(t')).$$

For *Congruence*, for the sake of the simplicity of presentation of the argument, let us consider a binary operation σ and adequate terms t_1, t'_1, t_2 and t'_2.[4] We have to prove that

$$\{t_1 = t'_1, t_2 = t'_2\} \vdash \sigma(t_1,t_2) = \sigma(t'_1,t'_2)$$

[4] The same argument can be easily extended to the case when σ has bigger arity.

We have the following:

$$\{t_2 = t_2'\} \vdash \sigma(t_1,t_2) = \sigma(t_1,t_2') \quad \text{by } \textit{Rewriting} \text{ for the context } c[z] = \sigma(t_1,z) \quad (2.53)$$

$$\{t_1 = t_1'\} \vdash \sigma(t_1,t_2') = \sigma(t_1',t_2') \quad \text{by } \textit{Rewriting} \text{ for the context } c[z] = \sigma(z,t_2') \quad (2.54)$$

$$\{\sigma(t_1,t_2) = \sigma(t_1,t_2'),\ \sigma(t_1,t_2') = \sigma(t_1',t_2')\} \vdash \sigma(t_1,t_2) = \sigma(t_1',t_2') \quad \text{by } \textit{Transitivity} \quad (2.55)$$

From (2.53) and (2.54) by 'monotonicity', 'transitivity' and 'union' properties of $\vdash$:

$$\{t_1 = t_1',\ t_2 = t_2'\} \vdash \{\sigma(t_1,t_2) = \sigma(t_1,t_2'),\ \sigma(t_1,t_2') = \sigma(t_1',t_2')\} \quad (2.56)$$

The desired relation now follows from (2.56) and (2.55) by the 'transitivity' of $\vdash$.

We have thus showed that the rewriting entailment system is less than the entailment system defined by the statement of the proposition. For showing the other opposite inclusion, we have to show that the rule of *Rewriting* is contained by the rewriting entailment system. This means that for each substitution $\theta : X \to T_{(S,F)}$ and for each appropriate context $c[z]$ we have to show that

$$\{(\forall X)H \Rightarrow (t = t')\} \cup \theta(H) \vdash^{\mathrm{r}} c[\theta(t)] = c[\theta(t')]$$

We have that

$$\{(\forall X)H \Rightarrow (t = t')\} \vdash^{\mathrm{r}} \theta(H) \Rightarrow (\theta(t) = \theta(t')) \quad \text{by } \textit{Substitutivity} \quad (2.57)$$

and from (2.57) by the meta-rule of *Implication* that

$$\{(\forall X)H \Rightarrow (t = t')\} \cup \theta(H) \vdash^{\mathrm{r}} (\theta(t) = \theta(t')) \quad (2.58)$$

Hence by the 'transitivity' property of entailment it would be enough to show, for any appropriate context $c[z]$, that

$$\{\theta(t) = \theta(t')\} \vdash^{\mathrm{r}} c[\theta(t)] = c[\theta(t')] \quad (2.59)$$

For this we may think of $\theta(t)$ and $\theta(t')$ as any two terms of the same sort, and therefore it is enough to show the following simpler variant of (2.59):

$$\{t = t'\} \vdash^{\mathrm{r}} c[t] = c[t'] \quad (2.60)$$

We show (2.60) by induction on the structure, or on the depth, of the context $c[z]$.

The base case of this induction is represented by the situation when $c[z]$ is just the variable z. Then (2.60) follows directly from the 'monotonicity' property of entailment systems.

For the step case, let us assume that $c = \sigma(c_1, \ldots, c_n)$. Without any loss of generality we may assume c_1 is the immediate sub-context of c. The induction hypothesis means that the considered property holds for c_1, i.e.

$$\{t = t'\} \vdash^{\mathrm{r}} c_1[t] = c_1[t']. \quad (2.61)$$

For each $2 \leq i \leq n$, by *Reflexivity* we have that

$$\emptyset \vdash c_i = c_i \quad (2.62)$$

and from (2.61) and (2.62) by 'monotonicity' and 'union' of $\vdash^{\mathrm{r}}$ we have that

$$\{t = t'\} \vdash^{\mathrm{r}} \{c_1[t] = c_1[t'], c_2 = c_2, \dots, c_n = c_n\}. \tag{2.63}$$

We apply now the rule of *Congruence* and obtain that

$$\{c_1[t] = c_1[t'], c_2 = c_2, \dots, c_n = c_n\} \vdash^{\mathrm{r}} \sigma(c_1(t), c_2, \dots c_n) = \sigma(c_1(t'), c_2, \dots c_n) \tag{2.64}$$

Finally, (2.60) is obtained from (2.63) and (2.64) by the 'transitivity' of $\vdash^{\mathrm{r}}$. □

The rewriting relation on terms. Given a set Γ of conditional equations for a signature (S,F) we may define the following *rewriting relation* on (S,F)-terms:

$$t \xrightarrow{\star}_{\Gamma} t' \text{ if and only if } \Gamma \vdash^{\mathrm{r}} t = t'.$$

Because $\vdash^{\mathrm{r}}$ contains the rules of *Reflexivity* and *Transitivity* we have the following immediate consequence:

Corollary 2.58. *The rewriting relation $\xrightarrow{\star}_{\Gamma}$ is reflexive and transitive.*

The description of the rewriting entailment relation given by Prop. 2.57 shows that $t \xrightarrow{\star}_{\Gamma} t'$ means that there exists a sequence of terms $t = t_0, t_1, \dots, t_n = t'$ such that for each $k \in \{1, \dots, n-1\}$ the equality $t_k = t_{k+1}$ is obtained as a conclusion by applying (once) the *Rewriting* rule for some conditional equation $(\forall X)H \Rightarrow C$ in Γ. For this we often use terminology such as 'performing one rewrite step' and denote it by $t_k \longrightarrow_{\Gamma} t_{k+1}$.

The following simple result represents the standard procedure for performing equational proofs by rewriting.

Proposition 2.59. *If there exists a term t such that $t_1 \xrightarrow{\star}_{\Gamma} t$ and $t_2 \xrightarrow{\star}_{\Gamma} t$ then $\Gamma \models t_1 = t_2$.*

Proof. Since $t_i \xrightarrow{\star}_{\Gamma} t$ for each $i \in \{1,2\}$, by the definition of $\xrightarrow{\star}_{\Gamma}$ we have that $\Gamma \vdash^{\mathrm{r}} t_i = t$. By the soundness of $\vdash^{\mathrm{r}}$ (cf. Prop. 2.55) we have that $\Gamma \models t_i = t$ for each $i \in \{1,2\}$. It follows that $\Gamma \models t_1 = t_2$. □

Let us illustrate the applicability of the method suggested by Prop. 2.59 by a simple example. Recall the following specification of natural numbers that has been introduced above.

```
mod! SIMPLE-NAT {
  [ Nat ]
  op 0 : -> Nat
  op s_ : Nat -> Nat
  op _+_ : Nat Nat -> Nat
  vars M N : Nat
  eq [succ] : N + (s M) = s(N + M) .
  eq [zero] : N + 0 = N .
}
```

By using Prop. 2.59 let us show that

$$\texttt{SIMPLE-NAT} \models \texttt{(s 0) + (s s 0) = (s s 0) + (s 0)}.$$

In the following let Γ denote the two equations of SIMPLE-NAT. By the equation [succ] for the substitution N,M $\mapsto$ (s 0) and for the context $c[z] = z$ we have the following rewriting step

$$\texttt{(s 0) + (s s 0)} \longrightarrow_{\Gamma} \texttt{s((s 0) + (s 0))}. \tag{2.65}$$

By [succ] again for the substitution M $\mapsto$ (s 0) and N $\mapsto$ 0 and for the context $c[z] = \texttt{s}(z)$ we have the following rewriting step

$$\texttt{s((s 0) + (s 0))} \longrightarrow_{\Gamma} \texttt{(s s ((s 0) + 0))}. \tag{2.66}$$

By [zero] for the substitution N $\mapsto$ (s 0) and for the context $c[z] = \texttt{s}(\texttt{s}(z))$ we have the following rewriting step

$$\texttt{(s s ((s 0) + 0))} \longrightarrow_{\Gamma} \texttt{(s s s 0)}. \tag{2.67}$$

From (2.65), (2.66) and (2.67) we have that

$$\texttt{(s 0) + (s s 0)} \stackrel{\star}{\longrightarrow}_{\Gamma} \texttt{(s s s 0)}. \tag{2.68}$$

By [succ] for the substitution M $\mapsto$ (s s 0) and N $\mapsto$ 0 and for the context $c[z] = z$ we have the following rewriting step

$$\texttt{(s s 0) + (s 0)} \longrightarrow_{\Gamma} \texttt{s((s s 0) + 0)}. \tag{2.69}$$

By [zero] for the substitution N $\mapsto$ (s s 0) and for the context $c[z] = \texttt{s}(z)$ we have the following rewriting step

$$\texttt{s((s s 0) + 0)} \longrightarrow_{\Gamma} \texttt{(s s s 0)}. \tag{2.70}$$

From (2.69) and (2.70) we have that

$$\texttt{(s s 0) + (s 0)} \stackrel{\star}{\longrightarrow}_{\Gamma} \texttt{(s s s 0)}. \tag{2.71}$$

From (2.68) and (2.71) by Prop. 2.59 we have that

$$\texttt{SIMPLE-NAT} \models \texttt{(s 0) + (s s 0) = (s s 0) + (s 0)}.$$

The rewriting algorithm. The derivation of terms $t2$ from a term $t1$ by virtue of the rewriting relation $t1 \stackrel{\star}{\longrightarrow}_{\Gamma} t2$ can be considered a computation process when the application of the *Rewriting* rule is presented as an algorithm as follows. We assume a given set Γ of conditional equations for a signature (S,F).

0. Let T be the input term $t1$.
1. Select $(\forall X)H \Rightarrow (t = t')$ from Γ.
2. Select a sub-term T_0 of T such that $T_0 = \theta(t)$ for some substitution $\theta : X \to T_{(S,F)}$.
3. If the step 2. is succesful and $\Gamma \models \theta(H)$

   then replace in T_0 in T by $\theta(t')$ and go to step 1.,

   else select a *new* $(\forall X)H \Rightarrow (t = t')$ from Γ and go to step 2.

There are several aspects of this rewriting algorithm that need special attention. Some of these, such as termination and confluence, have led to extensive studies by the research community.

One aspect is how to deal with the condition $\Gamma \models \theta(H)$ from step 3. Of course, this condition occurs only in the cases when the equation used is not unconditional. In such cases, $\Gamma \models \theta(H)$ is usually proved by rewriting which implies performing a nested proof process by rewriting inside of the actual rewriting algorithm. Alternatively, $\theta(H)$ can be stored as a goal to be proved later, but in this case the rewriting algorithm can be considered terminated only when the proof of $\theta(H)$ is accomplished.

Another aspect is that unless the set X coincides with the set of the variables ocurring in t, the process of finding a substitution $\theta : X \to T_{(S,F)}$ such that $T_0 = \theta(t)$ for T_0 sub-term of T, process that is called *matching*, may have an infinity of solutions since there would be variables (not occuring in t) that could be mapped to any term of appropriate sort. In order that for any T_0 there exists at most one substitution θ such that $T_0 = \theta(t)$, this condition is necessary. For this reason, in many specification languages, including CafeOBJ the notation for the universal quantifier in conditional equations is missing since the set of the variables is implicitly assumed to be that of the variables occuring in t. In these situations, one needs to check that these variables cover all the variables occuring in both t' and in the condition H. For example, CafeOBJ silently does not use for rewriting the equations that do not conform to this condition.

Termination. An important aspect of the rewriting algorithm, which is however of a rather general nature, is that of the termination. Termination is a crucial property of any algorithm, its absence means that the algorithm runs forever and we do not get any result. It is easy to imagine situations when a rewriting algorithm does not terminate, for example a very simple one being when Γ contains a commutativity equation $(\forall$`M,N`$)$ `M + N = N + M`. Then a term such as $a + b$ may get rewritten to $b + a$, then to $a + b$ again and so on. Another non-termination situation, but of a different nature than the one above, is when we have an equation such as $(\forall$`M`$)$ `(s M) = (s s M)`. In this section we will discuss more about termination and in Chap. 3 we will discuss generic techniques for proving algorithm termination.

Confluence. The rewriting algorithm is *non-deterministic* due to the possibility to have several choices for the selections of the equations (steps 1. and 3.) and for the selection of the sub-term T_0 (step 2.).

For example, within the context of the specification `SIMPLE-NAT` above, the term `(0 + (s 0)) + 0` may be rewritten in one step in two different ways:

1. To `0 + (s 0)` by using the equation [`zero`].
2. To `s(0 + 0) + 0` by using the equation [`succ`].

Moreover, in the second case a further rewriting step may also be performed in two different ways by using the same equation [`zero`] depending upon the choice of the sub-term T_0 which can be either the whole term or else the sub-term `0 + 0`.

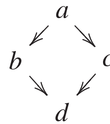
The confluence property means that the result will be the same regardless of the choices we make. We will introduce confluence as a formal property below in this section. As an example, it is easy to see that for `SIMPLE-NAT` the confluence property holds because each term gets eventually rewritten to a term of the form (`ss`...`s0`).

Abstract termination and confluence. Termination and confluence are paramount properties of rewriting, with important implications. The study of these properties can be partially done at the level of abstract sets (instead of sets of terms) and of abstract binary relations (instead of the rewriting relations $\xrightarrow{\star}_\Gamma$). Doing such study abstractly is important in two different ways. One is the simplicity, for some concepts and results we can do without details that are redundant or irrelevant but may make the understanding more difficult. The other is the level of generality involved, some concepts and results maye be used in other contexts, for other algorithms, including more sophisticated versions of rewriting. For example such an abstract approach can be applied to the so-called 'rewriting modulo axioms' discussed below in this section or to the analysis of algorithms in Chap. 3.

Definition 2.60 (Terminating relation)**.** *A binary relation $>$ on a set A is* terminating *if and only if*

– *it is antisymmetric, i.e. $a > b$ and $b > a$ implies $a = b$, and*
– *for each $a \in A$ the set $\{b \mid a > b\}$ is finite.*

Definition 2.61 (Confluent relation)**.** *A binary relation $>$ on a set A is* confluent *if and only if for each $a > b$ and $a > c$ there exists $d \in A$ such that $b > d$ and $c > d$.*



Definition 2.62 (Normal forms)**.** *An element $n \in A$ is a* normal form *for a binary relation $>$ on A when for each element $x \in A$, $n > x$ implies $n = x$. The element n is a normal form of another element a with respect to $>$ when it is a normal form for $>$ and $a > n$.*

In the case of the rewriting relation on terms, from a functional programming perspective, normal forms can be regarded as results of evaluations of terms. The existence of unique normal forms is crucial for the smooth applicability of the proof method by rewriting derived from the Prop. 2.59. As example, the rewriting relation $\stackrel{*}{\longrightarrow}_\Gamma$ defined by `SIMPLE-NAT` has normal forms of all terms, the normal forms being the terms `(s s ...s 0)`.

Proposition 2.63. *Let $>$ be a confluent and terminating preorder relation on a set A. Then each element of A has a unique normal form with respect to $>$.*

Proof. Let us first handle the uniqueness. Assume that an element a has two normal forms, namely n_1 and n_2. By confluence there exists n such that $n_1 > n$ and $n_2 > n$. Because n_i are both normal forms it follows that $n_i = n$. Hence $n_1 = n_2$.

For showing the existence of normal forms let us fix $a \in A$ and suppose that the set $\{b \mid a > b\}$ does not contain any normal form. Note that $\{b \mid a > b\}$ is non-empty by the reflexivity of $>$. Let us pick any $b_0 \in \{b \mid a > b\}$. By induction on $k \in \omega$ we construct a chain of elements $(b_k)_{k\in\omega}$ such that $b_k \in \{b \mid a > b\}$, $b_k > b_{k+1}$ and $b_k \neq b_{k+1}$. At the induction step, we use the fact that b_k is not a normal form since $b_k \in \{b \mid a > b\}$ hence there exists $b_{k+1} \in A$ such that $b_k > b_{k+1}$ and $b_k \neq b_{k+1}$. By the transitivity of $>$ we also obtain that $b_{k+1} \in \{b \mid a > b\}$. By the termination hypothesis we have that $\{b \mid a > b\}$ is finite, hence there exists $k < n$ such that $b_k = b_n$. We have that $b_k > b_{k+1}$ and by the transitivity of $>$ we also have $b_{k+1} > b_n$. Since $b_n = b_k$ this means $b_k > b_{k+1}$ and $b_{k+1} > b_k$. By the antisymmetry condition (since $>$ is terminating) it follows that $b_k = b_{k+1}$. We have thus reached a contradiction with $b_k \neq b_{k+1}$. The conclusion is that our assumption that $\{b \mid a > b\}$ does not contain any normal form is false. □

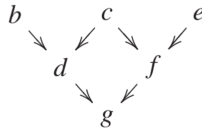
The following result will be used below in this section.

Proposition 2.64. *If $>$ is a confluent preorder relation on a set A then the relation $\downarrow$ defined by*

$$b \downarrow c \text{ if and only if there exists } d \in A \text{ with } b > d \text{ and } c > d$$

is the least equivalence containing $>$.

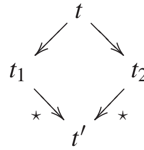
Proof. The symmetry is immediate. For the transitivity assume $b \downarrow c$ and $c \downarrow e$. Then there exists d such that $b > d$ and $c > d$ and there exists f such that $c > f$ and $e > f$.



By confluence there exists g such that $d > g$ and $f > g$. By the transitivity of the preorder we have that $b > g$ and $e > g$, hence $b \downarrow e$. □

Newmann's Lemma. There exists a body of rather elaborated techniques for proving the confluence of a rewriting relation determined by a set Γ of equations, most of this beyond the aims of this textbook. Here we restrict ourselevs only to the presentation of one basic result. This rather famous result is very useful in the applications since it reduces the checking of confluence to the one step rewriting situations.

Definition 2.65 (Church-Rosser relation)**.** *A relation* $\longrightarrow$ *is* Church-Rosser *when its reflexive and transitive closure* $\stackrel{\star}{\longrightarrow}$ *is confluent. It is* locally Church-Rosser *when for any* $t \longrightarrow t_1$ *and* $t \longrightarrow t_2$



there exists t' *such that* $t_1 \stackrel{\star}{\longrightarrow} t'$ *and* $t_2 \stackrel{\star}{\longrightarrow} t'$.

Note that the relation $>$ in the paragraph above corresponds to $\stackrel{\star}{\longrightarrow}$ of Dfn. 2.65, i.e. the reflexive and transitive closure of $\longrightarrow$, rather than to $\longrightarrow$.

Definition 2.66 (Noetherian relation)**.** *A relation* $\longrightarrow$ *is* Noetherian *when there are no infinite chains* $t_0 \longrightarrow t_1 \longrightarrow t_2 \longrightarrow \dots$.

It is immediate to see that if $\stackrel{\star}{\longrightarrow}$ is terminating then $\longrightarrow$ is Noetherian. The opposite does not hold in general, it is easy to find a Noetherian relation $\longrightarrow$ such that its reflexive and transitive closure $\stackrel{\star}{\longrightarrow}$ is not terminating. However in the case of the rewriting relations these two concepts coincide, i.e. $\longrightarrow_\Gamma$ is Noetherian if and only if $\stackrel{\star}{\longrightarrow}_\Gamma$ is terminating (see Ex. 2.19).

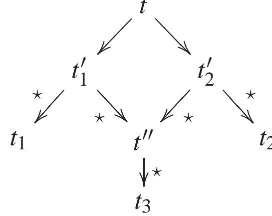
Proposition 2.67 (Newmann's Lemma)**.** *A Noetherian relation* $\longrightarrow$ *is Church-Rosser if and only if it is locally Church-Rosser.*

Proof. We focus on the non-trivial implication, from the right to the left. Because the relation $\longrightarrow$ is Noetherian each element has at least one normal form with respect to $\stackrel{\star}{\longrightarrow}$. Let us say that an element is *ambiguous* when it has at least two normal forms. The conclusion of the proposition follows if we showed that there are no ambiguous elements.

If we showed that for each ambiguous element t there exists another ambiguous element t' such that $t \longrightarrow t'$, then the existence of at least one ambiguous element would contradict the hypothesis that $\longrightarrow$ is Noetherian.

Consider a ambiguous element t and let t_1 and t_2 be two different normal forms of t. We have $t \stackrel{\star}{\longrightarrow} t_1$ and $t \stackrel{\star}{\longrightarrow} t_2$. Each of t_1 and t_2 is also different from t (otherwise we would immediately have $t = t_1 = t_2$). Thus there exists t_1' and t_2' such that $t \to t_1' \stackrel{\star}{\longrightarrow} t_1$ and $t \to t_2' \stackrel{\star}{\longrightarrow} t_2$. By the locally Church-Rosser hypothesis there exists t'' such that $t_1' \stackrel{\star}{\longrightarrow} t''$

and $t_2' \xrightarrow{\star} t''$. Let t_3 be a normal form for t''.



Since $t_1 \neq t_2$ there exists $i \in \{1,2\}$ such that $t_3 \neq t_i$. Then t_i' is ambiguous. □

The rewriting relation on arbitrary algebras. A set Γ of conditional equations for a signature (S,F) induces the following relation on an arbitrary (S,F)-algebra A:

$$\{(A_t, A_{t'}) \mid \Gamma \vdash^{\mathrm{r}} t = t'\}$$

Note that when A is the initial term algebra $0_{(S,F)}$ then the above relation is just the rewriting relation $\xrightarrow{\star}_{\Gamma}$. However, in general the relation $\{(A_t, A_{t'}) \mid \Gamma \vdash^{\mathrm{r}} t = t'\}$ may lack the basic properties of a rewriting relation such as reflexivity and transitivity. Reflexivity can achieved immediately by the assumption of *reachability* of the algebra A (we have just to apply the rule of *Reflexivity*) whilst for the transitivity we have to do the corresponding closure.

Notation 2.68. *Let* $\xrightarrow{+}_{\Gamma,A}$ *denote the least transitive relation containing* $\{(A_t, A_{t'}) \mid \Gamma \vdash^{\mathrm{r}} t = t'\}$ *and* $\xrightarrow{\star}_{\Gamma,A} = \xrightarrow{+}_{\Gamma,A} \cup \{(a,a) \mid a \in A\}$ *be its reflexive-transitive closure.*

We have the following important characterization of $\xrightarrow{\star}_{\Gamma,A}$.

Proposition 2.69. *For any* (S,F)*-algebra* A *and for any set* Γ *of conditional equations for* (S,F)

$$A_t \xrightarrow{\star}_{\Gamma,A} A_{t'} \text{ if and only if } E(A) \cup \Gamma \vdash^{\mathrm{r}} t = t'$$

for any (S,F)*-terms* t *and* t' *and where* $E(A) = \{t = t' \mid A \models t = t'\}$.

Proof. $E(A) \cup \Gamma \vdash^{\mathrm{r}} t = t'$ is equivalent to the existence of a chain of one step rewritings

$$t = t_0 \longrightarrow_{E(A)\cup\Gamma} t_1 \longrightarrow_{E(A)\cup\Gamma} \cdots \longrightarrow_{E(A)\cup\Gamma} t_n = t'.$$

By grouping separately the rewriting steps using equations from $E(A)$ and Γ, respectively, we obtain that $E(A) \cup \Gamma \vdash^{\mathrm{r}} t = t'$ is equivalent to the existence of a chain of terms $t = T_0$, $T_1, \ldots, T_n = t'$ such that

$$T_{2i-1} \xrightarrow{\star}_{E(A)} T_{2i} \text{ and } T_{2i} \xrightarrow{\star}_{\Gamma} T_{2i+1}.$$

The equivalence to be shown now follows by noticing that

1. $T_{2i-1} \xrightarrow{\star}_{E(A)} T_{2i}$ is equivalent to $A_{T_{2i-1}} = A_{T_{2i}}$ because

- on the one hand $T_{2i-1} \stackrel{\star}{\longrightarrow}_{E(A)} T_{2i}$ means $E(A) \vdash^{\mathrm{r}} T_{2i-1} = T_{2i}$ which by the soundness of rewriting (Prop. 2.55) implies $E(A) \models T_{2i-1} = T_{2i}$ meaning $T_{2i-1} = T_{2i} \in E(A)$, and
- on the other hand $A_{T_{2i-1}} = A_{T_{2i}}$ means $T_{2i-1} = T_{2i} \in E(A)$ which by 'monotonicity' of $\vdash^{\mathrm{r}}$ implies $T_{2i-1} \longrightarrow_{E(A)} T_{2i}$,

and

2. $T_{2i} \stackrel{\star}{\longrightarrow}_{\Gamma} T_{2i+1}$ means $\Gamma \vdash^{\mathrm{r}} T_{2i} = T_{2i+1}$.

□

The following is obtained easily as a consequence of the above characterization of $\stackrel{\star}{\longrightarrow}_{\Gamma,A}$.

Proposition 2.70. *For any (S,F)-algebra A the relation $\stackrel{\star}{\longrightarrow}_{\Gamma,A}$ is preserved by the operations of A.*

Proof. Let $\sigma \in F_{s_1 \ldots s_n \to s}$ be an operation symbol of the signature and let t_i, t'_i be (S,F)-terms of sort s_i for $1 \leq i \leq n$ such that $A_{t_i} \stackrel{\star}{\longrightarrow}_{\Gamma,A} A_{t'_i}$ We have to show that $A_\sigma(A_{t_1},\ldots,A_{t_n}) \stackrel{\star}{\longrightarrow}_{\Gamma,A} A_\sigma(A_{t'_1},\ldots,A_{t'_n})$.

By virtue of Prop. 2.69 we have that $A_{t_i} \stackrel{\star}{\longrightarrow}_{\Gamma,A} A_{t'_i}$ means $E(A) \cup \Gamma \vdash^{\mathrm{r}} t_i = t'_i$. By 'union', *Congruence*, and 'transitivity' it follows that $E(A) \cup \Gamma \vdash^{\mathrm{r}} \sigma(t_1,\ldots,t_n) = \sigma(t'_1,\ldots,t'_n)$. By Prop. 2.69 this implies $A_\sigma(A_{t_1},\ldots,A_{t_n}) \stackrel{\star}{\longrightarrow}_{\Gamma,A} A_\sigma(A_{t'_1},\ldots,A_{t'_n})$. □

Definition 2.71 (Preordered algebras)**.** *Any (S,F)-algebra A endowed with a preorder relation $>_s$ on A_s for each $s \in S$ such that $>$ is preserved by the interpretations of the operations, in other words such that A_σ is monotone with respect to $>$ for each $\sigma \in F_{w \to s}$, is called a* preordered algebra*.*

Rewriting modulo axioms. An important instance of rewriting relations on algebras is rewriting modulo axioms. We have seen above that some equations, in spite of the fact that they need to be involved in the deduction process, are not suitable for rewriting because of various different reasons. For example, we have seen that the use of commutativity for in rewriting may lead to non-termination of the rewriting algorithm. Also associativity is bad to use in rewriting, but for a different reason: it may lead to a deadlock of the rewriting process due to accumulation of the bracketing to the left or to the right (depending on how we write the associativity equation). The solution to these problems is to recognize the 'bad' equations, separate them from the ones that are suitable for rewriting, and use them for deduction in a rather implicit way. This idea is directly supported by modern specification languages through the so-called 'operation attributes' (in Sect. 2.1) we have discussed commutativity and associativity as CafeOBJ operation attributes). After the separation of the equations according to their suitability for rewriting, the rewriting is performed on the elements of the initial algebra of the set of the equations not suitable for rewriting. As we may recall from Sect. 2.1, these elements are congruence classes of

terms modulo these equations (recall from Sect. 2.1 that the class of a term t modulo E is denoted $t/_{=E}$).

In other words, we separate the set of the equations of a given specification into a set Γ of equations to be used for rewriting and a set of equations E not to be used for rewriting, and we consider the rewriting relation $\xrightarrow{\star}_{\Gamma,A}$ where A is $0_{(S,F)}/_{=E}$, the initial algebra satisfying E.

Notation 2.72. *If Γ and E are sets of (S,F)-equations, then by $\xrightarrow{\star}_{\Gamma,E}$ we denote the rewriting relation determined by Γ on $0_{(S,F)}/_{=E}$, the initial algebra satisfying E.*

Corollary 2.73. *For any (S,F)-terms t and t' we have that*

$$(t/_{=E}) \xrightarrow{\star}_{\Gamma,E} (t'/_{=E}) \text{ if and only if } \{t_1 = t_2 \mid E \models t_1 = t_2\} \cup \Gamma \vdash^{\mathrm{r}} t = t'.$$

Proof. From Prop. 2.69 when A is 0_E, the initial algebra satisfying E and by noting that for each equation $t_1 = t_2$ we have that $E \models t_1 = t_2$ if and only if $0_E \models t_1 = t_2$. □

From Cor. 2.73 and Prop. 2.55 we obtain immediately the following soundness result.

Corollary 2.74 (Soundeness of rewriting modulo axioms)**.** *For any (S,F)-terms t and t' we have that*

$$(t/_{=E}) \xrightarrow{\star}_{\Gamma,E} (t'/_{=E}) \text{ implies } \Gamma \cup E \models t = t'.$$

As a concrete example of equational deduction by rewriting modulo axioms let us recall the specification of group theory of Sect. 2.3:

```
mod* GROUP {
  [ G ]
  op 0 : -> G
  op _+_ : G G -> G {assoc}
  op -_ : G -> G
  var X : G
  eq  [id] : 0 + X = X .
  eq [inv] : (- X) + X = 0 .
}
```

Since associativity is not suitable for rewriting, the associativity axiom for + is specified as an operation attribute rather than an ordinary equation. Thus only the equations `[id]` and `[inv]` are used for rewriting, which in this case takes place in the initial algebra modulo, i.e. satisfying, the associativity axiom for +.

Let us do again (see Sect. 2.3) the proof of

(∀X)`X + (- X) = 0`

this time by rewriting modulo associativity. By the meta-rule of *Universal Quantification* this would be equivalent to proving

$$a + (-\, a) \;=\; 0 \tag{2.72}$$

in the signature of GROUP extended with a new constant a. By following the same conventions like for the GROUP examples of Sect. 2.3, let Σ denote the signature of GROUP and $\Sigma + a$ the extended signature mentioned above. Then the algebra in which rewriting is performed is the quotient $0_{\Sigma+a}/_{=_E}$ where $0_{\Sigma+a}$ is the initial (term) algebra of $\Sigma + a$, $=_E$ is the least E-congruence on $0_{\Sigma+a}$, and where E consists of the associativity equation for + only.

In the case of associativity the elements of the corresponding quotient algebras have a very intuitive representations, obtained by the elimination of the brackets related to + from representations of the terms as mix-fix expressions. For example, in $0_{\Sigma+a}/_{=_E}$ the expression a + (− a) + a stands for the equivalence class $\{(a + (-a)) + a,\ a + ((-a) + a)\}$. In fact this representation is used by many current implementations of rewriting modulo associativity, including the CafeOBJ and Maude rewriting engines.

Coming back to the proof of (2.72), let Γ denote the two equations [inv] and [id] to be used in rewriting. We have that:

$$((--a)+(-a))+(a+(-a)) \longrightarrow_\Gamma 0+(a+(-a)) \quad \text{by [inv]} \tag{2.73}$$

$$(0+a)+(-a) \longrightarrow_\Gamma a+(-a) \quad \text{by [id]} \tag{2.74}$$

$$(--a)+(((-a)+a)+(-a)) \longrightarrow_\Gamma (--a)+(0+(-a)) \quad \text{by [inv]} \tag{2.75}$$

$$(--a)+(0+(-a)) \longrightarrow_\Gamma (--a)+(-a) \quad \text{by [id]} \tag{2.76}$$

$$(--a)+(-a) \longrightarrow_\Gamma 0 \quad \text{by [inv]} \tag{2.77}$$

From (2.73) and (2.74) we have

$$(--a)+(-a)+a+(-a) \xrightarrow{\star}_{\Gamma,E} a+(-a) \tag{2.78}$$

and from (2.75), (2.76) and (2.77) we have

$$(--a)+(-a)+a+(-a) \xrightarrow{\star}_{\Gamma,E} 0 \tag{2.79}$$

From (2.78) and (2.79) by Cor. 2.73 and Cor. 2.74 we have that GROUP $= \Gamma \cup E \models_{\Sigma+a} a+(-a) = 0$.

A CafeOBJ proof score. Let us now see how the proof above can be coded into an actual language, namely in CafeOBJ. The first step, that consists of transforming the problem into a quantifier-free problem by the application of *Universal Quantification*, is coded as follows.

```
open GROUP .
  op a : -> G .
```

Now comes the proof of (2.78) but considered in an opposite direction. Note that in this case for each rewriting step we need to specify precisely the equation that is used, the position in the term, and eventually the substitution to be used. The latter is necessary in the case of the second rewrite since when [inv] is considered from the right to the left the variable X is not matched by anything.

```
start a + (- a) .
apply -.[id] at (1) .
apply -.[inv] with X = (- a) at [1] .
```

The partial result of executing this part of the proof score is $(--a)+(-a)+a+(-a)$. The final part of the proof score, corresponding to the proof of (2.79), consists of a simple rewriting command on this partial result that leads to an automatic execution (in the standard direction) of the steps (2.75), (2.76) and (2.77).

```
apply reduce at term .
close
```

The resulting term is 0.

Automation of proofs by rewriting. The proof `GROUP` $\vdash (\forall$`X`$)$`X + (- X) = 0` presented above is not automatic since one needs to 'guess' the expression $(--a)+(-a)+a+(-a)$ to be rewritten to both sides of the equation to be proved. This heuristic aspect depends upon a certain insight into the problem, which is a human mind aspect. Insight is beyond automation. The automation of rewriting is based upon a procedure related to Prop. 2.59 that is somehow contrary to the way the proof `GROUP` $\vdash (\forall$`X`$)$`X + (- X) = 0` has been performed. Therefore, in order to prove $\Gamma \vdash t_1 = t_2$, instead of finding a term t such that $t \overset{\star}{\longrightarrow}_{\Gamma} t_1$ and $t \overset{\star}{\longrightarrow}_{\Gamma} t_2$, we rather find t such that $t_1 \overset{\star}{\longrightarrow}_{\Gamma} t$ and $t_2 \overset{\star}{\longrightarrow}_{\Gamma} t$. The latter method has a huge advantage towards the former: t can be obtained automatically as a normal form of rewriting. However this needs some conditions that will be developed in the following.

Recall from Prop. 2.64 that the relation $\downarrow$ of defined for any preorder $>$ on A by

$$(b \downarrow c) \text{ if and only if } b > d \text{ and } c > d \text{ for some } d \in A.$$

is the least equivalence on A that contains $>$.

The typical application of the following technical result is by means of Prop. 2.70 when the role of $>$ is played by the relation $\overset{\star}{\longrightarrow}_{\Gamma,A}$ for A reachable algebra. This includes the case of rewriting modulo axioms, when A is the initial algebra of a set E of axioms. In the particular case of plain rewriting, when E is empty, then A is just the initial term algebra $0_{(S,F)}$.

Proposition 2.75. *Given a preordered (S,F)-algebra $(A,>)$, if $>$ is a confluent then $\downarrow$ is an (S,F)-congruence.*

Proof. By Prop. 2.64 we know that $\downarrow$ is equivalence. For showing the preservation of $\downarrow$ by the operations let σ be any operation symbol in F and $a_1,\dots,a_n$ and $a'_1,\dots,a'_n$ strings of arguments for A_σ such that $a_i \downarrow a'_i$ for each $i \in \{1,\dots,n\}$. For each i there exists a''_i such that $a_i > a''_i$ and $a'_i > a''_i$. Because A_σ preserves $>$ we have that $A_\sigma(a_1,\dots,a_n) > A_\sigma(a''_1,\dots,a''_n)$ and $A_\sigma(a'_1,\dots,a'_n) > A_\sigma(a''_1,\dots,a''_n)$ which shows that $A_\sigma(a_1,\dots,a_n) \downarrow A_\sigma(a'_1,\dots,a'_n)$. □

Note that although Prop. 2.75 does not require A to be reachable, however its applications for the rewriting relation $\xrightarrow{\star}_{\Gamma,A}$ does require the reachability of A for the reflexivity of $\xrightarrow{\star}_{\Gamma,A}$. Note also that in this case the preservation by operations condition of Prop. 2.75 is obtained from Prop. 2.70. These remarks are formally captured by the following Corollary.

Notation 2.76. *For any reachable (S,F)-algebra A and any set Γ of conditional (S,F)-equations let $\downarrow_{\Gamma,A}$ denote the least equivalence containing $\xrightarrow{\star}_{\Gamma,A}$ (see Prop. 2.64).*

Corollary 2.77. *On any reachable algebra A, if $\xrightarrow{\star}_{\Gamma,A}$ is confluent then $\downarrow_{\Gamma,A}$ is a congruence.*

Proof. By Prop. 2.70 and Prop. 2.75. □

Generalized soundness of rewriting. The soundness result of Prop. 2.59 can be extended from the initial (term) algebra to arbitrary reachable algebras by comparing $\downarrow_{\Gamma,A}$ to $a =^A_\Gamma a'$, the least Γ-congruence on A. One of the benefits of this generalization consists of applications to rewriting modulo axioms.

Proposition 2.78 (General soundness of rewriting)**.** *For any reachable (S,F)-algebra A and for any set Γ of conditional equations,*

$$a \downarrow_{\Gamma,A} a' \text{ implies } a =^A_\Gamma a'.$$

Proof. If $\Gamma \vdash^{r} t = t'$ then by the soundness result of Prop. 2.55 we have that $\Gamma \models t = t'$ which implies $A/_{=^A_\Gamma} \models t = t'$ which means $A_t =^A_\Gamma A_{t'}$. Since by definition $\xrightarrow{\star}_{\Gamma,A}$ is the transitive closure of $\{(A_t, A_{t'}) \mid \Gamma \vdash^{r} t = t'\}$ (see Dfn. 2.68) and by the transitivity of $=^A_\Gamma$ we have that $\xrightarrow{\star}_{\Gamma,A} \subseteq =^A_\Gamma$. By the symmetry and the transitivity of $=^A_\Gamma$ from this it follows immediately that $\downarrow_{\Gamma,A} \subseteq =^A_\Gamma$. □

The soundness of the equational proofs performed by rewriting modulo a set E of axioms to the same element is an instance of Prop. 2.78 when the role A is played by the initial algebra of E. However the same result can be obtained also from Prop. 2.57 and Cor. 2.74. We leave the rather straightforward proofs as a task to the reader.

Corollary 2.79. *For any set of equations E and any set Γ of conditional equations we have that*

$$(t/_{=E}) \downarrow_{\Gamma,E} (t'/_{=E}) \text{ implies } \Gamma \cup E \models t = t'.$$

Generalized completeness of rewriting. The completeness of the equational proof method by rewriting both sides of an equation to the same element corresponds to the inclusion $=^A_\Gamma \subseteq \downarrow_{\Gamma,A}$. This property is significantly harder than its dual, the soundness, which means that it needs some special conditions.

Proposition 2.80. *In a signature (S,F) let us consider a set Γ of conditional equations of the form $(\forall X)(H = \mathtt{true}) \Rightarrow (t = t')$ with $\mathtt{true}$ a constant. On any reachable (S,F)-algebra A, if $\xrightarrow{\star}_{\Gamma,A}$ is confluent and $A_{\mathtt{true}}$ is normal form for $\xrightarrow{\star}_{\Gamma,A}$ then $A/_{\downarrow_{\Gamma,A}} \models \Gamma$.*

Proof. Because $\stackrel{\star}{\longrightarrow}_{\Gamma,A}$ and A is reachable, cf. Cor. 2.77 the relation $\downarrow_{\Gamma,A}$ is indeed a congruence, hence the statement of the proposition is correctly formulated.

Now let $\stackrel{\star}{\longrightarrow}_{\Gamma,A}$ be a conditional equation from Γ. Let B' be any $(S, F \cup X)$-expansion of $A/_{\downarrow_{\Gamma,A}}$ such that $B'_H = B'_{\mathtt{true}}$. We consider any $(S, F \cup X)$-expansion of A such that $A'_x \in B'_x$ for each $x \in X$. Note that $B' = A'/_{\downarrow_{\Gamma,A}}$. We have that $A'_H/_{\downarrow_{\Gamma,A}} = B'_H = B'_{\mathtt{true}} = A'_{\mathtt{true}}/_{\downarrow_{\Gamma,A}}$. This implies $A'_H \downarrow_{\Gamma,A} A'_{\mathtt{true}}$. Since $A'_{\mathtt{true}} = A_{\mathtt{true}}$ is normal form for $\stackrel{\star}{\longrightarrow}_{\Gamma,A}$ it follows that $A'_H \stackrel{\star}{\longrightarrow}_{\Gamma,A} A'_{\mathtt{true}}$. By Prop. 2.69 this implies

$$E(A') \cup \Gamma \vdash^{\mathrm{r}}_{(S,F \cup X)} (H = \mathtt{true}). \tag{2.80}$$

By 'monotonicity' and *Universal Quantification* we have that

$$E(A') \cup \Gamma \vdash^{\mathrm{r}}_{(S,F \cup X)} (H = \mathtt{true}) \Rightarrow (t = t'). \tag{2.81}$$

From (2.80) and (2.81) by 'union', 'transitivity' and *Implication* we have that

$$E(A') \cup \Gamma \vdash^{\mathrm{r}}_{(S,F \cup X)} (t = t').$$

By Prop. 2.69 this implies $A'_t \stackrel{\star}{\longrightarrow}_{\Gamma,A} A'_{t'}$ which implies $A'_t \downarrow_{\Gamma,A} A'_{t'}$ which means $B'_t = B'_{t'}$. □

This way to handle the hypotheses of the equations as a quasi-Boolean term assumed by the conditions of Prop. 2.80 is rather common within the OBJ family of specification languages and has certain operational benefits. For example in CafeOBJ the constant `true` is the corresponding constant of the built-in Boolean type and the conditions of equations are encoded as Boolean terms by means of the Boolean function $\mathtt{==}_s$ as presented above in Sect. 2.1.

The following consequence of Prop. 2.80 can be regarded as a general completeness result for rewriting as a decision procedure for equations.

Theorem 2.81 (Completeness of rewriting)**.** *Under the conditions of Prop. 2.80, we have that* $\downarrow_{\Gamma,A} = =^A_\Gamma$.

Proof. By Prop 2.80 we have that $A/_{\downarrow_{\Gamma,A}} \models \Gamma$ which by Prop. 2.24 implies that $\downarrow_{\Gamma,A}$ is a Γ-congruence. Since $=^A_\Gamma$ is the *least* Γ-congruence on A it follows that $=^A_\Gamma \subseteq \downarrow_{\Gamma,A}$. The opposite inclusion is given by Prop. 2.78. □

The relation $\downarrow_{\Gamma,A}$ is realized in the OBJ family of languages by the built-in semantic equality predicate `==`, the same which is used for encoding conditions of equations as Boolean terms.

Exercises.

2.17. The rewriting relation defined by `BASIC-INT` (see Ex. 2.10) is terminating and confluent.

2.18. Give an example of a partial order $>$ on a set A such that each element of A has a unique normal form with respect to $>$ but which is *not* terminating.

2.19. 1. Give an example of a relation $\longrightarrow$ which is Noetherian but its reflexive and transitive closure $\stackrel{\star}{\longrightarrow}$ is *not* terminating.
2. If $\longrightarrow$ is Noetherian and for each element a the set $\{b \mid a \longrightarrow b\}$ is finite then $\stackrel{\star}{\longrightarrow}$ is terminating. Apply this result to rewriting relations.

2.5 Induction

Inductive properties. When verifying properties of data types specified as initial algebras of sets of conditional equations ordinary equational deduction may not be enough because initial algebras may satisfy more sentences than what can be deduced from the sentences of the specification. A very simple example is the specification SIMPLE-NAT of the natural numbers that we have already seen here several times.

```
mod! SIMPLE-NAT {
  [ Nat ]
  op 0 : -> Nat
  op s_ : Nat -> Nat
  op _+_ : Nat Nat -> Nat
  vars M N : Nat
  eq [succ] : N + (s M) = s(N + M) .
  eq [zero] : N + 0 = N .
}
```

The denotation of SIMPLE-NAT, that consists of the initial algebra satisfying of [succ] and [zero], is the algebra A of the natural numbers interpreting + as addition of numbers and s as the succesor function (adding 1 to a number). Since addition of numbers is commutative we have that $A \models (\forall m,n)m+n = n+m$. However the commutativity property is *not* a consequence of the two equations of SIMPLE-NAT because there exists algebras satisfying [succ] and [zero] that do *not* satisfy the commutativity property. The algebra B defined below gives a very simple example of such a situation.

- $B_{\mathtt{Nat}} = \{0,1\}$,
- $B_0 = 0$,
- $B_{\mathtt{s}}(x) = x$ for each $x \in \{0,1\}$, and
- $B_+(0,x) = 0$ and $B_+(1,x) = 1$ for each $x \in \{0,1\}$.

The sentences satisfied by the initial algebras of specifications are called *inductive properties*. A proof that a certain sentence is satisfied by an initial algebra of a set of conditional equations is called an *inductive proof*. We have seen above that equational deduction is not enough for proving inductive properties. Therefore in this section we focus on introducing a general method for inductive proofs on top of ordinary equational deduction.

Constructors. In may situations the elements of the initial algebras of specifications are denoted by terms that are constructed from a subset of the operation symbols of the signature. This is for example the case of our benchmark example, `SIMPLE-NAT`. The elements of the initial algebra A of `SIMPLE-NAT` are denoted only by terms formed from `0` and `s`, the operation + is not needed. Identifying a smallest possible such as subset of operations, called *constructors*, can greatly reduce the complexity of inductive proofs.

Definition 2.82 (Sub-signature of constructors)**.** *Given a signature (S,F) and Γ a set of conditional equations for (S,F), a sub-signature (S,F^c) of (S,F) (i.e. $F^c_{w\to s} \subseteq F_{w\to s}$ for all arities w and sorts s) is a* sub-signature of constructors for Γ *if and only if the unique (S,F^c)-homomorphism from the initial (term) (S,F^c)-algebra $0_{(S,F^c)}$ to the (S,F^c)-reduct of 0_Γ (the initial algebra satisfying Γ) is surjective.*

The following simple characterization for the sub-signatures of constructors, which sometimes in the literature is called *sufficient completeness*, constitutes a basis for actually proving the constructor property of Dfn. 2.82. Moreover, this characterization can be used as an alternative definition for sub-signature of constructors that can be used in more general contexts when Γ is any set of sentences (which means Γ may not have initial models).

Proposition 2.83. *(S,F^c) is a sub-signature of constructors for Γ if and only if for each (S,F)-term t there exists an (S,F^c)-term t' such that $\Gamma \models_{(S,F)} t = t'$.*

Proof. The elements of 0_Γ are equivalence classes of (S,F)-terms under $=_\Gamma$, the least Γ-congruence on $0_{(S,F)}$. Hence the constructor property of Dfn. 2.82 means that for each (S,F)-term t there exists an (S,F^c)-term t' that is interpreted in 0_Γ as the equivalence class of t, namely $t/_{=_\Gamma}$. This is the same as saying that $(t/_{=_\Gamma}) = (t'/_{=_\Gamma})$ which means $t =_\Gamma t'$ which is equivalent to $\Gamma \models t = t'$. □

The sufficient completeness property of Prop. 2.83 above can be shown by induction on the structure of the term t by skipping the constructors as follows.

Proposition 2.84. *(S,F^c) is a sub-signature of constructors for Γ if and only if for any operation symbol $\sigma \in F_{s_1\ldots s_n\to s} \setminus F^c_{s_1\ldots s_n\to s}$ and for any (S,F^c)-terms $t_1,\ldots,t_n$ of sorts $s_1,\ldots,s_n$, respectively, there exists an (S,F^c)-term t' such that $\Gamma \models \sigma(t_1,\ldots,t_n) = t'$.*

Proof. By Prop. 2.83 we have to show that each (S,F)-term t there exists an (S,F^c)-term t' such $\Gamma \models t = t'$. We show this by induction on the structure of t. Let $t = \sigma(t_1,\ldots,t_n)$ where σ is an operation symbol in F and t_i, for $1 \le i \le n$, are sub-terms. We assume that that for each $1 \le i \le n$ there exists an (S,F^c)-term t'_i such that $\Gamma \vdash^e t_i = t'_i$. By *Congruence* (for the semantic entailment) it follows that $\Gamma \models t = \sigma(t'_1,\ldots,t'_n)$. There are two cases:

1. When σ is a constructor then $\sigma(t'_1,\ldots,t'_n)$ is already an (S,F^c)-term.
2. When σ is *not* a constructor, let t' be an (S,F^c)-term such that $\Gamma \models \sigma(t'_1,\ldots,t'_n) = t'$. By *Transitivity* (for the semantic entailment) it follows that $\Gamma \models t = t'$.

□

Let us now apply the general method indicated by Prop. 2.84 to actually prove that `0` and `s` (of course together with the sort `Nat`) constitute a signature of constructors for `SIMPLE-NAT`. By the Soundness andf Completeness of equational deduction, in our proof below we may replace $\models$ by $\vdash^{e}$. According to Prop. 2.84 we have only to show that for any terms $t1$ and $t2$ formed only from 0 and s there exists a term t' formed only from 0 and s also such that $\Gamma \vdash^{e} t1+t2=t'$ where Γ consists of the equations [`succ`] and [`zero`] of `SIMPLE-NAT`. We can do this by induction on the structure of $t2$. There are two cases:

– $t2$ is `0`. In this case by [`zero`] we have $\Gamma \vdash^{e} t1+t2=t1$.

– $t2$ is $(\mathtt{s}\ t'2)$ with $t'2$ term formed from `s` and `0`. In this case by [`succ`] we have

$$\Gamma \vdash^{e} t1+t2=\mathtt{s}(t1+t'2). \tag{2.82}$$

By the induction hypothesis there exists a term t'' formed from `s` and `0` such that

$$\Gamma \vdash^{e} t1+t'2=t''. \tag{2.83}$$

From (2.83) by the *Congruence* and general entailment system properties we get that

$$\Gamma \vdash^{e} \mathtt{s}(t1+t'2)=(\mathtt{s}\ t''). \tag{2.84}$$

From (2.82) and (2.84) by *Transitivity* and general entailment system properties we get that

$$\Gamma \vdash^{e} t1+t2=(\mathtt{s}\ t'').$$

Since t'' is formed only from `0` and `s`, we have that $(\mathtt{s}\ t'')$ is also formed only from `0` and `s`.

Reducing inductive proofs to ordinary equational proofs. The following fundamental result, which constitutes the basis for inductive proofs, reduces the task of proving inductive properties to a set of ordinary equational proofs.

Proposition 2.85. *Let Γ be a set of conditional equations for a signature (S,F) and let (S,F^{c}) be a sub-signature of constructors for Γ. Let 0_Γ denote the initial algebra of Γ. Let E be any set of sentences such that $0_\Gamma \models E$. Then for any $(S,F\cup X)$-sentence ρ*

$$0_\Gamma \models (\forall X)\rho \ \textit{ if } \ \Gamma\cup E \models \theta(\rho) \ \textit{ for all substitutions } \ \theta : X \to T_{(S,F^c)}.$$

Proof. We assume the hypothesis $\Gamma\cup E \models \theta(\rho)$ for all substitutions $\theta : X \to T_{(S,F^c)}$. Let A' be any $(S,F\cup X)$-expansion of 0_Γ. Because (S,F^c) is a sub-signature of constructors for Γ this yields a substitution $\theta : X \to T_{(S,F^c)}$ such that $\theta(x)\in A'_x$ for each $x\in X$ (recall A'_x is a class of equivalent terms under $=_\Gamma$).

Since $0_\Gamma \models \Gamma$, $0_\Gamma \models E$ and $\Gamma \cup E \models \theta(\rho)$ we have that $0_\Gamma \models \theta(\rho)$. The proof of the proposition is completed if we proved that

$$0_\Gamma \models \theta(\rho) \text{ if and only if } A' \models \rho. \tag{2.85}$$

Let us do this by induction on the structure of ρ.

1. When ρ is $t1 = t2$ the property (2.85) holds by the observation that for each term t, $(0_\Gamma)_{\theta(t)} = A'_t$ (which can be shown by a simple induction on the structure of t).
2. When $\rho = \rho_1 \star \rho_2$, with $\star \in \{\wedge, \vee, \Rightarrow\}$ or $\rho = \neg\rho'$ the induction step is rather straightforward.
3. When $\rho = (\forall Y)\rho'$ we have to show that $0_\Gamma \models \theta((\forall Y)\rho') = (\forall Y)\theta(\rho')$ is equivalent to $A' \models (\forall Y)\rho'$. This follows from the induction hypothesis by noting the bijective correspondence between the the $(S, F \cup Y)$-expansions B of 0_Γ and the $(S, F \cup X \cup Y)$-expansions B' of A', correspondence determined by $B_y = B'_y$ for each $y \in Y$. Here the induction hypothesis means the property (2.85) considered for B instead of 0_Γ, which works because B is the initial algebra of Γ but considered as $(S, F \cup Y)$-sentences.

□

In practice the set E of Prop. 2.85 above plays the role of 'lemmas', and need not necessarily consist only of conditional equations, it can be a set of any sentences.

Structural induction. The big problem raised by the result of Prop. 2.85 is that one needs to perform infinitely many equational proofs corresponding to the substitutions θ. In order to be able to actually perform inductive proofs it is mandatory to have a finitary proof procedure for inductive properties. The standard one is the so-called *structural induction* method that can be presented as a property of semantic deduction, and when dealing only with conditional equations, of the equational entailment system $\vdash^{\mathrm{e}}$.

Proposition 2.86 (Structural induction)**.** *Let X be a finite set of variables for a signature (S, F) and let ρ be any $(S, F \cup X)$-sentence. Let (S, F^c) be a sub-signature of constructors for a set Γ of (S, F)-sentences (in the sense of the alternative definition given by Prop. 2.83).*

If for any sort preserving mapping $Q: X \to F^c$ (i.e. the sort of Q_x is the sort of x),

$$\Gamma \cup \{\psi(\rho) \mid \psi: X \to Z = \cup_{x \in X} Z_x \text{ with } \psi(x) \in Z_x\} \models_{(S, F \cup Z)} Q^\sharp(\rho)$$

where

- *Z_x are strings of variables for the arguments of Q_x such that $Z_{x1} \cap Z_{x2} = \emptyset$ for $x1 \neq x2 \in X$, and*
- *$Q^\sharp$ is the substitution $X \to T_{(S, F^c \cup Z)}$ defined by $Q^\sharp(x) = Q_x(Z_x)$,*

then

$$\Gamma \models_{(S,F)} \theta(\rho) \text{ for all substitutions } \theta: X \to T_{(S, F^c)}.$$

Proof. We prove the conclusion of the proposition by induction on the maximum depth of the set of terms $\{\theta(x) \mid x \in X\}$. Note that this maximum exists as a consequence of X being finite.

For each $x \in X$ let Q_x be the topmost operation symbol of the term $\theta(x)$ and T_x be the string of the immediate sub-terms of $\theta(x)$. In other words $\theta(x) = Q_x(T_x)$. By the hypothesis we have that

$$\Gamma \cup \{\psi(\rho) \mid \psi\colon\ X \to Z = \cup_{x\in X} Z_x \ \text{ with } \ \psi(x) \in Z_x\} \models_{(S,F\cup Z)} Q^\sharp(\rho).$$

Because X is finite we have that the set $\{\psi\colon\ X \to Z = \cup_{x\in X} Z_x \ \text{ with } \ \psi(x) \in Z_x\}$ is finite too. By *Implication* it follows that

$$\Gamma \models_{(S,F\cup Z)} \bigwedge\{\psi(\rho) \mid \psi\colon\ X \to Z = \cup_{x\in X} Z_x \ \text{ with } \ \psi(x) \in Z_x\} \Rightarrow Q^\sharp(\rho).$$

By *Universal Quantification* it follows that

$$\Gamma \models_{(S,F)} (\forall Z) \bigwedge\{\psi(\rho) \mid \psi\colon\ X \to Z = \cup_{x\in X} Z_x \ \text{ with } \ \psi(x) \in Z_x\} \Rightarrow Q^\sharp(\rho).$$

By *Substitutivity* for the substitution $Z \to T_{(S,F^c)}$ mapping componentwise each Z_x to T_x and by the 'transitivity' of $\models_{(S,F)}$ it follows that

$$\Gamma \models_{(S,F)} \bigwedge\{\gamma(\rho) \mid \gamma\colon\ X \to T_{(S,F^c)} \ \text{ with } \ \gamma(x) \in T_x\} \Rightarrow \theta(\rho).$$

By *Implication* it follows that

$$\Gamma \cup \{\gamma(\rho) \mid \gamma\colon\ X \to T_{(S,F^c)} \ \text{ with } \ \gamma(x) \in T_x\} \models_{(S,F)} \theta(\rho). \tag{2.86}$$

Because each $\gamma(x) \in T_x$ we have that for each substitution γ as in the relation (2.86) above the maximum depth of the terms $\{\gamma(x) \mid x \in X\}$ is strictly less than the maximum depth of the terms $\{\theta(x) \mid x \in X\}$, hence we can apply the induction hypothesis, meaning

$$\Gamma \models_{(S,F)} \gamma(\rho). \tag{2.87}$$

Because the set $\{\gamma\colon\ X \to T_{(S,F^c)} \mid \gamma(x) \in T_x\}$ is finite, by the entailment system properties of $\models$ from (2.86) and (2.87) it follows that

$$\Gamma \models_{(S,F\cup Y)} \theta(\rho).$$

□

In the proof of Prop. 2.86 above the meta-rules of *Implication* and *Universal Quantification*, as well as the rule of *Substitutivity* have been considered for the semantic entailment $\models$ in a slightly more general form than introduced in Sect. 2.3, i.e. for any sentences instead of only for conditional equations. It is easy to check that the above mentioned rule and meta-rules hold within this extended framework, the proof being similar to the case of conditional equations.

Corollary 2.87. *If in Prop. 2.86 we consider Γ to be a set of conditional equations and ρ to be a conditional equation, then by the Soundness (Prop. 2.49) and Completeness (Thm. 2.50) of equational deduction, in Prop. 2.86 we may replace $\models$ by $\vdash^{\mathrm{e}}$.*

By noting that Γ in Prop. 2.86 plays the role of $\Gamma \cup E$ in Prop. 2.85, this means that the hypothesis of the structural induction Prop. 2.86 constitutes a sufficient condition for the inductive property $0_\Gamma \models_{(S,F)} (\forall X)\rho$. The finitary character of proofs by structural induction resides in the fact that if the sub-signature of constructors (S, F^c) is finite then because also of the finiteness of X there is only a finite number of equational proofs to be performed, this number being equal to the number of the mappings $Q : X \to F^c$ from Prop. 2.86. Note that getting a sub-signature of constructors as small as possible reduces the complexity of the structural induction proofs. Note also that structural induction as formalized by Prop. 2.86 is based upon the well known Peano induction for natural numbers, which is the simplest and the most basic form of induction, and which is a proof 'principle' that is beyond formal provability and which is generally assumed through all bodies of mathematics from elementary school mathematics to the most advanced areas of mathematical investigation.

An example of inductive proof by structural induction. Let us prove that the initial algebra of `SIMPLE-NAT` satisfies the commutativity of addition, $(\forall m,n)m+n = n+m$. Let Σ denote the signature and Γ denote the two sentences `[succ]` and `[zero]` of `SIMPLE-NAT`. We have proved above in this section that `0` and `s` form a sub-signature of constructors for Γ. We use the following two lemmas, both of them being proved by the structural induction method of Prop. 2.86:

$$0_\Gamma \models (\forall n)0+n = n. \tag{2.88}$$

$$0_\Gamma \models (\forall m,n)\mathtt{s}(m)+n = \mathtt{s}(m+n). \tag{2.89}$$

The proof of lemma (2.88): With the notations of Prop. 2.86, we let the set X of the variables $\{n\}$. Because `0` and `s` form a sub-signature of constructors for Γ, we have only two possibilities for the mapping $Q : X \to F^c$:

1. $Q_n = \mathtt{0}$, in which case according to Prop. 2.86 we have to prove that

   $$\Gamma \vdash^{\mathrm{e}}_{\Sigma} \mathtt{0}+\mathtt{0} = \mathtt{0}.$$

   Its proof consists of one rewriting step, namely $\mathtt{0}+\mathtt{0} \longrightarrow_\Gamma \mathtt{0}$.

2. $Q_n = \mathtt{s}$, in which case according to Prop. 2.86 we have to prove that

   $$\Gamma \cup \{\mathtt{0}+z = z\} \vdash^{\mathrm{e}}_{\Sigma+z} \mathtt{0}+\mathtt{s}(z) = \mathtt{s}(z).$$

   Its proof consists of two rewriting steps, namely $\mathtt{0}+\mathtt{s}(z) \longrightarrow_\Gamma \mathtt{s}(\mathtt{0}+z) \longrightarrow_{0+z=z} \mathtt{s}(z)$.

Note that the set Z of the variables as in Prop. 2.86 is $\emptyset$ for the first proof and $\{z\}$ for the second proof.

The proof of lemma (2.89): We let the parameters of Prop. 2.86, be $X = \{n\}$ and ρ be $(\forall m)\mathtt{s}(m)+n = \mathtt{s}(m+n)$. As in the proof of lemma (2.88) we have only two possibilities for the mapping $Q : X \to F^c$:

1. $Q_n = \mathtt{0}$, in which case according to Prop. 2.86, by the meta-rule of *Universal Quantification*, we have to prove that

$$\Gamma \vdash^{\mathrm{e}}_{\Sigma+m} \mathtt{s}(m) + \mathtt{0} = \mathtt{s}(m + \mathtt{0}).$$

   Its proof consists of the following two rewritings:

$$\mathtt{s}(m) + \mathtt{0} \longrightarrow_\Gamma \mathtt{s}(m) \text{ and } \mathtt{s}(m + \mathtt{0}) \longrightarrow_\Gamma \mathtt{s}(m)$$

   implying $(\mathtt{s}(m) + \mathtt{0}) \downarrow_\Gamma \mathtt{s}(m + \mathtt{0})$.

2. $Q_n = \mathtt{s}$, in which case according to Prop. 2.86, by the meta-rule of *Universal Quantification*, we have to prove that

$$\Gamma \cup \{(\forall M)\mathtt{s}(M) + z = \mathtt{s}(M + z)\} \vdash^{\mathrm{e}}_{\Sigma+m+z} \mathtt{s}(m) + \mathtt{s}(z) = \mathtt{s}(m + \mathtt{s}(z)).$$

   Its proof consists of the following two rewritings:

$$\mathtt{s}(m) + \mathtt{s}(z) \longrightarrow_\Gamma \mathtt{s}(\mathtt{s}(m) + z) \longrightarrow_{(\forall M)\mathtt{s}(M)+z=\mathtt{s}(M+z)} \mathtt{s}(\mathtt{s}(m + z)) \text{ and}$$

$$\mathtt{s}(m + \mathtt{s}(z)) \longrightarrow_\Gamma \mathtt{s}(\mathtt{s}(m + z))$$

   implying $(\mathtt{s}(m) + \mathtt{s}(z)) \downarrow_{\Gamma \cup \{(\forall M)\mathtt{s}(M)+z=\mathtt{s}(M+z)\}} \mathtt{s}(m + \mathtt{s}(z))$.

Note that the set Z of the variables as in Prop. 2.86 is the same as in the proof of lemma (2.88), namely $\emptyset$ for the proof corresponding to $Q_n = \mathtt{0}$ and $\{z\}$ for the proof corresponding to $Q_n = \mathtt{s}$.

The proof of $0_\Gamma \models (\forall m,n) m + n = n + m$*:* Now let E be the set consisting of the two sentences of the lemmas (2.88) and (2.89). We apply Prop. 2.86 with $\Gamma \cup E$ in the role of Γ, with $X = \{n\}$ and with ρ being $(\forall m) m + n = n + m$. Like in the proofs of the lemmas (2.88) and (2.89), here we have two possibilities for the mapping $Q : X \to F^c$:

1. $Q_n = \mathtt{0}$, in which case according to Prop. 2.86, by the meta-rule of *Universal Quantification*, we have to prove that

$$\Gamma \cup E \vdash^{\mathrm{e}}_{\Sigma+m} m + \mathtt{0} = \mathtt{0} + m.$$

   Its proof consists of the following two rewritings:

$$m + \mathtt{0} \longrightarrow_\Gamma m \text{ and } \mathtt{0} + m \longrightarrow_E m$$

   implying $(m + \mathtt{0}) \downarrow_{\Gamma \cup E} (\mathtt{0} + m)$.

2. $Q_n = \mathtt{s}$, in which case according to Prop. 2.86, by the meta-rule of *Universal Quantification*, we have to prove that

$$\Gamma \cup E \cup \{(\forall M)M + z = z + M\} \vdash^{\mathrm{e}}_{\Sigma + m + z} m + \mathtt{s}(z) = \mathtt{s}(z) + m.$$

Its proof consists of the following two rewritings:

$$m + \mathtt{s}(z) \longrightarrow_{\Gamma} \mathtt{s}(m+z) \longrightarrow_{\{(\forall M)M+z=z+M\}} \mathtt{s}(z+m) \text{ and}$$

$$\mathtt{s}(z) + m \longrightarrow_{E} \mathtt{s}(z+m)$$

implying $m + \mathtt{s}(z) \downarrow_{\Gamma \cup E \cup \{(\forall M)M+z=z+M\}} \mathtt{s}(z) + m$.

Note that the set Z of the variables as in Prop. 2.86 is the same as in the proof of lemmas (2.88) and (2.89), namely $\emptyset$ for the proof corresponding to $Q_n = \mathtt{0}$ and $\{z\}$ for the proof corresponding to $Q_n = \mathtt{s}$.

A CafeOBJ proof score. Now we present a translation of the structural induction proof of $(\forall)m + n = n + m$ as an inductive property of `SIMPLE-NAT` that we developed above, into a proof score coded in CafeOBJ. The CafeOBJ system will perform the rewritings automatically while we will specify the *proof score*, i.e. the sequence of proof tasks to be executed by the system. Except the lemmas, the proof tasks are derived from Prop. 2.86, which implies that in principle they can also be introduced automatically by the system; however the core CafeOBJ language does not have this facility. Our CafeOBJ proof score follows closely the proof of $(\forall m,n)m + n = n + m$ presented above, including the two lemmas involved.

The relation $\downarrow$ is denoted in CafeOBJ by the binary operation of Boolean sort `==`, which means that our proof tasks will be given as t`==`t'. When CafeOBJ runs the proof score below these reductions will all produce the result true, meaning the the respective relation $t \downarrow t'$ holds, which implies that $t = t'$ holds.

Proof score for lemma (2.88):

```
open SIMPLE-NAT
```

The case $Q_n = \mathtt{0}$:

```
  red 0 + 0 == 0 .
```

The case $Q_n = \mathtt{s}$:

```
  op z : -> Nat .
  eq 0 + z = z .
  red 0 + (s z) == s z .
close
```

Proof score for lemma (2.89):

```
open SIMPLE-NAT
```

The case $Q_n = 0$:

```
  op m : -> Nat .
  red (s m) + 0 == s(m + 0) .
```

The case $Q_n = \mathtt{s}$:

```
  op z : -> Nat .
  eq (s M:Nat) + z = s(M + z) .
  red (s m) + (s z) == s(m + (s z)) .
close
```

The proof score of $(\forall m,n)m+n = n+m$*:*

```
open SIMPLE-NAT
```

We introduce lemmas (2.88) and (2.89):

```
  eq 0 + N:Nat = N:Nat .
  eq (s M:Nat) + N:Nat = s (M + N) .
```

The case $Q_n = 0$:

```
  op m : -> Nat .
  red m + 0 == 0 + m .
```

The case $Q_n = \mathtt{s}$:

```
  op z : -> Nat .
  eq M:Nat + z = z + M .
  red m + (s z) == (s z) + m .
close
```

Another example. The following is an application of the structural induction method as given by Prop. 2.86 to a situation when induction is done simultaneously on more than one variable. Let us consider the following specification of natural numbers with a semantic equality relation.

```
mod! PNAT= {
  [ Nat ]
  op 0 :  -> Nat
  op s_ : Nat -> Nat
  op _=_ : Nat Nat -> Bool {comm}
  vars M N : Nat
  eq ((s M) = 0) = false .
  eq (0 = 0) = true .
  eq (s M = s N) = (M = N) .
}
```

The following defines a strict ‘less than’ relation on the natural numbers.

```
mod! PNAT< {
  protecting(PNAT=)
  op _<_ : Nat Nat -> Bool
  vars M N : Nat
  eq 0 < s M = true .
  eq M < 0 = false .
  eq (s M < s N) = M < N .
}
```

The proof score by structural induction (by Prop. 2.86) of the total order property

$$(\forall \mathtt{M,N})(\mathtt{M < N) \ or \ (N < M) \ or \ (M = N)}$$

as an inductive property considers `0`, `s`, `true`, and `false` as a sub-signature of constructors (we exile this fact to the exercise part of the section) and considers four cases for Q:

```
open PNAT< .
  ops m n :  -> Nat .
```

The case $Q_{\mathrm{M}} = \mathtt{0}$, $Q_{\mathrm{N}} = \mathtt{0}$:

```
red (0 < 0) or (0 < 0) or (0 = 0) .
```

The case $Q_{\mathrm{M}} = \mathtt{0}$, $Q_{\mathrm{N}} = \mathtt{s}$:

```
red (0 < s n) or (s n < 0) or (0 = s n) .
```

The case $Q_{\mathrm{M}} = \mathtt{s}$, $Q_{\mathrm{N}} = \mathtt{0}$:

```
red (s m < 0) or (0 < s m) or (s m = 0) .
```

The case $Q_{\mathrm{M}} = \mathtt{s}$, $Q_{\mathrm{N}} = \mathtt{s}$:

```
  eq (m < n) or (n < m) or (m = n) = true .
red (s m < s n) or (s n < s m) or (s m = s n) .
close
```

Note that the only case that involves a premise is the fourth one; this is because in all other cases there is no $\psi : X \to Z = \cup_{x \in X} Z_x$ with $\psi(x) \in Z_x$ because either Z_{M} or Z_{N} is empty since either Q_{M} or Q_{N} is `0`.

Exercises.

2.20. Prove a that Prop. 2.85 admits a reciprocal for the case when ρ is a finite conjunction of equations.

2.21. For the specification `SIMPLE-NAT` show that the associativity of the addition `+` is not an (ordinary) equational consequence of the two sentences of the specification, but however it is an inductive property of the specification. Build and run a CafeOBJ proof score of this inductive property.

2.22. For the specification `PNAT<` above:

1. Show that `0`, `s_`, `true` and `false` form a signature of constructors.
2. Formulate the transitivity property of `_<_` and show that it cannot be proved by equational deduction from the axioms of `PNAT<`.
3. Build and run a proof score for the transitivity of `_<_` as inductive property.

2.23. Consider the following CafeOBJ specification:

```
mod! PNAT- {
  protecting(PNAT=)
  op _-_ : Nat Nat -> Nat
  vars M N : Nat
  eq (s M) - (s N) = M - N .
  eq M - 0 = M .
  eq 0 - M = 0 .
}
mod! GCD {
  protecting(PNAT- + PNAT<)
  op gcd : Nat Nat -> Nat
  vars M N : Nat
  eq gcd(M,0) = M .
  eq gcd(0,N) = N .
  eq gcd(M,M) = M .
  cq gcd(M,N) = gcd(M - N,N) if (0 < N) and (N < M) .
  cq gcd(M,N) = gcd(M,N - M) if (0 < M) and (M < N) .
}
```

Prove the inductive property $(\forall \mathtt{M},\mathtt{N})\mathtt{gcd(M,N)} = \mathtt{gcd(N,M)}$ as follows:

1. Extend the given specification with the specification of addition + on `Nat`.
2. Prove that `0`, `s`, `true` and `false` is a subsignature of constructors for the extended specification.
3. Build a run a proof score for the lemma $(\forall \mathtt{X},\mathtt{M},\mathtt{N})\mathtt{gcd(M,N)} = \mathtt{gcd(N,M)}$ if `M + N < X`.
4. Build a run a proof score showing that the lemma of the above item implies the goal property.

2.24. Build a proof score for the associativity of `gcd` of Ex. 2.23 by the following steps:

1. Add a specification for a binary 'divides' predicate `_div_`.
2. Prove and use the following main lemmas:
   - $(\forall \mathtt{X},\mathtt{M},\mathtt{N})$ `gcd(M,N) div M` if `M + N < X`.
   - $(\forall \mathtt{X},\mathtt{Y},\mathtt{M},\mathtt{N})$ `Y div gcd(M,N)` if `Y div M` and `Y div N` and `M + N < X`.

2.6 Subsorts.

Partial versus total functions. In computing science we often encounter situations when for certain arguments some operations do not return any result. There can be different reasons for this such as non-termination or some error in the computation process.

These two situations are rather familiar to programmers. Let us say that an operation is 'partial' when it does not give a result for some of the arguments, otherwise we say that the operation is 'total'.

Also partiality may occur naturally for some data types, for example division of numbers by 0 is not defined. The same happens with the head of empty lists. Thus division and head should be thought as partial rather than total operations.

It is useful to understand mathematically the concept of partial function. For this we have to recall the mathematical definition of the concept of (total) function within axiomatic set theory. Thus a function $f:\ A \to B$ is a binary relation $f \subseteq A \times B$ that satisfies the following two conditions:

1. For each $x \in A$ there exists $y \in B$ such that $f(x,y)$.
2. If $f(x,y)$ and $f(x,y')$ then $y = y'$.

The second condition allows us to use the functional notations $f(x) = y$ instead of the relational notation $f(x,y)$. Then the concept of *partial* function is obtained by dropping off the first condition above and keeping only the second condition. The set $\{x \in A \mid$ there exists $y \in B$ such that $f(x) = y\}$ is called the domain of f and is denoted by $dom(f)$.

Specifications with partial functions have the advantage of greater expressivity but the disadvantages of weaker computational properties and of more complex mathematical foundations. With respect to the latter issue, it is possible to refine the mathematics of equational logic from total to partial functions; this is called *partial algebra* theory. This refinement includes the concepts of signature, algebra, equation, satisfaction, and results such as the existence of initial algebra for a set of conditional equations and a sound and complete proof calculus for conditional equations. There are two schools of thinking in algebraic specification: one advocates staying with total functions while the other favours the specifications with partial functions. For example, among the modern algebraic specification languages, CafeOBJ and Maude belong to the former school, while CASL belongs to the latter.

Our lecture notes are rooted within the trend of total functions. The readers interested to find more about partial algebra may look into classical works such as [25, 4] or at the more recent CASL literature.

Order sorted algebra. The total functions approach has had to find various ways to deal with cases of partiality. Although one can never reach the expressivity power of partial algebra by means of total algebra, some special cases of partiality, that cover a great deal of specification needs, can be handled by the total function approach. One solution is to extend the data type with new elements for errors or undefinedness, which may lead to the necessity to extend the definition of the ordinary data type operations to handle the new error values. On the other hand many partiality situations may be handled directly by noticing that the domain of the respective partial operations can be specified by the use of the so-called 'subsorts'. For example division of numbers by 0 falls in this latter category, we have just to consider the subsort of the non-zero numbers for the second argument of the division operation. The same may be done for the head of lists

(List with elements from a sort Elt) by considering the subsort of the non-empty lists (NList).

```
op head : NList -> Elt
```

Moreover, the concept of subsort may be used to handle error values through the so-called 'error supersort' approach.

```
op head : List -> ?Elt
```

In the former case NList is a subsort of List while in the latter case Elt is a subsort of ?Elt.

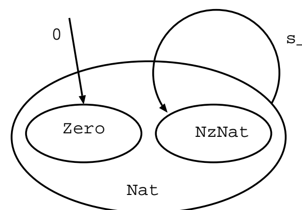
The theory of many sorted algebra and many sorted equational logic, including all the concepts and results developed until this point in the chapter, has been extended to the refined framework using subsorts in a way that parallels many sorted algebra; this is called *order sorted algebra* or *order sorted equational logic*. However this extension has some very subtle points that have led to slightly different formalisms (a survey on this matter is [18]). In this section we refrain from developing order sorted algebra in detail, and instead we present its basic concepts by emphasizing methodologies of using subsorts. This may be enough for our use of subsorts in these lecture notes.

An example. The following simple specification of natural numbers only with zero and succesor that uses subsorts realizes the idea that the succesor operation never gives zero as result.

```
mod! BARE-NAT {
  [ NzNat Zero < Nat ]
  op 0 : -> Zero
  op s_ : Nat -> NzNat
}
```

In the specification BARE-NAT we have used NzNat as a sort name for the subset of the non-zero naturals and Zero as a sort name for the subset containing only the element zero (0). Because both NzNat and Zero are meant to name subsets of the natural numbers, named by the sort Nat), they are called *subsorts* of Nat.

As the reader may guess by looking at BARE-NAT, in CafeOBJ subsort relationship is specified by using the 'less than' symbol < inside a sort declaration. The graphical representation of signatures with subsorts also upgrades the graphical convention that we have introduced for the signatures by representing subsorting as disk inclusion. For example the following is a graphical representation for the signature of BARE-NAT.



Order sorted signatures. In general, if s is a subsort of s', then $A_s \subseteq A_{s'}$ for any algebra A of the respective signature. The fact that subset relationship is a partial order is reflected in the fact that subsort relationship is considered to be a partial order too. Thus when considering signatures with subsorts we have to upgrade the set of sort symbols S to a partial order $(S, \leq)$ of sort symbols. Such signatures are called *order sorted signatures*. The following is the mathematical definition for this concept.

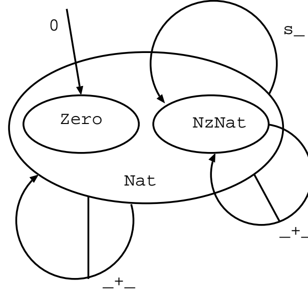
Definition 2.88 (Order sorted signature)**.** *An* order sorted signature *is a triple* $(S, \leq, F)$ *such that* (S, F) *is a many sorted signature, and* $(S, \leq)$ *is a partially ordered set.*

An order sorted signature is monotone *iff for any non-empty arities* w_1 *and* w_2 *and any sorts* s_1 *and* s_2

$$\sigma \in F_{w_1 \to s_1} \cap F_{w_2 \to s_2} \quad \textit{and} \ \ w_1 \leq w_2 \ \textit{(component-wise) imply} \ s_1 \leq s_2.$$

The following is an example of a monotone order sorted signature with the operation _+_ specified twice with different arities and sorts:

```
op _+_ : Nat Nat -> Nat
op _+_ : NzNat Nat -> NzNat
```



Algebras of order sorted signatures. The coherence of the interpretations of the overloaded operation symbols is taken care by the monotonicity condition in the following definition.

Definition 2.89. *Given an order sorted signature* $(S, \leq, F)$*, an* $(S, \leq, F)$-algebra *is just an* (S, F)*-algebra* A *such that* $s \leq s'$ *in* S *implies* $A_s \subseteq A_{s'}$*. An* $(S, \leq, F)$*-algebra* A *is* monotone *when*

> $\sigma \in F_{w_1 \to s_1} \cap F_{w_2 \to s_2}$ *and* $w_1 \leq w_2$ *and* $s_1 \leq s_2$ *imply that* $A_{\sigma : w_1 \to s_1} : A_{w_1} \to A_{s_1}$ *equals* $A_{\sigma : w_2 \to s_2} : A_{w_2} \to A_{s_2}$ *on* A_{w_1}.

For example, the standard algebra of the natural numbers (interpreting + as addition) is an example of a monotone algebra for the order sorted signature given above as example.

Weak versus strong overloading. Given an order sorted signature $(S, \leq, F)$, the interpretations of an overloaded operation symbol $\sigma \in F_{w_1 \to s_1} \cap F_{w_2 \to s_2}$ in an algebra A need not necessarily agree on elements that belong to the intersection of carriers for w_1 and

w_2; thus, a strong form of overloading is supported. For this reason, in the literature this approach is called *overloaded order sorted algebra*.

It is useful to distinguish between two forms of overloading: in *weak overloading*, operation symbols may have more than one arity and sort, so that some expressions can be typed in more than one way, but a given (well formed) expression can only have one value in a given algebra; however, in *strong overloading* expressions can have different values. For example, weak overloading would allow the expression '`3 + 5`' to be interpreted as addition of naturals, integers, or rationals, each of which gives the same value `8`, but it would not also allow addition of integers modulo `8`, where the result would be `0`; however, strong overloading would allow both of these.

Strong overloading is important for ordinary mathematical notation. For example, the value of an expression like '`1 + 1`' depends on how it is parsed. The result can be `2` when it is parsed within natural numbers or it can be `0` when it is parsed within integers modulo `2`. Vector spaces provide a similar example. Thus, expressions like '`0 + 0`' are ambiguous, becasue `0` is used for both the zero vector and the real `0`. However such ambiguous expressions do not occur often in practice.

Non-monotonicities. Notice that in the monotone case, if we take `Z8`, for the integers modulo `8`, to be a subsort of the naturals, then `_+_ : Z8 Z8 -> Z8` cannot be interpreted as addition modulo `8` in an algebra where `_+_ : Nat Nat -> Nat` is addition of naturals, because it must agree with addition of natural numbers on `Z8`, and natural number addition does not restrict to `Z8`. But there is no such difficulty if we remove the restriction of monotonicity. Alternatively, we could retain monotonicity but remove the subsort relation `Z8 < Nat`. Either way we get strong overloading, but the non-monotonic approach provides a more faithful modelling of mathematical practice, because the subsort relation can ensure that the elements of sort `Z8` really are the usual numbers `0`,...,`7`.

On the other hand, non-monotonicities are rare enough in practice to be considered exceptional, and we should not build a theory that fails to handle the most common case in a smooth way. The solution to this dilemma is a mechanism for declaring which operation declarations should be considered non-monotonic. This leads to another level of refinement of the mathematical concept of signature that the interested reader can find [18]. All standard concepts and results of many sorted algebra (such as congruence, term, deduction, initial and free algebras, completeness, etc.) carry through for signatures with non-monotonicities but that respect the following condition.

Definition 2.90 (Locally filtered signature)**.** *A partially ordered set $(S,\leq)$ is (upward)* filtered *iff for any two elements $s,s' \in S$ there is an element $s'' \in S$ such that $s,s' \leq s''$. A partially ordered set S is* locally filtered *iff each of its connected components*[5] *is filtered. An order sorted signature $(S,\leq,F)$ is* locally filtered *iff $(S,\leq)$ is locally filtered.*

[5]Given a poset $(S,\leq)$, let $\equiv$ denote the transitive and symmetric closure of $\leq$. Then $\equiv$ is an equivalence relation whose equivalence classes are called the *connected components* of $(S,\leq)$.

Dynamic type checking. One of the benefits of an accurate specification style making use of subsorts is that it may give very precise information on the type of the elements by computing their smallest sorts. The CafeOBJ command `parse` not only shows whether a term is well formed, but it also gives its smallest sort syntactically. For example

```
BARE-NAT> parse s s 0 .
```

gives the answer

```
(s (s 0)) : NzNat
```

However the syntactic information on the type of elements may be less accurate than the semantic one. Take the following case with rational numbers (with `Rat` the sort of the rationals and `NzRat` the subsort of the non-zero rationals).

```
RAT> parse 2 / 2 .
```

gives

```
(2 / 2) : NzRat
```

but when evaluating this expression by the CafeOBJ command `red`:

```
RAT> red 2 / 2 .
```

we get

```
-- reduce in RAT : 2 / 2
1 : NzNat
```

which means that one of the effects of the computation process is that the system finds a sort of the respective element which may be smaller than the sort resulting from the parsing process. This is called *dynamic type checking*.

Error supersorts. When attempting to evaluate a partial function for values of the arguments not belonging to its domain, the CafeOBJ system indicates the error. For example

```
RAT> parse 2 / 0 .
```

gives

```
(2 / 0) : ?Rat
```

The error message comes as an *error supersort*, `?Rat` in this case. 'Supersort' means just that `RAT < ?RAT`. The main function of error supersorts is that they store values corresponding to some of the ill-formed expressions outside of the ordinary sorts. Which are precisley these ill-formed expressions and why do we need them? These are pseudo-terms that are defined by the following rules:

– each constant symbol of sort s is a pseudo-term of sorts s, and

– for each operation symbol $\sigma \in F_{s_1 \ldots s_n \to s}$ and any pseudo-terms of $t_1, \ldots t_n$ of sorts $s'_1 \ldots s'_n$, respectively, $\sigma(t_1, \ldots, t_n)$ is a pseudo-term of sort s when for each $k \in \{1, \ldots, n\}$ the sorts s_k and s'_k have a common subsort.

Some of the pseudo-terms, although they are ill-formed syntactically, may still evaluate to values corresponding to ordinary terms. `2 / ((3 / 2) + (1 / 2))` is such an example. This is the reason we should not discard them completely.

The error supersorts may sometimes be used for the purpose of specification; we will see such example below. In CafeOBJ the error supersorts are built into the system, meaning that they are made automatically available to the user.

Retracts. The mechanism handling the pseudo-terms consists of introducing a operations $r_{s'>s}: s' \to s$ for each subsort relationship $s < s'$ which are subject to the following equations:

- $(\forall x : s) r_{s'>s}(x) = x$, and
- $(\forall x : s'') r_{s'>s}(r_{s''>s'}(x)) = r_{s''>s}(x)$.

These operations are called *retracts* and they are inserted in pseudo-terms in places where the sorts of the subterms do not correspond to the arity of the respective operation. Such insertion transforms the pseudo-terms into proper terms of the signature extended with retracts. For example the pseudo-term `2 / 0` is transformed into the term `2 /` $r_{\mathtt{Rat}>\mathtt{NzRat}}$ `(0)`. The term `2 / 0` is an example of proper pseudo-term because the inserted retract cannot be elimintated in a computation process.

A different situation is with `2 / ((3 / 2) + (1 / 2))`, in which the dynamic type checking shows its strength. The parsing gives an error although semantically it should not be the case.

```
RAT> parse 2 / ((3 / 2) + (1 / 2)) .
(2 / ((3 / 2) + (1 / 2))) : ?Rat
```

The source of this error is due to the static aspect of the parsing process, the sort of `(3 / 2) + (1 / 2)` being computed `Rat` rather than `NzRat`. In fact if we were to think of

```
2 / ((3 / 2) + (1 / 2))
```

as a proper term we would have to consider the insertion of retracts, in reality the term being

`2 /`$r_{\mathtt{Rat}>\mathtt{NzRat}}$`((3 / 2) + (1 / 2))`.

During the evaluation of this term, using the following CafeOBJ command

```
RAT> red 2 / ((3 / 2) + (1 / 2)) .
```

at the stage when it becomes

`2 /`$r_{\mathtt{Rat}>\mathtt{NzRat}}$`(1)`

by applying the equation $(\forall x : \mathtt{NzRat}) r_{\mathtt{Rat}>\mathtt{NzRat}}(x) = x$ the retract operation gets eliminated and the computation process continues for getting the following result

```
-- reduce in RAT : 2 / (3 / 2 + 1 / 2)
1 : NzNat
```

Thus retracts provide more strength to dynamic type checking. Retract operations are also built into the CafeOBJ system, however they are not transparent to the user unless the user requires explicitly this.

The consistency of using retracts. Within the context of specifications with initial denotation, the use of retracts may pose the following consistency problem. Any retract operation $r_{s'>s}$ may introduce new (undesired) elements on the sort s. However this is a problem only when this affects the 'old' elements, in the sense that different elements may get identified by the use of retracts. This could happen for example if the user wrote an equation such as $r_{\texttt{Rat}>\texttt{NzRat}}(1) = 2$.

Definition 2.91 (Consistency of retracts)**.** *Let Γ be a set of conditional equations for an order sorted signature $(S,\leq,F)$ and let $\Gamma^{\otimes}$ be an extension of Γ to set of conditional equations for the extension $(S,\leq,F^{\otimes})$ with retracts. We say that $\Gamma^{\otimes}$ is* consistent with respect to Γ *when the unique $(S,\leq,F)$-homomorphism from (the initial $((S,\leq,F),\Gamma)$-algebra) $0_{(S,\leq,F),\Gamma}$ to the reduct of (the initial $((S,\leq,F^{\otimes}),\Gamma^{\otimes})$-algebra) $0_{(S,\leq,F^{\otimes}),\Gamma^{\otimes}}$ to $(S,\leq,F)$ is injective.*

Adauga ceva despre sort constraints??

Exercises.

2.25. Specify a data type of lists of natural numbers with the Lisp-like operations `car` for the head of a list, `cdr` for the tail of a list, and `cons` for adding an element as head to a list.

2.26. Specify a partial minus operation on natural numbers using the error supersort. For this specification, first parse and then evaluate an expression such as $3-(2-1)$.

2.27. Write a specification of the set of the positive rational numbers that is a confluent and terminating rewriting system.

2.7 Example: compiler correctness

In this section we develop an equational specification of a very simple programming language, of a compiler from this language to machine code, and formulate its correctness as an inductive property, and formally prove it by structural induction using rewriting. We use the CafeOBJ both as specification and as execution language.

The programming language considered is that of arithmetic expressions. However the same method can be used for more complex programming languages involving imperative constructs such as loops, `if-then-else`, etc.

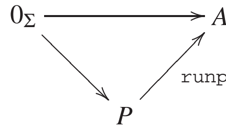
Initial algebra semantics principle. This is a very classical principle for defining formally the semantics of programming languages. According to this rather elegant principle, the syntax of a programming language can be specified as an algebraic signature Σ such that the programs are the precisely the terms over this signature. Then a semantics

for this language is given by a Σ-algebra A. Given any such semantics, each program can be evaluated to a semantic value, which is the value of the corresponding term through the unique Σ-algebra homomorphism from 0_Σ (the Σ-algebra of terms) to A. This is based upon the initiality property of the term algebra 0_Σ.

For the compiler example, this principle is applied at two different levels:

1. *denotational semantics*, and
2. *operational semantics*.

In the denotational semantics the algebra A is an algebra of the values which the programs of the language are expected to compute. The unique homomorphism $0_\Sigma \to A$ gives the denotation of each (source program). In the operational semantics the respective algebra is an algebra P of programs in a target lower level language, such as machine code. The compiler appears as the unique homomorphism $0_\Sigma \to P$. The running of compiled programs appears as a homomorphism `runp` between the operational algebra and the denotational algebra. The initiality property of 0_Σ guarantees the commutativity of the diagram below



which gives the correctness property for the compiler.

Summary of the arithmetic expression compiler specification and verification. The steps of the process of the specification of the compiler and of the formal verification of its correctness are as follows:

1. We specify a data type of simple arithmetic expressions with integer numbers.
2. We specify the denotational semantics for the arithmetic expressions.
3. We specify a compiler from arithmetic expressions to lists of machine instructions.
4. We specify a machine executing configurations formed by lists of machine instructions and value stacks storing intermediate computation results.
5. Finally, we formulate the correctness of the compiler as inductive property of our specification and prove it by structural induction using rewriting.

Our specifications, although not really large, still quite significantly involves a structuring mechanism. Therefore it would be useful if the reader had some familiarity with the material of Chap. 6. However we will try to explain on the spot the structuring involved in a rather informal and elementary way in order to minimize the dependence on a deep understanding of the structuring concepts presented in Chap. 6.

The syntax for arithmetic expressions. We consider simple arithmetic expressions formed only with three binary operation symbols representing addition, minus, and multiplication, and with integers as constants. The following is a specification of the set of the binary operation symbols.

```
mod! OPsym {
    [ Opsym ]
    ops ++ -- x : -> Opsym
}
```

For the integers we use the built-in module `INT` provided automatically by the CafeOBJ system.

```
mod! EXP {
    protecting(INT + OPsym)
    [ Int < Exp ]
    op (_ _ _) : Exp Opsym Exp -> Exp
}
```

The second line of module `EXP` means just that `EXP` first puts together the specifications `OPsym` and `INT` and after adds something more. Two things are thus added: (1) a subsorting relationship specifying the fact that in our arithmetic expressions language the integers are (primitive) expressions, and (2) a mix-fix constructor specifying the fact that the arithmetic expressions are built recursively from (smaller) arithmetic expressions by using the binary operators. We could have done all these without any structuring by writing together the specifications `OPsym` and `INT` plus (1) and (2) as a single specification module, but in that case we could not (re)use `INT` (made already available by the system) and also the specification would have been slightly harder to read. At this stage only slightly harder, but at a later stage, after more and more accumulation of data specifications, an eventually flat unstructured specfication may be pretty unreadable.

The denotation of `EXP`, which is tight (initial) consists of the algebra of the binary trees with the inner nodes labeled by the three binary operation symbols, and with the leaves labeled by integers. As a set of elements, this is the same as the terms constructed with the three binary operations and with the integers as constants, which is exactly the set of the arithmetic expressions.

From the perspective of the principle of initial algebra semantics we can consider the syntax for our simple language of arithmetic expressions as a signature $\Sigma = (S, F)$ with only one sort (namely `Exp`; the other sorts `OPsym` and `Int` playing just an auxiliary role) and such that

- $F_{\to \mathtt{Exp}}$ is the set of integer numbers, i.e. $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$,
- $F_{\mathtt{Exp}\,\mathtt{Exp} \to \mathtt{Exp}} = \{\mathtt{++}, \mathtt{--}, \mathtt{x}\}$, and
- $F_{w \to \mathtt{Exp}} = \emptyset$ otherwise.

The term algebra 0_Σ is indeed the set of all programs, in this case of all arithmetic expressions formed from the integers with the three above introduced binary operations.

Denotational semantics for arithmetic expressions. We continue to realize the principle of initial algebra semantics by specifying the denotational semantics Σ-algebra A which is as follows.

- $A_{\mathtt{Exp}} = \mathbb{Z}$, i.e. the set of the integers,

and for all integers m,n

- $A_m = m$,
- $A_{++}(m,n) = m+n$,
- $A_{--}(m,n) = m-n$, and
- $A_{\mathtt{x}}(m,n) = m \times n$.

The actual specification of A is given by the mapping `apply_to__` that specifies the interpretations of the binary operations. For example $A_{--}(m,n)$ is just (`apply -- to` m n).

```
mod! APP { protecting(EXP)
    op apply_to_ _ : Opsym Int Int -> Int
    vars I J : Int
    eq apply ++ to I J = I + J .
    eq apply -- to I J = I - J .
    eq apply x to I J = I * J .
}
```

The unique Σ-homomorphism $0_\Sigma \to A$, which gives the denotations to the individual expresions, is specified below as the function `eval_`. Note that the equations of `EVAL` just give the homomorphism conditions for `eval`.

```
mod! EVAL { protecting(APP)
    op eval_ : Exp -> Int
    vars E1 E2 : Exp
    eq eval (V:Int) = V .
    eq eval(E1 Op:Opsym E2) = apply Op to eval(E1) eval(E2) .
}
```

Testing of the arithmetic expressions. Now that we are done with the specification of the denotational semantics for the arithmetic expressions, and our specification has grown to certain size, we may take a short break and perform a little testing. Such testings during the building of specifications are important for several reasons. One is that it gives us a feeling of how our data types work in a very concrete way. It may also happen that we discover some bugs or things we want to adjust. Finally, testing may give us some confidence about the quality of our specification building process. So, our little testing goes as follows.

```
red eval(((2 -- 3) x 5) ++ (3 x 2)) .
```

This gives the expected result `1`.

Compiler: preliminary list data type. The second stage of our specification is devoted to the specification of the operational semantics for the arithmetic expressions. We first need a 'parameterized' data type of lists. This means that our data type is generic in the sense that the set of the elements for the lists is abstract and various list data types may be obtained just by instantiating it to various concrete sets. The parameter `TRIV` is a built-in module in the CafeOBJ system and consists of only one sort `Elt`, specified with loose semantics. This means its denotation consists of all sets. The lists over `Elt` are specified by a constructor operation `_@_` adding an element to a list and an associative operation `_+_` standing for the appending of lists. We have also to consider the empty list, denoted `nil`. The following is the CafeOBJ coding of these.

```
mod! LIST (ELT :: TRIV) {
    [ List ]
    op nil : -> List
    op (_@_) : Elt List -> List
    op (_+_) : List List -> List { assoc }
    vars l l1 l2 : List
    eq nil + l = l .
    eq (e:Elt @ l1) + l2 = e @ (l1 + l2) .
}
```

Thus the denotation of `LIST` consists of all algebras interpreting `Elt` as any set and `List` (together with `nil` and `_@_`) as the lists constructed from the elements of the interpretation of `Elt`. Note that because we have specified `nil` only as identity to the left its real functionality is as the end marker for lists.

Compiler: the target code. Now we are going to define the algebra P of the operational semantics. This algebra consists of lists of machine code-like instructions (specified by the sort `Inst`) and containing two kinds of instructions:

- loading an integer to the value stack, and
- applying an arithmetic operation.

```
mod! INST {
    protecting(OPsym + INT)
    [ Inst ]
    op Load_ : Int -> Inst
    op Apply_ : Opsym -> Inst
}
```

The data type of lists of instructions is obtained by instantiating the set of the elements `Elt` of the generic list data type `LIST` to the set of our instructions. For the sake of expressivity, the sort `List` is renamed to `InstList` and `List` to `InstList`. These are achieved by the following CafeOBJ code.

```
make INST-LIST (LIST(ELT <= view to INST {sort Elt -> Inst})
    * {sort List -> InstList})
```

Thus the denotation of `INST-LIST` consists of the algebra of lists of instructions.

The Σ-algebra P of the operational semantics is defined as follows.

- $P_{\mathtt{Exp}}$ is the set of lists of instructions, i.e. the interpretation of `InstList` in the denotation of `INST-LIST`,
- $P_m = (\mathtt{Load}\ m)\mathtt{@nil}$ for each integer m,

and for any lists l_1 and l_2,

- $P_{\mathtt{++}}(l_1, l_2) = l_1 + l_2 + (\mathtt{Apply\ ++})\mathtt{@nil}$,
- $P_{\mathtt{--}}(l_1, l_2) = l_1 + l_2 + (\mathtt{Apply\ --})\mathtt{@nil}$, and
- $P_{\mathtt{x}}(l_1, l_2) = l_1 + l_2 + (\mathtt{Apply\ x})\mathtt{@nil}$.

Compiler: the compiler function. Let us denote the unique Σ-homomorphism $0_\Sigma \to P$ by `compile`. Its name suggests its functionality: it just compiles expressions to lists of instructions. Its formal specification is given below.

```
mod! COMPILER {
    protecting (EXP + INST-LIST)
    op compile_ : Exp -> InstList
    eq compile V:Int = (Load V) @ nil .
    eq compile (E1:Exp Op:Opsym E2:Exp) =
       compile(E1) + compile(E2) + (Apply Op) @ nil .
}
```

Note that `compile` is *not* a surjective homomorphism, which means that there are elements of P which are not the result of `compile`. Not all lists of instructions, or programs in the machine code language, are compilations of arithmetic expressions. Note also that the execution aspect of CafeOBJ by rewriting gives it real programming capabilities shown by the fact that we can program a compiler; moreover this is achieved in a highly declarative way.

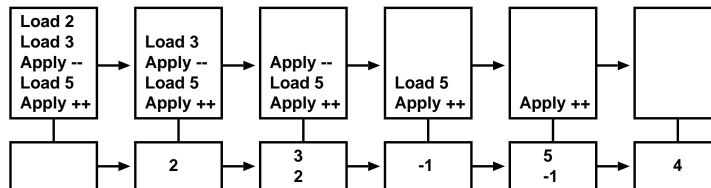
Testing the compiler. Now we may test our compiler. Let us do it on the same arithmetic expression that we have used for our previous testing.

```
red compile (((2 -- 3) x 5) ++ (3 x 2)) .
```

This gets the following program.

```
Load 2 @ Load 3 @ Apply -- @ Load 5 @ Apply x @ Load 3 @
Load 2 @ Apply x @ Apply ++ @ nil : InstList
```

The execution machine. The next step is to define the mapping `runp` that takes machine code programs and runs them for getting an integer as result. For this relatively complex task we have to define an execution machine that consists of configurations formed by a list of instructions and value stacks for storing intermediate computation results. The following is a sample of the execution of some code on the machine, which corresponds to the compiled code for the arithmetic expression `(2 -- 3) ++ 5`:



A value stack for storing intermediate computation results is specified as an instance of the `LIST` data type by instantiating the elements (`Elt`) with integer values (`Int`). For the sake of expresivity we also rename the sort `List` to `ValStack`. Note that this is a second reuse of the parameterized data type of lists.

```
make VALUE-STACK (LIST(ELT <= view to INT { sort Elt -> Int })
    * { sort List -> ValStack })
```

Execution machine: the specification.

```
mod! EXEC-MACHINE {
    protecting (VALUE-STACK + APP + COMPILER)
```

The operation `do_on_od` builds the configurations.

```
    [ Config ]
    op do_on_od : InstList ValStack -> Config
```

One step execution of configurations are specified by `exec_`. To address the situations when configurations cannot be executed, such as for example when the head of the list of instructions is an `Apply` and there are not at least two integers in the value stack, the result sort of `exec` is specified to be the error supesort for configurations. For such situations we *under-specify* the functionality of `exec`, which simply means that we do not write any equations corresponding to the respective cases.

```
    op exec_ : Config -> ?Config
    vars Il : InstList
    vars Vs : ValStack
    eq exec do ((Load E:Int) @ Il) on Vs od = do Il on (E @ Vs) od .
    eq exec do ((Apply Op:Opsym) @ Il) on (I:Int @ J:Int @ Vs) od =
            do Il on ((apply Op to J I) @ Vs) od .
```

The operation `run_` is meant for a complete execution of the configurations by using the auxiliary operation `exec_`. The normal form of terms with `run_` on the top represent the result of the complete execution of the corresponding configuration.

```
    op run_ : Config -> ?Config
    eq run do nil on Vs od = do nil on Vs od .
    eq run do (I:Inst @ Il) on Vs od =
       run exec do (I @ Il) on Vs od .
}
```

Note that the right hand side of the latter equation does not really parse well because the result sort of `exec` is `?Config` while the argument sort of `run` is `Config`. This is silently solved by the system by introducing a corresponding retract function (see Sect. 2.6). Now we can peform some testing of the execution machine:

```
red run do compile(((2 -- 3) x 5) ++ (3 x 2)) on nil od .
```

This gets

```
(do nil on (1 @ nil) od):Config.
```

The following is an example giving error value.

```
red run do Apply ++ @ nil on 1 @ nil od .
```

This gets

```
(run (retract (exec (do ((Apply ++) @ nil) on (1 @ nil) od)))):?Config
```

`runp` **revisited**. The operation `run_` specified above determines immediately the mapping `runp` from machine code programs to integer values. This takes programs and gives the values resulting from their execution. However as defined so far, `runp` is a partial function because as we have seen before not all machine code programs run to produce an integer value result. We solve this problem by considering for all machine code programs that are not compiled code of some arithmetic expression, an error result value, denoted `err`. Thus

$$\mathtt{runp}(IL) = \begin{cases} i & \text{if } \mathtt{run}(\mathtt{do}\ IL\ \mathtt{on}\ \mathtt{nil}\ \mathtt{od}) = \mathtt{do}\ \mathtt{nil}\ \mathtt{on}\ (i\ @\ \mathtt{nil})\ \mathtt{od} \\ \mathtt{err} & \text{otherwise.} \end{cases}$$

In order to have `runp` as a Σ-homomorphism from the operational semantics algebra P to the denotational semantics algebra we have to upgrade the latter by extending it with the error value `err`. Let us denote this upgraded algebra by A', and define it as follow.

- $A'_{\mathtt{Exp}} = \mathbb{Z} \cup \{\mathtt{err}\}$, and
- for any $\sigma \in \{\mathtt{++}, \mathtt{--}, \mathtt{x}\}$ we let $A'_\sigma(x,y) = \begin{cases} A_\sigma(x,y) & \text{if } x,y \in \mathbb{Z} \\ \mathtt{err} & \text{otherwise.} \end{cases}$

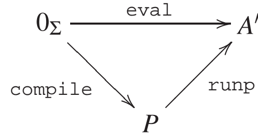
The formulation of compiler correctness. The correctness of the compiler can be informally stated as

> *The running of the compiled code for an arithmetic expression gets the same result as the (semantic) evaluation of the expression.*

which may be formally expressed as

$$\texttt{runp} \circ \texttt{compile} = \texttt{eval}.$$

If we proved that `runp` is a Σ-homomorphism $P \to A'$ then the compiler correctness is obtained directly from the initiality property of 0_Σ as shown by the diagram below.



The proof of the compiler correctness is thus reduced to the proof of the Σ-homomorphism property for `runp`. However we prefer here to use a simpler proof alternative, namely to prove directly

```
run do (compile E) on nil od = do nil on eval(E) nil od.
```
(2.90)

The proof of compiler correctness. It is actually more convenient to prove the following generalized version of (2.90).

```
run do ((compile E) + Il) on Vs od = run Il on (eval(E) @ Vs)
```
(2.91)

Since the denotation of our specification is initial, this is an inductive property and we prove it by using the structural induction method of Prop. 2.85. For our property the set X of variables of Prop. 2.85 is $\{\texttt{E:Exp}\}$. Since there are only two constructors of sort `Exp`, namely the integer numbers and `(___)`, we have only two cases for $Q_{\texttt{E}}$ of Prop. 2.85.

The proof score. Now we are ready to write the proof score for our inductive property.

```
open (EXEC-MACHINE + EVAL)
```

The case $Q_{\texttt{E}} = \texttt{v}$ for $\texttt{v} \in \mathbb{Z}$:

```
  op il : -> InstList .
  op vs : -> ValStack .
  op v : -> Int .
red run do (compile v) + il on vs od ==
    run do il on ((eval v) @ vs) od .
```

The case $Q_{\texttt{E}} = \texttt{(___)}$:

```
  ops e1 e2 : -> Exp .
  op op : -> Opsym .
eq run do ((compile e1) + Il:InstList) on Vs:ValStack od =
   run do Il on ((eval e1) @ Vs) od .
eq run do ((compile e2) + Il:InstList) on Vs:ValStack od =
   run do Il on ((eval e2) @ Vs) od .
red run do (compile (e1 op e2)) + il on vs od ==
    run do il on ((eval (e1 op e2)) @ vs) od .
close
```

Exercises.

2.28. Specify the Σ-algebra P of the operational semantics for the arithmetic expressions. Specify the Σ-algebra A' and the Σ-homomorphism `runp` and prove that the latter is indeed a homomorphism.

Chapter 3

Specification with Transitions

This chapter is devoted to rewriting as a specification formalism rather as an execution mechanism. The chapter is structured as follows.

1. We show how rewriting can be turned into a specification formalism, including the definition of an underlying logic that extends the logic of (many sorted) algebras developed in the previous chapter.
2. Next we develop a sound and complete calculus for this new logic as an extension of the equational proof calculus.
3. The rest of the chapter is devoted to methodologies for rewriting as specification, the most important being specification and verification of algorithms.

3.1 The logic of transitions

An example: bubble sorting by rewriting Rewriting is a generic algorithm in the sense that many algorithm can be coded as rewriting systems. Bubble sorting is such an example. Let the following be a specification of the data type of natural numbers with a strict 'less than' relation:

```
mod! PNAT= {
  [ Nat ]
  op 0 :  -> Nat
  op s_ : Nat -> Nat
  op _=_ : Nat Nat -> Bool {comm}
  vars M N : Nat
  eq ((s M) = 0) = false .
  eq (0 = 0) = true .
  eq (s M = s N) = (M = N) .
}
```

```
mod! PNAT< {
  protecting(PNAT=)
  op _<_ : Nat Nat -> Bool
  vars M N : Nat
  eq 0 < s M = true .
  eq M < 0 = false .
  eq (s M < s N) = M < N .
}
```

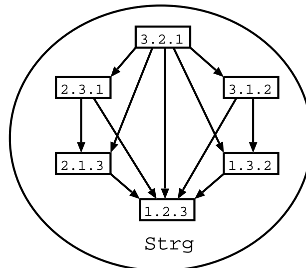
The following is a specification of strings of natural numbers with concatenation:

```
mod! STRG-PNAT< {
  protecting(PNAT<)
  [ Nat < Strg ]
  op _;_ : Strg Strg -> Strg  {assoc}
}
```

The bubble sorting may be specified by only one rewriting rule modulo the associativity of the concatenation operation. The rather compact coding of the bubble sorting algorithm owes to the power of using operation attributes and of rewriting modulo axioms.

```
mod! SORTING-NAT {
  protecting(STRG-PNAT<)
  cq M:Nat ; N:Nat = N ; M  if N < M .
}
```

The denotation of `SORTING-NAT` consists of the sorted strings of natural numbers, i.e. the results of the sorting computation. Thus, at the level of denotational semantics the whole process of sorting is collapsed to the final results, which means that in this case equational logic is too gross for specifying the states of the algorithm. However the states of the sorting algorithm are recovered at the level of the preordered algebra of the rewriting relation modulo associativity. In more precise words, if E consists of associativity of concatenation (in addition to the equations of `PNAT<`) and Γ of sorting equation of `SORTING-NAT`, then the elements of this preordered algebra are the strings of naturals, i.e. classes of terms modulo $=_E$, and the preorder relation is $\xrightarrow{\star}_{\Gamma,E}$, the rewriting relation modulo E. The following figure shows the fragment of this preorder relation on on strings of naturals that corresponds to the sorting of the string (`3 ; 2 ; 1`).



schimba . cu ; in figura de mai sus si in cele de mai jos

Transitions versus equations. In order to have the preorder algebra of $\stackrel{\star}{\longrightarrow}_{\Gamma,E}$ as the denotation of the specification, we have to refine the logic of many sorted algebras of Chap. 2 in a way that we distinguish between rewrite rules (E) that are used for computing equalities, and which collapse elements, and those (Γ) that do not collapse elements but are rather used for specifying transitions between elements. In CafeOBJ and Maude, respectively, this is achieved by denoting transitions by `trans` (or `ctrans` in the conditional case) and `rl` (or `crl`), respectively, rather than by `eq` (or `ceq`).

This means a refinement of the logic of Chap. 2 to a logic that considers preordered algebras rather than algebras as models and that has two kinds of atoms: equations and transitions. This is called the logic of preordered algebras (abbreviatted *POA*). Both CafeOBJ and Maude support directly *POA* as a specification paradigm, although for Maude this is the main focus and is treated at a more sophisticated level than presented here. Therefore the CafeOBJ code of the *POA* specification of bubble sorting is

```
mod! SORTING-NAT {
  protecting(STRG-PNAT<)
  ctrans M:Nat ; N:Nat => N ; M  if N < M .
}
```

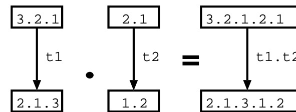
The formal definition of *POA*. In the following we define *POA* formally as a logic by extending the logic of (many sorted) algebras of Chap. 2. Later on we extend the logic results of Chap. 2 to *POA*, including existence of initial models and a sound and complete proof calculus for conditional sentences. These results constitute the foundations for *POA* specification with conditional sentences.

Definition 3.1 (*POA* signatures)**.** *The* signatures of *POA are just the algebraic signatures, i.e. of the form* (S,F).

Definition 3.2 (*POA* models and homomorphisms)**.** *Given a signature* (S,F)*, a POA* (S,F)-model *is just a preordered* (S,F)*-algebra. We may denote preordered* (S,F)*-algebras by* $(A,\leq)$ *where* A *is the underlying (discrete)* (S,F)*-algebra and* $\leq$ *is the family of preorders* $(\leq_s)_{s\in S}$ *on the interpretations of the sorts.*

A homomorphism of preordered (S,F)-algebras $h\colon (A,\leq) \to (A',\leq')$ *is just an* (S,F)*-homomorphism* $A \to A'$ *that is monotone, i.e.* $h_s(a) \leq'_s h_s(b)$ *for any* $s \in S$ *and any* $a \leq_s b$*.*

The monotonicity of the interpretation of the operations in preordered algebras can be read as an extension of the interpretation of the operations from elements to transitions. For example, in the preordered algebra of `SORTING-NAT` discussed above we may think that transitions between strings concatenate like shown by the figure below.



The concatenation of transitions represents the parallel aspect of the bubble sorting, we can sort independently in parallel different parts of a string and then join the results and eventually continue from there.

Definition 3.3 (*POA* sentences)**.** *The sentences of POA are constructed like in Dfn. 2.6 with the difference that we consider two kinds of atoms*

1. equations*, of the form* $t = t'$ *(like in Dfn. 2.6), and*
2. transitions*, of the form* $t \Rightarrow t'$ *where* t *and* t' *are terms of the same sort.*

A Horn clause *in POA is just a sentence of the form* $(\forall X)H \Rightarrow C$ *where* H *is a finite conjunction of atoms (either equations or transitions) and* C *is a single atom (either equation or transition).*

Definition 3.4 (*POA* satisfaction)**.** *For a given signature* (S,F)*, the satisfaction between preordered algebras and POA sentences extends the definition of satisfaction for algebras (Dfn. 2.10) with the satisfaction of transitions:*

$$(A, \leq) \models_{(S,F)} t \Rightarrow t' \text{ if and only if } A_t \leq A_{t'}.$$

Now it is the moment to note that the logic of *POA* goes beyond rewriting modulo axioms since in *POA* the sentences may involve equations and transitions without restrictions; for example we may have equations conditioned by transitions, a situation that cannot be captured by splitting the rewriting rules into Γ (for transitions) and E (for equations).

Initial semantics in *POA*. It is rather easy to show (we leave this as exercise to the reader) that the preordered algebra corresponding to the bubble sorting algorithm on the strings of naturals discussed above is the initial preordered algebra satisfying the sentences of `SORTING-NAT`. In fact, like for conditional equations, any set of Horn clauses in *POA* admits an initial model. This result allows for initial semantics specifications with Horn clauses in *POA*, which is an important specification methodology using transitions. In the rest of this section we develop the result on existence of initial models for Horn clauses in *POA* by following the same proof pattern as for the existence of initial algebras for conditional equations.

Congruences and quotients in *POA*. First, we need to upgrade the notion of congruence from algebras to preordered algebras.

Definition 3.5 (*POA* congruence)**.** *A POA*-congruence *on a preordered* (S,F)*-algebra* $(A, \leq)$ *is a pair* $(\equiv, \sqsubseteq)$ *such that*

- $\equiv$ *is an* (S,F)*-congruence on* A*,*
- $(A, \sqsubseteq)$ *is a preordered algebra such that* $\leq_s \subseteq \sqsubseteq_s$ *for each sort* $s \in S$*, and*
- *for each* $s \in S$*,* $a' \equiv_s a, a \sqsubseteq_s b, b \equiv_s b'$ *implies* $a' \sqsubseteq_s b'$ *for any elements* $a, a', b, b' \in A_s$*.*

Congruences form a partial order under inclusion, i.e. $(\equiv, \sqsubseteq) \subseteq (\equiv', \sqsubseteq')$ *if and only if* $\equiv_s \subseteq \equiv'_s$ *and* $\sqsubseteq_s \subseteq \sqsubseteq'_s$ *for each* $s \in S$.

The following is an example of *POA*-congruence. Let us consider a signature consisting of one sort s and a binary operation $_+_$ and a preordered algebra $(A, \leq)$ for this signature defined by

- A_s is the set of strings of natural numbers,
- A_+ performs component-wise addition of naturals, for example $(1;2) + (2;3;1) = (3;5;1)$, and
- $x1 \leq x2$ if and only if $x1$ is a 'lax prefix' of $x2$ in the sense that the length of $x1$ is less than or equal to the length of $x2$ and the elements of $x1$ are less than or equal to the elements of $x2$ component-wise; for example $(1;2) \leq (2;2;3)$.

Then $(\equiv, \sqsubseteq)$ is a *POA*-congruence on $(A, \leq)$ where

- $x1 \equiv x2$ if and only if $x1$ and $x2$ have the same length, and
- $x1 \sqsubseteq x2$ if and only if the length of $x1$ is less than or equal to the length of $x2$.

Proposition 3.6. *Each POA-congruence* $(\equiv, \sqsubseteq)$ *on a preordered algebra* $(A, \leq)$ *determines a* quotient *preordered algebra homomorphism* $q: (A, \leq) \to (A/_{\equiv}, \leq')$ *where*

– $q: A \to A/_{\equiv}$ *is the quotient algebra homomorphism determined by the congruence* $\equiv$ *as in Dfn. 2.18, and*

– $a/_{\equiv} \leq' b/_{\equiv}$ *if and only if* $a \sqsubseteq b$.

The quotient preordered algebra $(A/_{\equiv}, \leq')$ *may be also denoted by* $(A, \leq)/_{(\equiv, \sqsubseteq)}$.

Proof. The definition of the preorder relation $\leq'$ is correct since it is independent on the choice of a and b. Indeed, let $a \equiv a'$ and $b \equiv b'$. By the definition of *POA*-congruences we have that $a \sqsubseteq b$ if and only if $a' \sqsubseteq b'$. Moreover that $\leq'$ is a preorder follows from the fact that $\sqsubseteq$ is a preorder.

In order to complete the argument that $(A/_{\equiv}, \leq')$ is a preordered algebra we have to show that the interpretations of the operations on $A/_{\equiv}$ are monotone with respect to the preorder $\leq'$. Let σ be any operation symbol and $(a_1, \ldots, a_n)$ and $(a'_1, \ldots, a'_n)$ appropriate lists of arguments for A_σ. We have to prove that

$$(A/_{\equiv})_\sigma(a_1/_{\equiv}, \ldots, a_n/_{\equiv}) \leq' (A/_{\equiv})_\sigma(a'_1/_{\equiv}, \ldots, a'_n/_{\equiv})$$

if $a_k/_{\equiv} \leq' a'_k/_{\equiv}$ for each $1 \leq k \leq n$. This is equivalent to

$$A_\sigma(a_1, \ldots, a_n)/_{\equiv} \leq' A_\sigma(a'_1, \ldots, a'_n)/_{\equiv} \text{ and further to } A_\sigma(a_1, \ldots, a_n) \sqsubseteq A_\sigma(a'_1, \ldots, a'_n).$$

The latter relation holds because $a_k \sqsubseteq a'_k$ (since $a_k/_{\equiv} \leq' a'_k/_{\equiv}$) for each $1 \leq k \leq n$ and by the definition of the *POA*-congruence which guarantees that A_σ is monotone with respect to the preorder $\sqsubseteq$.

The quotient homomorphism q is monotone because if $a \leq b$ then $a \sqsubseteq b$ which implies $a/_{\equiv} \leq' b/_{\equiv}$. □

In the case of the *POA* congruence presented above, the corresponding quotient preordered algebra $(A/_{\equiv}, \leq')$ has the natural numbers as its elements, interprets $_+_$ as the maximum between two natural numbers, and $\leq'$ is the ordinary 'less than or equal' relation between naturals.

The following is the refinement of the concept of kernel of homomorphism of algebras (Dfn. 2.19) to the situation of preordered algebras.

Definition 3.7 (Kernel of homomorphism in *POA*)**.** *For any preordered algebra homomorphism* $h: (A,\leq) \to (A',\leq')$ *its* kernel*, denoted* $ker(h)$*, is defined as the pair of families of binary relations* $(=_h, \leq_h)$ *where*

- $=_h$ *is the kernel of* h *as algebra homomorphism* $A \to A'$*, and*
- *for each sort* s *of the signature and any* $a,b \in A_s$*,* $a(\leq_h)_s b$ *if and only if* $h_s(a) \leq_s h_s(b)$*.*

Fact 3.8. *$ker(h)$ is a POA-congruence.*

The following is a *POA* refinement of Prop. 2.21 to preordered algebras.

Proposition 3.9. *For any surjective POA homomorphism* $q: (A,\leq) \to (A',\leq')$ *and any POA homomorphism* $h': (A,\leq) \to (B,\leq'')$*, there exists an unique POA homomorphism* $h': (A',\leq') \to (B,\leq'')$ *such that* $q;h' = h$ *if and only if* $ker(q) \subseteq ker(h)$*.*

$$\begin{array}{ccc} (A,\leq) & \xrightarrow{q} & (A',\leq') \\ & h \searrow \quad \swarrow h' & \\ & (B,\leq'') & \end{array}$$

Proof. The direct implication is almost trivial, hence we focus on the inverse implication. Recall that $ker(q) \subseteq ker(h)$ means that $=_q \subseteq =_h$ and $\leq_q \subseteq \leq_h$. Because $=_q \subseteq =_h$ we can use Prop. 2.21 and obtain the existence and uniqueness of h' as algebra homomorphism $A' \to B$. It thus remains to show that h' is also monotone.

Let us assume $a'_1 \leq'_s a'_2 \in A'_s$. Then if we write $a'_1 = q_s(a_1)$ and $a'_2 = q_s(a_2)$, we have that $a_1(\leq_q)_s a_2$ which by hypothesis implies $a_1(\leq_h)_s a_2$ which means $h_s(a_1) \leq''_s h_s(a_2)$. By definition we have that $h'_s(a'_k) = h_s(a_k)$, hence $h'_s(a'_1) \leq''_s h'_s(a'_2)$. □

Free preordered algebras. The following extends the concept of Γ-congruence from ordinary algebras (Dfn. 2.22) to preordered algebras.

Definition 3.10 (Γ-congruence in *POA*)**.** *For any finite conjunction* $H = (t_1 = t'_1) \wedge \cdots \wedge (t_n = t'_n) \wedge (t_{n+1} \Rightarrow t'_{n+1}) \wedge \cdots \wedge (t_k \Rightarrow t'_k)$ *of POA* (S,F)*-atoms and any preordered* (S,F)*-algebra* $(A,\leq)$*, by* A_H *we abbreviate the pair of sets* $(\{(A_{t_i}, A_{t'_i}) \mid 1 \leq i \leq n\}, \{(A_{t_i}, A_{t'_i}) \mid n+1 \leq i \leq k\})$*.*

Given Γ *a set of POA Horn clauses for a signature* (S,F)*, a POA congruence* $(\equiv, \sqsubseteq)$ *on a preordered* (S,F)*-algebra* $(A,\leq)$ *is a* Γ-congruence *when for each Horn clause* $(\forall X)H \Rightarrow C$ *in* Γ *and for any expansion* A' *of* A *to* $(S, F \cup X)$*,* $A'_H \subseteq (\equiv, \sqsubseteq)$ *implies* $A'_C \subseteq (\equiv, \sqsubseteq)$*.*

Fact 3.11. *The Γ-congruences on a preordered algebra $(A,\leq)$ are closed under arbitrary intersections. Let $(=_\Gamma^{(A,\leq)},\leq_\Gamma^{(A,\leq)})$ denote the least Γ-congruence on $(A,\leq)$, which is the intersection of all Γ-congruences on $(A,\leq)$. Then by $q_\Gamma^{(A,\leq)}$ we denote the quotient homomorphism that corresponds to $(=_\Gamma^{(A,\leq)},\leq_\Gamma^{(A,\leq)})$.*

The following three results refine the corresponding results about algebras to preorder algebras. Since their proofs just mimic the proofs of Prop. 2.23, Prop. 2.24, and of Cor. 2.26, respectively, we omit them here.

Proposition 3.12. *For any set of POA Horn clauses Γ for a signature (S,F) and any homomorphism of preordered (S,F)-algebras $h:\ (A,\leq)\to(B,\leq')$, if $(B,\leq')\models\Gamma$ then ker(h) is a POA Γ-congruence.*

Proposition 3.13. *For any set of POA Horn clauses Γ for a signature (S,F) and for any POA congruence $(\equiv,\sqsubseteq)$ on a preordered (S,F)-algebra $(A,\leq)$*

$$(A,\leq)/_{(\equiv,\sqsubseteq)}\models\Gamma \ \text{ if and only if } \ (\equiv,\sqsubseteq) \ \text{ is a POA } \Gamma\text{-congruence.}$$

The concept of free preordered algebras in the result below is like the concept of free algebras of Cor. 2.26.

Corollary 3.14. *For any set of POA Horn clauses Γ for a signature (S,F) and for any preordered (S,F)-algebra $(A,\leq)$, $(A,\leq)/_{(=_\Gamma,\leq_\Gamma)}$ is the* free preordered algebra over $(A,\leq)$ *satisfying* Γ.

$$\begin{array}{ccc}(A,\leq) & \xrightarrow{q_\Gamma} & (A,\leq)/_{(=_\Gamma,\leq_\Gamma)} \\ & {\scriptstyle\forall h}\searrow \quad \swarrow{\scriptstyle\exists! h_\Gamma} & \\ & (B,\leq')\models\Gamma & \end{array}$$

Term preordered algebras. From Prop. 2.27 let us recall the initial term algebra $0_{(S,F)}$.

Proposition 3.15. *For any signature (S,F), $(0_{(S,F)},=)$ is the initial preordered (S,F)-algebra.*

Proof. We have just to note that for each preordered (S,F)-algebra $(A,\leq)$, the unique algebra homomorphism $h:\ 0_{(S,F)}\to A$ is trivially monotone. □

Existence of initial preordered algebras for Horn clause specifications. The following is obtained as an instance of Cor. 3.14 for the initial preordered algebra (of Prop. 3.15).

Corollary 3.16. *For each set Γ of Horn clauses in POA, $(0_{(S,F)},=)/_{(=_\Gamma,\leq_\Gamma)}$ is the initial preordered algebra which satisfies Γ.*

For example, the initial preordered algebra of `SORTING-NAT` is the preordered algebra discussed at the beginning of this section, having the strings of naturals as elements of sort `Strg`, and the preorder relation on strings being generated by one step transitions of the bubble sorting algorithm.

Exercises.

3.1. Show that bijective homomorphisms of preordered algebras are not necessarily isomorphisms.

3.2. Let Γ and E be two sets of (S,F)-equations. Show that the preordered algebra determined by $\xrightarrow{\star}_{\Gamma,E}$, the rewriting relation by Γ modulo E, is the initial preordered algebra that satisfies Γ as transitions and E as equations.

3.2 Proof in preordered algebra.

Preordered algebra deduction.

Definition 3.17 (Proof theoretic entailment system for *POA* Horn clauses)**.** *The* proof theoretic entailment system for *POA* Horn clauses*, denoted* $\vdash^{HPOA}$*, is the least entailment system which satisfies the meta-rules of* Implication *and of* Universal Quantification *and which contains the the equational proof rules* Reflexivity*,* Symmetry*,* Transitivity*, and* Congruence *of Dfn. 2.37, the following rules for transitions*

Trans-Reflexivity: $\dfrac{\emptyset}{t \Rightarrow t}$ *for all* (S,F)*-terms* t*.*

Trans-Transitivity: $\dfrac{\{t \Rightarrow t',\ t' \Rightarrow t''\}}{t \Rightarrow t''}$ *for all* (S,F)*-terms* t*,* t' *and* t'' *of the same sort.*

Trans-Congruence: $\dfrac{\{t_i \Rightarrow t'_i \mid 1 \le i \le n\}}{\sigma(t_1,\ldots,t_n) \Rightarrow \sigma(t'_1,\ldots,t'_n)}$ *for each operation symbol* $\sigma \in F_{s_1\ldots s_n \to s}$ *and any terms* t_i*,* t'_i *of sort* s_i *for* $1 \le i \le n$*.*

the following upgraded (from Dfn. 2.37) rule

POA-Substitutivity: $\dfrac{\{(\forall X)H \Rightarrow C\}}{\{\theta(H \Rightarrow C)\}}$ *for any Horn clause* $(\forall X)H \Rightarrow C$ *for* (S,F) *and for each substitution* $\theta : X \to T_{(S,F)}$*.*

and the following rule

Compatibility: $\dfrac{\{t_1 = t_2,\ t_1 \Rightarrow t'_1,\ t'_1 = t'_2\}}{\{t_2 \Rightarrow t'_2\}}$ *for all* (S,F)*-terms* t_1, t_2, t'_1, t'_2*.*

Soundness and completeness. The soundness of $\vdash^{HPOA}$ is obtained in manner very similar to the soundness of equational deduction (Prop. 2.49) by checking the soundness of each of the proof rules of Dfn. 3.17. For the equational proof rules this has already been done by Prop. 2.38,and for the rules for transitions it can be done like in Prop. 2.38, therefore. we omit these details here.

Proposition 3.18 (Soundness of *HPOA* deduction)**.** *The entailment system* $\vdash^{HPOA}$ *is sound.*

Thecompleteness of $\vdash^{HPOA}$ is obtained along the same lines as for the completeness of equational deduction (Thm. 2.50), therefore here we just sketch its proof here leaving the details to the reader.

Theorem 3.19 (Completeness of *HPOA* deduction)**.** *The entailment system $\vdash^{HPOA}$ is complete.*

Proof. For any fixed set Γ of Horn clauses in *POA* let us consider the following two relations on the initial (term) preordered algebra $(0_{(S,F)}, =)$:

- $\equiv_\Gamma = \{(t,t') \mid \Gamma \vdash^{HPOA} t = t'\}$, and
- $\sqsubseteq_\Gamma = \{(t,t') \mid \Gamma \vdash^{HPOA} t \Rightarrow t'\}$.

Then pair $(\equiv_\Gamma, \sqsubseteq_\Gamma)$ can be shown to be a *POA* Γ-congruence and by using the quotient preordered algebra in the manner of the proof of Thm. 2.50 we obtain that $(0_{(S,F)}, =)/_{(\equiv_\Gamma, \sqsubseteq_\Gamma)} \models \Gamma$ that leads for each equation or transition C to

$$\Gamma \models C \text{ implies } \Gamma \vdash^{HPOA} C. \tag{3.1}$$

The relation (3.1) then lifts to all Horn clauses by using the meta-rules of *Implication* and *Universal Quantification* like in the proof of Thm. 2.50. □

Induction in *POA*. The issue of inductive properties and proof methods for them in *POA* is almost identical to that of induction for ordinary algebras. This means that the concepts and results of Sect. 2.5 carry to the framework of *POA* with almost no modifications, modulo the fact that for *POA* we consider sentences constructed also from atomic transitions besides equations, and also that the considered models are preordered algebras rather than algebras. In particular, the statements and the proofs of the crucial results of Prop. 2.85 and Prop. 2.86 can be interpreted in *POA* without any change in their original form. Therefore the structural induction proof method is available for *POA* in the same form as for ordinary algebra. In Sect. 3.4 we will develop an example of induction proof based upon the interpretation in *POA* of Prop. 2.85 and Prop. 2.86.

Rewriting in *POA*. The proof calculus for *POA* Horn clauses given by the entailment system $\vdash^{HPOA}$ can be mechanised by rewriting in a way very similar to equational reasoning. This means a rather straightforward extension of results of Sect. 2.4 as follows.

Definition 3.20 (Rewriting entailment in *POA*)**.** *The POA* (term) rewriting entailment system *(denoted $\vdash^{r}$ like for ordinary algebras) is the least entailment system for POA Horn clauses containing the proof rules of Dfn. 3.17 minus the rule of* Symmetry *and which satisfies the meta-rules of* Implication *and of* Universal Quantification.

The following is a replica of Prop. 2.57 to the *POA* framework, its proof being almost identical to the proof of Prop. 2.57.

Proposition 3.21. *The POA rewriting entailment system is the least entailment system containing the proof rules of* Reflexivity, Transitivity*, and*

POA-Rewriting: $\dfrac{\{(\forall X)H \Rightarrow C\} \cup \theta(H)}{c[\theta(C)]}$ *for any substitution $\theta : X \to T_{(S,F)}$ and each context c.*

(where by $c[\theta(C))]$ we mean $c[\theta(t)] = c[\theta(t')]$ when C is $t = t'$ and and $c[\theta(t)] \Rightarrow c[\theta(t')]$ when C is $t \Rightarrow t'$) and which satisfies the meta-rules of Implication *and of* Universal Quantification.

For any set Γ of *POA* Horn clauses we define the following *rewriting relation* on (S,F)-terms:

$$t \overset{\star}{\longrightarrow}_\Gamma t' \text{ if and only if } \Gamma \vdash^{\mathrm{r}} t \Rightarrow t'.$$

Since $\vdash^{\mathrm{r}}$ is less than $\vdash^{HPOA}$ we have immediately the following soundness consequence for *POA* rewriting.

Corollary 3.22. *If $t \overset{\star}{\longrightarrow}_\Gamma t'$ then $\Gamma \models t \Rightarrow t'$.*

Like in the case of rewriting for ordinary algebras, the rewriting relation $\overset{\star}{\longrightarrow}_\Gamma$ can be considered more generally on an arbitrary (S,F)-algebra A:

$$\overset{\star}{\longrightarrow}_{\Gamma,A} \text{ is the least reflexive-transitive closure of } \{(A_t, A_{t'}) \mid \Gamma \vdash^{\mathrm{r}} t \Rightarrow t'\}.$$

Moreover Prop. 2.69 and Prop. 2.70 admit replicas similar to their original form. As in the ordinary algebraic framework, when A is the initial algebra for a set E of conditional equations, the relation $\overset{\star}{\longrightarrow}_{\Gamma,A}$ is denoted $\overset{\star}{\longrightarrow}_{\Gamma,E}$ and we have *POA* replicas of Cor. 2.73 and Cor. 2.74, respectively, collected by the following single result.

Corollary 3.23. *For any (S,F)-terms t and t' we have that*

$$(t/_{=E}) \overset{\star}{\longrightarrow}_{\Gamma,E} (t'/_{=E}) \text{ if and only if } \{t_1 = t_2 \mid E \models t_1 = t_2\} \cup \Gamma \vdash^{\mathrm{r}} t \Rightarrow t'.$$

Consequently

$$(t/_{=E}) \overset{\star}{\longrightarrow}_{\Gamma,E} (t'/_{=E}) \text{ implies } \Gamma \cup E \models t \Rightarrow t'.$$

`exec` versus `red`. The computation of normal forms for $\overset{\star}{\longrightarrow}_{\Gamma,E}$ is supported in CafeOBJ by the command `exec`, the corresponding Maude command being `rewrite`. Thus, while in CafeOBJ the command `red` uses only the equations of the specification in rewriting, the command `exec` uses both equations and transitions in rewriting. In both situations rewriting is performed modulo the declared operation attributes.

For example, the sorting of strings of naturals may be performed in CafeOBJ as follows:

```
SORTING-NAT> exec (s s s 0 ; s s 0 ; s 0) .
```

which gives the result

```
  s 0 ; s s 0 ; s s s 0 : Strg
```

In this example E consists of the associativity of concatenation and Γ of the three equations of `PNAT<` plus the conditional transition declared by `SORTING-NAT`. Thus the rather compact specification of bubble sorting given by `SORTING-NAT` functions also as a sorting program, a highly *declarative* one based upon rewriting modulo associativity.

3.3 Algorithm specification and verification

One of the most meaningful applications of *POA* is for formal specification and verification of algorithms. We have already seen the example of an (executable) *POA* specification of bubble sorting for strings of naturals. In this section we develop a formal verification for two of the important properties of this algorithm, namely termination and confluence. Moreover, in this section we consider the bubble sorting algorithm in a more general form than in the previous section.

Generic bubble sorting. Many algorithms have a certain degree of independence of the actual data type, in the sense that they do not depend on all details of the data type, but rather only on few of the properties of the data type. If we have a careful look at bubble sorting then we see easily that this is the case of bubble sorting too. Without any problem we may replace in `SORTING-NAT` the naturals by other number types such as integers, reals, etc. What is important is to have an ordering, which does not even need to be total, it can be partial. Moreover, bubble sorting would work also for some binary relations that are not even orderings, but in those cases the confluence of the algorithm may be lost. However we will see that termination always holds. Sorting over such non-conventional relations is sometimes referred to as *topological sorting*. While the properties of termination and confluence are quite obvious in the case of the natural numbers, they are less so for other structures.

Our first step is to specify a data type of strings parameterised by abstract binary relations for the elements. Next we will specify bubble sorting for those strings and formally verify the above mentioned properties.

Specifying generic bubble sorting. We first specify strings over any sets of elements (specified here by `TRIV`).

```
mod* TRIV { [ Elt ] }

mod! STRG (X :: TRIV) {
  [ Elt < Strg ]
  op nil :  -> Strg
  op _;_ : Strg Strg -> Strg assoc id: nil
}
```

Note that in `STRG` the fact that `nil` is identity for the concatenation of strings is specified as an operation attribute (`id: nil`).

Next we specify a class of binary relations, that includes the partial orders. The additional predicate `_not<_` stands for the negation of the main relation `_<_`.

```
mod* PSEUDO-ORDER {
  [ Elt ]
  op _<_ : Elt Elt -> Bool
  op _not<_ : Elt Elt -> Bool
```

```
  vars E1 E2 E3 : Elt
  cq (E1 not< E2) = true if E2 < E1 or not(E1 < E2) .
  eq (E1 < E2) and (E1 not< E2) = false .
}
```

For specifying the generic bubble sorting we just specify that the parameter of the elements of the strings is model of PSEUDO-ORDER and add the sorting transition.

```
mod! SORTING-STRG(Y :: PSEUDO-ORDER) {
  protecting(STRG(Y))
  ctrans E:Elt ; E':Elt => E' ; E  if (E' < E) .
}
```

From generic to concrete sorting. The sorting of naturals can be obtained from the generic sorting specification SORTING-STRG by instantiating the parameter Y to PNAT<, the ordering of the naturals. This is done by the following 'view' which specifies the way _<_ and _not<_ are interpreted in PNAT<. The crucial point of this instantiation is that the interpretations of _<_ and _not<_ satisfy the axioms of PSEUDO-ORDER. This has to be done formally through a proof score, however we omit this here.

```
view PNAT<asPO from PSEUDO-ORDER to PNAT<
     {op (E:Elt not< E':Elt) -> ((E:Nat = E':Nat) or (E' < E))} .
```

Then the module SORTING-STRG(PNAT<asPO) is the same as SORTING-NAT.

Proving termination. Now we prove the termination of SORTING-STRG. This implies that any of its instances is also terminating. Mathematically, the termination property considered here is that the (sorting) preorder relation of the initial preordered algebra of SORTING-STRG is Noetherian. For this we use a rather common technique, that of defining a so-called *weight function* w on the states of the algorithm, with natural numbers as values, such that

$$t \Rightarrow t' \text{ implies } w(t') < w(t)$$

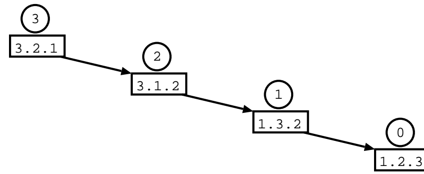
for any strings t and t'. Because the natural numbers are well-founded with respect to $<$, the existence of a such weight function w means that there are no infinite chains of transitions between the states of the sorting algorithm.

The weight function. The weight function w is defined as a measure for the distance from the states of the algorithm to the sorted state and is specified below by the function disorder.

```
mod! PNAT+ {
  protecting(PNAT=)
  op _+_ : Nat Nat -> Nat
  vars M N : Nat
  eq [succ+] : M + (s N) = s(M + N) .
```

```
  eq M + 0 = M .
}
mod! SORTING-DISORDER (Y :: PSEUDO-ORDER) {
  protecting(SORTING-STRG(Y) + PNAT+)
  op _>>_ : Elt Strg -> Nat
  op disorder : Strg -> Nat
  vars E E' : Elt
  vars S S' : Strg
  eq E >> nil = 0 .
  cq E >> E' =  s 0 if (E' < E) .
  cq E >> E' =  0   if (E' not< E) .
  eq E >> (S ; S') = (E >> S) + (E >> S') .
  eq disorder(E) = 0 .
  eq disorder(E ; S) = disorder(S) + (E >> S) .
}
```

The following shows some of the behaviour of `disorder` for its instance to the natural numbers.



We may test this with the CafeOBJ system by giving this instance a run as follows:

```
select SORTING-DISORDER(PNAT<asPO) .
red disorder(s s s 0 ; s 0 ; s s 0) .
```

Termination proof score. Our proof of termination may be classified as 'semi-formal' because we will build and run a proof score for the property

$$\texttt{disorder}(S;E1;E2;S') < \texttt{disorder}(S;E2;E1;S') \text{ if } E1 < E2. \tag{3.2}$$

for all S, S' : `Strg` and E_1, E_2 : `Elt`. From this we derive the desired property

$$\texttt{disorder}(s) < \texttt{disorder}(s') \text{ if } s \Rightarrow s' \text{ (for } s \neq s')$$

by using the fact that any transition $s \Rightarrow s'$ with $s \neq s'$ is a finite composition of one-step transitions like in (3.2). This latter rather obvious fact follows from the mathematical theory (see the proof of Prop. 2.69), and constitutes the non-formal part of the termination proof. However note that both this property and (3.2) are inductive properties, they are not general consequences of the axioms of the specifications.

The proof score for (3.2) requires the following auxiliary relation on the strings:

$$S <> S' \text{ if and only if } (\forall E : \texttt{Elt})\, E >> S = E >> S'.$$

Since conditional equations are not expressive enough to specify this relation, we only introduce the notation and use the definition of <> by hand in the proof score.

```
mod* SORTING<> (Y :: PSEUDO-ORDER) {
  protecting(SORTING-DISORDER(Y))
  op _<>_ : Strg Strg -> Bool
}
```

The proof score of (3.2) given below uses four lemmas, from which the last two refer to properties of the natural numbers.

```
open SORTING<> + PNAT< .
  vars E E' : Elt
  vars S S1 S2 : Strg
  vars M N P : Nat
  cq [Lemma-1] : disorder(S ; S1) < disorder(S ; S2) = true
     if S1 <> S2 and disorder(S1) < disorder(S2) .
  eq [Lemma-2] : (E ; E' ; S) <> (E' ; E ; S) = true .
  ops e1 e2 :  -> Elt .
  ops s s' :  -> Strg .
-- [Lemma-3] :
  op _+_ : Nat Nat -> Nat {assoc comm}
  eq [Lemma-4] : M < s M = true .
```

We introduce the condition of (3.2):

```
  eq e1 < e2 = true .
```

and proceed with the proof of the conclusion.

```
red disorder(s ; e1 ; e2 ; s') < disorder(s ; e2 ; e1 ; s') .
close
```

Proof score for `Lemma-1`. The proof of `Lemma-1` is by induction on `S` and is based upon the result of the structural induction Prop. 2.86, where

- $X = \{\texttt{S}\}$ and $Y = \emptyset$, and
- ρ is $(\forall S1, S2 : \texttt{Strg})(S1 <> S2) \wedge \texttt{disorder}(S1) < \texttt{disorder}(S2) = \texttt{true} \Rightarrow \texttt{disorder}(S\,;\,S1) < \texttt{disorder}(S\,;\,S2) = \texttt{true}$.

.
The case $Q_{\texttt{S}} = \texttt{e} : \texttt{Elt}$:

```
open SORTING<> + PNAT< .
  ops s1 s2 :  -> Strg .
  op e :  -> Elt .
  var E : Elt
  vars M N P : Nat
```

We introduce the condition of the property

```
  eq disorder(s1) < disorder(s2) = true .
  eq E >> s1 = E >> s2 .
```

and we use again `Lemma-3` and introduce a new lemma on natural numbers.

```
-- [Lemma-3] :
  op _+_ : Nat Nat -> Nat {assoc comm}
  cq [Lemma-5] : M + N < P + N = true if M < P .
```

Now we execute the conclusion of `Lemma-1` for this case:

```
red disorder(e ; s1) < disorder(e ; s2) .
close
```

The case $Q_S = _;_$:

```
open SORTING<> + PNAT< .
  ops x y s1 s2 :  -> Strg .
  vars S S1 S2 : Strg
```

This case involves the following induction hypothesis:

```
  cq disorder(x ; S1) < disorder(x ; S2) = true
        if disorder(S1) < disorder(S2) and S1 <> S2 .
  cq disorder(y ; S1) < disorder(y ; S2) = true
        if disorder(S1) < disorder(S2) and S1 <> S2 .
```

Now we introduce the condition for this case

```
  eq s1 <> s2 = true .
  eq disorder(s1) < disorder(s2) = true .
```

and a new lemma

```
  cq [Lemma-6] : S ; S1 <> S ; S2 = true if S1 <> S2 .
```

and proceed with the reduction of the conclusion for this case:

```
red disorder(x ; y ; s1) < disorder(x ; y ; s2) .
close
```

Proof score for `Lemma-2`.

```
open SORTING<> + PNAT< .
  ops e1 e e' :  -> Elt .
  op s :  -> Strg .
```

We use `Lemma-3` for the third time in our termination proof and introduce a new lemma on natural numbers:

```
-- [Lemma-3] :
  op _+_ : Nat Nat -> Nat {assoc comm}
  eq [Lemma-7] : (M:Nat = M) = true .
```

and execute the conclusion of `Lemma-2`

```
red e1 >> e ; e' ; s = e1 >> e' ; e ; s .
close
```

Proof score for `Lemma-6`.

```
open SORTING<> + PNAT< .
  ops s s1 s2 :  -> Strg .
  op e :  -> Elt .
  var E : Elt
```

This is the second place in our termination proof score that we use the definition of `<>`, which is required by the condition of the lemma:

```
  eq E >> s1 = E >> s2 .
```

We also use again `Lemma-7`:

```
  eq [Lemma-7] : (M:Nat = M) = true .
```

and we proceed with the execution of the conclusion of the lemma

```
red e >> s ; s1 = e >> s ; s2 .
close
```

This completes our termination proof. We have skipped the proof scores of all lemmas about natural numbers, and leave this task to the reader.

Mind semantic traps! The last equation of `PSEUDO-ORDER`, namely

```
 eq (E1 < E2) and (E1 not< E2) = false .
```

has not been used in any of the computations of the termination proof score, which means that the proof score would run and give the desired results without it. Hence this equation seems to be redundant. Apparently this implies termination holds for relations < that do not necessarily satisfy the above mentioned equation. However it is easy to have simple counterexamples of relations < for which $a < b$ and $b < a$ for some a and b, which means an infinite chain of sorting transitions

$$a\,;b \Rightarrow b\,;a \Rightarrow a\,;b \ldots$$

hence non-termination! So, how do we explain this apparent paradox?

In the absence of the above mentioned equation there is the possibility to have for some e, e' both $e >> e' = 0$ and $e >> e' = \mathtt{s}\ 0$ which implies $0 = \mathtt{s}\ 0$. This means that the natural numbers are collapsed which via the equations of `PNAT<` leads to the collapse of the Booleans too, hence our specifications become inconsistent.

The conclusion is very clear: the correctness of a proof score depends intimately upon the semantic correctness of the specification, proof scores are *not* mere proof theoretic entities. If the specification lack semantic correctness, even if the proof score is built correctly and its running gives the desired results, then it is still possible that the conclusion of the proof score may be wrong.

This means we have to take great responsability upon the semantic correctness of our specifications which concretely may imply the necessity to write axioms with semantic meaning but that may have absolutely no operational meaning.

The confluence of the sorting. It is rather easy to see that the preorder relation on strings of naturals induced by the bubble sorting algorithm specified by `SORTING-NAT` is confluent since at the end each string gets rewritten to its sorted form. This property is less obvious when employ other binary relations instead of the standard ordering on the naturals. In fact the confluence may even fail to hold as shown by the example below.

```
mod! CONF-CEX {
  [ Elt ]
  ops a b c :  -> Elt
  op _<_ : Elt Elt -> Bool
  op _not<_ : Elt Elt -> Bool
  eq a < b = true .
  eq b < c = true .
  eq a < c = false .
  eq E:Elt not< E':Elt = not(E < E') .
}
```

Then we have that $(c\,;b\,;a) \Rightarrow (b\,;c\,;a)$ and $(c\,;b\,;a) \Rightarrow (c\,;a\,;b)$ with both $(b\,;c\,;a)$ and $(c\,;a\,;b)$ being normal forms for the preorder induced by the sorting.

Checking confluence by the search commands. Under some conditions, confluence of algorithms can be checked automatically by using special searching commands in CafeOBJ or Maude associated to *POA* specifications. In this area, Maude has a rather special strength. From the set of Maude search commands the most appropriate for this task is `=>!`, which computes all normal forms of any term with respect to the rewriting relation $\xrightarrow{\star}_{\Gamma,E}$ for Γ a set of *POA* Horn clauses and E a set of operation attributes.

In general $t \xrightarrow{\star}_{\Gamma,E} t'$ implies $\Gamma \cup E \models t \Rightarrow t'$ (cf. Cor. 3.23), while the reverse implication does not hold in general. However in this case we have an equivalence, meaning that $\xrightarrow{\star}_{\Gamma,E}$ coincides with the preorder induced by the sorting, hence in order to establish the non-confluence of this preorder we just need to have more than one normal form for some term. Hence the non-confluence example above can be obtained by using the Maude search mechanism as follows.

```
select SORTING-STRG(CONF-CEX) .
search (c ; b ; a) =>! s:Strg .
```

The Maude system gives the following result showing non-confluence.

```
Solution 1 (state 1)
states: 3  rewrites: 5
s:Strg --> b ; c ; a

Solution 2 (state 2)
states: 3  rewrites: 6
s:Strg --> c ; a ; b

No more solutions.
states: 3  rewrites: 6
```

Proving confluence. However confluence of bubble sorting holds when the relation < is transitive. While for establishing non-confluence a counterexample was enough, the confluence need a proof. The character of this proof would again be semi-formal because like in the proof of termination we will use the fact that any sorting transition $s \Rightarrow s'$ with $s \neq s'$ is a finite composition of one-step sorting transitions $s;e';e;s \Rightarrow s;e;e';s'$ with $e < e'$. But the crucial aspect of our confluence proof is the usage of Newman's Lemma 2.67 which together with the termination property proved above reduces the confluence property to the local Church-Rosser property of the one-step sorting transition relation. This can be established through a formal proof score.

Proof score for local Church-Rosser of one-step sorting transition relation. Our proof score distinguished between two cases:
1. The two initial swaps do not overlap:

```
open SORTING-STRG .
  ops e e' e1 e1' :  -> Elt .
  ops s s' s'' :  -> Strg .
```

We introduce the hypothesis:

```
  eq e'  < e  = true .
  eq e1' < e1 = true .
```

The proof of local Church-Rosser for this case consists of two search evaluations, both of them giving the same result. Since, as we have seen above, `=>!` is in general included in the preorder of the initial preordered algebra of the corresponding specification (moreover in this particular example they are equal, but this not necessary here), this is enough for establishing the confluence of the algorithm.

```
search (s ; e' ; e ; s' ; e1 ; e1' ; s'') =>! x:Strg .
search (s ; e ; e' ; s' ; e1' ; e1 ; s'') =>! x:Strg .
```

2. The two initial swaps do overlap.

```
open SORTING-STRG .
  ops e e' e'' :  -> Elt .
  ops s s' :  -> Strg .
```

We introduce the hypothesis.

```
  eq e'  < e  = true .
  eq e'' < e' = true .
```

At this point our proof needs a transitivity hypothesis on < which is introduced as follows:

```
  eq [transitivity] : e'' < e = true .
```

Like in previous case, the proof of local Church-Rosser for this case consists of two search evaluations, both of them giving the same result.

```
search (s ; e' ; e ; e'' ; s') =>! x:Strg .
search (s ; e ; e'' ; e' ; s') =>! x:Strg .
```

Exercises.

3.3. Build and run a proof score for showing that the view `PNAT<asPO` satisfies the axioms of `PSEUDO-ORDER`.

3.4. Consider the following algorithm on strings of natural numbers:

```
vars N M : Nat
trans 0 ; M => M .
trans M ; 0 => M .
ctrans M ; N => (M - N) ; N  if  N < M .
ctrans M ; N => M ; (N - M)  if  M < N .
trans M ; M => M .
```

What does this algorithm compute? Is this algorithm terminating and/or confluent? Justify your answer by proof scores.

3.5. Consider the following problem of simplification of a system of debts between financial agents.

1. Specify a system of debts as a finite multiset of atomic debts. An atomic debt is a triple $(A\ n\ B)$ consisting of two agents A and B and a natural number n representing the fact that the first agent (A) owes n currency units to the second agent (B).
2. Specify an algorithm for reducing systems of debts that uses the following transition:

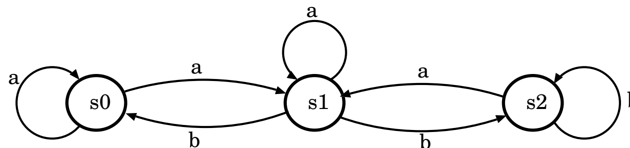
$$(A\ m\ B)(B\ n\ C) \Rightarrow \begin{cases} (A\ m\ C)(B\ n-m\ C) & \text{if } m \leq n \\ (A\ m-n\ B)(A\ n\ C) & \text{otherwise.} \end{cases}$$

3. Is the reducing debts algorithm confluent?
4. Prove that the reducing debts algorithm is terminating. (*Hint*: Define a 'weight' function which gives the total amount of debt in the system.)
5. Define a function that for each agent gives its balance with respect to a given system of debts. Prove that this is an invariant with respect to reduction operations on the systems of debts.

3.4 Example: non-deterministic automata

Non-determinism is a natural aspect of *POA* specifications that, is associated to the situations when the preorders of the denotations of the specifications are not confluent. In algorithmic terms we may say that non-confluent algorithms correspond to non-deterministic computations. We have already met this situation in Sect. 3.1 for the bubble sorting over non-transitive binary relations. In this section we give a special focus to *POA* specification and formal verification of non-determinism. This includes a proof by structural induction of an inductive property in *POA*.

Non-deterministic automata. Automata provides one of the best known examples of non-determinism. The following is an example of non-deterministic automata with three states and two letters or buttons.



The non-deterministic character of this automaton has two aspects. One is that there are states that do not admit transitions (to other states) by some of the letters, e.g. `s0` does not admit a transition by `b`. The second non-deterministic aspect of this automaton is that some states admit transitions to several different states by the same letter, e.g. `s1` has transitions by `b` to both `s0` and `s2`.

Specification of non-deterministic automata. The following is a generic specification of words with concatenation operation over arbitrary vocabulary.

```
mod! WORDS (L :: TRIV) {
  [ Elt < Word ]
  op nil : -> Word
  op __ : Word Word -> Word {assoc}
  var W : Word
  eq nil W = W .
}
```

Our specification of automata consists of a specification of transitions between configurations of states and words. These configurations can also be specified generically as follows.

```
mod! ND-AUT-GEN (L :: TRIV, S :: TRIV) {
  protecting(WORDS(L))
  [ Config ]
  op (_*_) : Elt.S Word -> Config
}
```

The rest of the specification of our automaton has a particular character. We first specify the concrete set of states and the concrete vocabulary used.

```
mod! LETTERS {
  [ Letter ]
  ops a b : -> Letter
}
```

```
mod! STATES {
  [ State ]
  ops s0 s1 s2 : -> State
}
```

Finally, we instantiate the vocabulary and the set of states and specify the transitions of our automaton.

```
mod! ND-AUT {
  protecting(ND-AUT-GEN(L <= view to LETTERS {sort Elt -> Letter},
                        S <= view to STATES  {sort Elt -> State}))
  var W : Word
  trans s0 * a W => s1 * W .
  trans s0 * a W => s0 * W .
  trans s1 * a W => s1 * W .
  trans s1 * b W => s0 * W .
  trans s1 * b W => s2 * W .
  trans s2 * a W => s1 * W .
  trans s2 * b W => s2 * W .
}
```

Running the automata. A word is 'accepted' by an automaton when there exists a chain of transitions from an 'initial' state to a 'final' state. In the case of our automaton let us fix the initial state to `s0` and the final state to `s2`. We may use the search command `=>*` of Maude for establishing whether a certain concrete word is accepted or not by automata. The command `=>*` represents an implementation of the rewriting relation $\xrightarrow{\star}_{\Gamma,E}$ for Γ the set of Horn clauses of the *POA* specification, used as rewrite rule, and E set of operation attributes. According to Cor. 3.23, this means that in general if t `=>*` t' then $\Gamma \cup E \models t \Rightarrow t'$. **discuta si commanda Maude `rewrite` cu diferite strategii (depth-first, fair, etc.)**

We may use this argument to validate the fact that `(a a b a b nil)` is accepted by the automaton by getting a positive answer to the following Maude search query.

```
search s0 * a a b a b nil =>* s2 * nil .
```

Since for this particular example we have that E consists of the monoid equations for words specified as operation attributes, and Γ consists of a set of (unconditional) transitions, it is easy to see that t `=>*` t' if and only if $\Gamma \cup E \models t \Rightarrow t'$. We may use this remark for establishing that `(a b b a nil)` is *not* accepted by the automaton by getting a negative answer to the following Maude search query.

```
search s0 * a b b a nil =>* s2 * nil .
```

An inductive property of the automaton specification. Let us consider the property that `(a W b nil)` is accepted by our automaton for any word `W`. Mathematically this can be written as

$$(\forall \mathtt{W})\ \mathtt{s0} * (\mathtt{a\ W\ b\ nil}) \Rightarrow \mathtt{s2} * \mathtt{nil}. \tag{3.3}$$

This is an inductive property that does hold in the initial preordered algebra of `ND-AUT` that has pairs formed from the states `s0`, `s1`, or `s2` and words over the vocabulary $\{\mathtt{a}, \mathtt{b}\}$. Its proof is based upon the following lemma:

$$(\forall \mathtt{W})\ (\mathtt{s1} * (\mathtt{W\ b\ nil}) \Rightarrow \mathtt{s2} * \mathtt{nil}) \wedge (\mathtt{s2} * (\mathtt{W\ b\ nil}) \Rightarrow \mathtt{s2} * \mathtt{nil}). \tag{3.4}$$

Therefore the proof score of (3.3) goes as follows:

```
open ND-AUT .
  op w :  -> Word .
  var W : Word
  var S : State
```

We introduce the lemma (3.4):

```
  ctrans S * W b nil => s2 * nil if (S == s1) or (S == s2) .
```

The proof of (3.3) can be now performed by the following search command

```
search s0 * a w b nil =>* s2 * nil .
close
```

The proof of lemma (3.4). This proof is done by structural induction on `W` by using the *POA* interpretation of Prop. 2.86. We consider a sub-signature of constructors formed by `s0`, `s1`, `s2`, `nil`, `a_`, and `b_`, where by `a_` and `b_`, respectively, we mean the concatenation to the front of the words with `a` and `b`, respectively. We leave to the reader the task to prove this rather obvious fact. The proof score for our lemma goes as follows.

```
open ND-AUT .
  op w :  -> Word .
  var W : Word
  var S : State
```

The proof for case $Q_{\mathtt{W}} = \mathtt{nil}$:

```
search s1 * nil b nil =>* s2 * nil .
search s2 * nil b nil =>* s2 * nil .
```

The following induction hypothesisis common to both cases $Q_{\mathtt{W}} = \mathtt{a_}$ and $Q_{\mathtt{W}} = \mathtt{b_}$.

```
  ctrans S * w b nil => s2 * nil if (S == s1) or (S == s2) .
```

The proof for case $Q_{\mathtt{W}} = \mathtt{a_}$:

```
search s1 * a w b nil =>* s2 * nil .
search s2 * a w b nil =>* s2 * nil .
```

The proof for case $Q_{\mathtt{W}} = \mathtt{b_}$:

```
search s1 * b w b nil =>* s2 * nil .
search s2 * b w b nil =>* s2 * nil .
close
```

All search commands of our proof score (including both the proof of the main property (3.3) and of lemma (3.4)) give a positive answer, and their usage in such proof scores is justified by the fact that in general t `=>*` t' implies that $t \Rightarrow t'$ is a consequence of the corresponding *POA* axioms.