

3 From Total Equational to Partial First Order Logic*

Maura Cerioli¹, Till Mossakowski², and Horst Reichel³

¹ DISI – Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova – Via Dodecaneso, 35 – Genova 16146 – Italy
cerioli@disi.unige.it <http://www.disi.unige.it/person/CerioliM/>

² Universität Bremen – FB Mathematik und Informatik
Postfach 330440 – D-28334 Bremen – Germany
till@informatik.uni-bremen.de

<http://www.informatik.uni-bremen.de/~till/>
³ TU Dresden – Fakultät Informatik – Institut für Theoretische Informatik
D-01062 Dresden – Germany
reichel@tcs.inf.tu-dresden.de
<http://www.tcs.inf.tu-dresden.de/~reichel/english-index.html>

Abstract. The focus of this chapter is the incremental presentation of partial first-order logic, seen as a powerful framework where the specification of most data types can be directly represented in the most natural way. Both model theory and logical deduction are described in full detail.

Alternatives to partiality, like (variants of) error algebras and order-sortedness are also discussed, showing their uses and limitations.

Moreover, both the total and the partial (positive) conditional fragment are investigated in detail, and in particular the existence of initial (free) models for such restricted logical paradigms is proved.

Some more powerful algebraic frameworks are sketched at the end.

Equational specifications introduced in last chapter, are a powerful tool to represent the most common data types used in programming languages and their semantics. Indeed, Bergstra and Tucker have shown in a series of papers (see [BT87] for a complete exposition of results) that a data type is semicomputable if and only if it is (isomorphic to) the initial model of a finite set of equations over a finite set of symbols.

However this result has two main limitations.

The first point is that initial approach is appropriate only if the specifying process of the data type has been completed, because it defines one particular realization (up to isomorphism), instead of a class of possible models, still to be refined. In particular, if the data type has partial functions, the treatment for the “erroneous” elements must be already fixed in all details.

* A preliminary and somehow extended version of this chapter, including proofs, is available as technical report DISI-TR-96-25 of DISI - University of Genova (Italy) at <ftp://ftp.disi.unige.it/person/CerioliM/IFIP96.ps.gz>.

The second, more problematic, point is that, since the expressive power of the logic used to axiomatize the data types is so poor, quite often it is not possible to define the intended data type through its abstract properties, but it is necessary to describe one of its possible implementations. Technically speaking, in order to define a data type, auxiliary types and operators can be needed, drastically decreasing the abstractness level of the specification and reducing its readability and naturalness. Consider, indeed, the following example, showing an artificial but simple data type, that cannot be finitary equationally specified. Other, far more interesting, data types, like the algebra of regular sets, cannot be expressed by a finite set of equations as well, but the proof that more powerful logics are needed is made too complex by their richer structure.

Example 3.1. We want to specify a data type having sorts for the natural numbers and for their quotient identifying all odd numbers, with the usual constructors for the natural numbers and an operation associating each number with its equivalence class. Thus, the signature of the type should be the following.

```

sig  $\Sigma_{\text{Nat}}$  =
  sorts   nat, nat/ $\equiv$ 
  opns   zero:  $\rightarrow$  nat
           succ: nat  $\rightarrow$  nat
           nat $\equiv$ : nat  $\rightarrow$  nat/ $\equiv$ 

```

Let us see whether we can give a finite set E of equations on this signature in a way that the initial model of such a specification is (isomorphic to) our intended data type.

First of all note that our set E cannot contain any non-trivial equation of sort **nat**. Indeed, using $f^k(\dots)$ to denote the iterative application of any function f a number k of times, an equation of sort **nat** can have (up to symmetry) four forms.

- $\text{succ}^k(\text{zero}) = \text{succ}^n(\text{zero})$ that is either trivial or not valid in our intended model, as the term $\text{succ}^k(\text{zero})$ is interpreted by the number k .
- $\text{succ}^k(\text{zero}) = \text{succ}^n(x)$ that cannot be satisfied by a non-trivial model, because the left-hand side is interpreted as a constant and the right-hand side changes its value depending on the interpretation of x ; in particular in our intended model it does not hold if $k + 1$ is substituted for x .
- $\text{succ}^k(x) = \text{succ}^n(x)$ that is either trivial or does not hold in our intended model if 0 is substituted for x .
- $\text{succ}^k(x) = \text{succ}^n(y)$ with x and y distinct variables, that does not hold if 0 is substituted for x and $k + 1$ for y .

Let us analyze, analogously, the non-trivial equalities of sort **nat**/ \equiv to see if they can belong to E . Since terms of sort **nat**/ \equiv are given by the application of $\text{nat}\equiv$ to terms of sort **nat**, we have again four cases.

$nat_{\equiv}(\text{succ}^k(\text{zero})) = nat_{\equiv}(\text{succ}^n(\text{zero}))$ that is not valid in our intended model if k or n is even (unless $k = n$, in which case is trivial), while it is valid whenever both k and n are odd.

$nat_{\equiv}(\text{succ}^k(\text{zero})) = nat_{\equiv}(\text{succ}^n(x))$ that is not valid in our intended model, because $nat_{\equiv}(\text{succ}^k(\text{zero}))$ is interpreted as a constant (either k , if k is even, or the class of all odd numbers), while the value of $nat_{\equiv}(\text{succ}^n(x))$ changes, and in particular $nat_{\equiv}(\text{succ}^n(\text{zero}))$ and $nat_{\equiv}(\text{succ}^{n+1}(\text{zero}))$ have different interpretations, the classes of an even and an odd number respectively, so $nat_{\equiv}(\text{succ}^k(\text{zero})) = nat_{\equiv}(\text{succ}^n(x))$ cannot be true both if 0 or if 1 is substituted for x .

$nat_{\equiv}(\text{succ}^k(x)) = nat_{\equiv}(\text{succ}^n(x))$ that is either trivial or does not hold in our intended model, because if k or n is even it is not satisfied substituting 0 for x , else it is not satisfied substituting 1 for x .

$nat_{\equiv}(\text{succ}^k(x)) = nat_{\equiv}(\text{succ}^n(y))$ with x and y distinct variables, that does not hold if k is substituted for x and $n + 2k + 2$ for y .

Therefore, all non-trivial equations in E must have the form

$$nat_{\equiv}(\text{succ}^{2k+1}(\text{zero})) = nat_{\equiv}(\text{succ}^{2n+1}(\text{zero})).$$

Any such equation can only identify the result of the interpretation of nat_{\equiv} on the two odd numbers $2k + 1$ and $2n + 1$. Thus, if E is finite, only a finite number of identities between terms of the form $nat_{\equiv}(\text{succ}^{2k+1}(\text{zero}))$ can be inferred.

Therefore, there is not a finite equational specification of the required data type using the signature Σ_{Nat} . However, if we enrich the signature, we can define the data type, using the extra symbols. Indeed, let us consider the following specification.

```

spec Odd = enrich  $\Sigma_{\text{Nat}}$  by
  sorts   bool
  opns   true, false :  $\rightarrow$  bool
         odd : nat  $\rightarrow$  bool
         cond : bool  $\times$  nat  $\times$  nat  $\rightarrow$  nat
  vars   x, y : nat
  axioms cond(true, x, y) = x
         cond(false, x, y) = y
         odd(zero) = false
         odd(succ(zero)) = true
         odd(succ(succ(x))) = odd(x)
          $nat_{\equiv}(x) = \text{cond}(\text{odd}(x), nat_{\equiv}(\text{succ}(\text{zero})), nat_{\equiv}(x))$ 

```

■

Roughly speaking, equational specifications are sufficiently expressive to initially define any semicomputable (total) data type, because recursive functions can be described using recursion and conditional choice. But recursion is immediately embedded in the equational framework, as recursive definitions *are* given by equalities, and Booleans and conditional choice can be

equationally implemented, as in the previous example. Thus, the intuition here is that if the logic used in specifications is poor, for instance equational, complex data types can be expressed as well, by implementing *inside* the data type a “Boolean” sort with operations to represent logical connectives and translating any assertion ϕ at the metalevel into an equation between the Boolean term corresponding to ϕ and the constant value `true`.

Since all logical connectives can be equationally described, all theories of predicate calculus (without quantifiers) can be translated into an equational specification with *hidden sorts and operations*, that is, adding auxiliary symbols to the data type, that should be not exported to the users of the specification. However, the resulting specification lacks of abstraction, because what logically is a statement on the data type has been implemented by an equation between elements of the data type itself. In other words, the equational specification is actually an *implementation* of the natural axiomatic description of the data type. This in particular implies that we have to fix the data type of Booleans to have just two elements. Such a thing cannot be done, though, within positive conditional logics (which are the logics having a nearly-executable proof theory).

This chapter will be devoted to introduce an algebraic framework sufficiently expressive in order to *directly* represents the most common data types. As we have seen, the first obstacle to overrun is the limitation of the formulae that can be used to specify the data types. Thus, we need a richer logic, but not too rich. Indeed, we want to keep the logical language easy to read and to implement, in order to have tools for rapid prototyping of the data types. Moreover, we do not want to lose the initial semantic approach. Thus, our formulae should be able to describe only classes of algebras having an initial object.

A far more challenging problem is the specification of partial functions. Indeed, many data types in practice have partial operations, whose result on some input is “erroneous”. Sometimes such errors can be avoided simply using a better typing, as it is the case, in a programming language with declarations

```
type my_array=array[1..k] of T
var i:integer;
    A: my_array;
```

for expressions of the form $A[i]$ if i assumes values outside the array range, but that could be forbidden declaring i of type $1..k$.

Even if a better typing is not possible (or not convenient), most errors can be statically detected and hence axioms to identify them to some “error element”, representing an error message, can be given.

But, whenever a partial recursive function that has no recursive domain has to be specified, it is obviously not possible to detect the errors introduced by its application. Hence there does not exist a (total) specification of the function identifying all its erroneous applications to some “error”. Note that

partial recursive functions without recursive domain are needed, for instance, whenever describing the semantics of (universal) programming languages. Hence any algebraic approach has to deal with them in some way – otherwise, it can be used only to describe the data types of a program but not to verify properties of the programs using them.

As usual, the easier the theory, the harder its use. Thus, in the total equational framework, having nice and intuitive semantics and efficient rewriting techniques, specifications of complex data types are often hard to find (if any). On the other hand, making the framework powerful can make its theory too complex and hence hinder its understanding by the users. Here we will incrementally introduce a very expressive partial framework, showing how and when its features are needed or simply convenient, so that users can restrict themselves to some subtheory of it, if dealing with sufficiently easy problems.

3.1 Conditional axioms

Following the guideline of Example 3.1, we need a way to impose equations only to those values that satisfy a condition. This was implemented in that example by introducing the operation `cond`, corresponding to a conditional choice, and then imposing the axiom

$$\text{nat}\equiv(x) = \text{cond}(\text{odd}(x), \text{nat}\equiv(\text{succ}(\text{zero})), \text{nat}\equiv(x))$$

that corresponds, logically, to requiring

$$\text{nat}\equiv(x) = \text{nat}\equiv(\text{succ}(\text{zero})) \text{ if } \text{odd}(x).$$

Thus, we move from *equational* to *equational conditional*, or simply conditional, specifications. Therefore, in the following the axioms will have the form

$$t_1 = t'_1 \wedge \dots \wedge t_n = t'_n \Rightarrow t = t'$$

that is *satisfied* by a valuation if the consequence $t = t'$ is satisfied whenever all the premises $t_i = t'_i$ are satisfied and *holds* in a model iff it is satisfied by all valuations in that model (see Definition 2.7.1 for the formal details).

Although, as shown in [BT87], conditional specifications, as well as equational ones, need hiding sorts and operations to define all semicomputable total data types, they are strictly more expressive than equational axioms, because the data type introduced in Example 3.1, that cannot be axiomatized by a finite set of equations on its signature, can be easily defined by the following conditional axiom

$$\phi_{\text{odd}} \quad \text{nat}\equiv(x) = \text{nat}\equiv(\text{succ}(\text{zero})) \Rightarrow \text{nat}\equiv(\text{succ}(\text{succ}(x))) = \text{nat}\equiv(x)$$

But note that the above specification works, because all even number are distinct from the odd ones. Indeed, let us consider the same problem, but

with \mathbf{nat}/\equiv the quotient identifying all odd numbers *and* 0. Then the axiom $\phi_{\mathbf{odd}}$ is incorrect, because if the classes of 0 and 1 coincide when instantiating x on 0, we identify all integers.

The point is that the informal specification of \mathbf{nat}/\equiv is based on the distinction between odd and even numbers, but our signature does not have syntactical means to express this concept. Thus, although using conditional axioms we actually enrich the expressive power of our logic, the logic we get is still too poor, because the atoms we can use to build axioms are only equations, while we would need symbols to state that a number is even or odd. Of course we can always use the same trick, implementing a Boolean sort with an \mathbf{odd} Boolean function, but it is much more clear to add a facility to our specification framework, providing symbols for *predicates*.

Definition 3.2. A *first-order signature* Σ is a triple (S, Ω, Π) where

- (S, Ω) is a many-sorted signature;
- Π is an S^* -sorted family (of *predicate symbols*).

Given a first-order signature $\Sigma = (S, \Omega, \Pi)$, the Σ -terms on an S -sorted family of variables X , denoted by $T_\Sigma(X)$, are the many-sorted term algebra $T_{(S, \Omega)}(X)$ on the many-sorted signature underlying Σ . \diamond

Example 3.3. A reasonable signature for the Example 3.1, then is the following.

```
sig  $\Sigma_{\mathbf{Nat}P}$  = enrich  $\Sigma_{\mathbf{Nat}}$  by
  preds  is_odd: nat
```

The signature $\Sigma_{\mathbf{Nat}P}$ completely captures our intuition that we want to enrich the natural numbers by the new sort of their quotient and that to describe the equivalence relation we discriminate odd from even numbers and hence we need a symbol stating whether a number is odd. \blacksquare

In the rest of this section, let $\Sigma = (S, \Omega, \Pi)$ be a first-order signature.

In each Σ -structure predicate symbols are interpreted by their truth set.

Definition 3.4. A Σ -structure consists of

- an (S, Ω) many-sorted algebra A , called the *underlying many-sorted algebra*;
- for each $p: s_1 \times \cdots \times s_n \in \Pi$ a subset p_A of $|A|_{s_1} \times \cdots \times |A|_{s_n}$, representing the *extent* of p in A , that is the tuples of elements on which the predicate is true.

Given two Σ -structures A and B , a *homomorphism of Σ -structures* from A into B is a truth preserving homomorphism of many-sorted algebras between the underlying many-sorted algebras, that is a homomorphism $h: A \rightarrow B$ s.t. if $(a_1, \dots, a_n) \in p_A$, then $(h_{s_1}(a_1), \dots, h_{s_n}(a_n)) \in p_B$ for all $p: s_1 \times \cdots \times s_n \in \Pi$ and all $a_i \in |A|_{s_i}$ for $i = 1, \dots, n$.

Let C be a class of Σ -structures. A Σ -structure I is *initial* in C iff $I \in C$ and for each $A \in C$ a unique homomorphism of Σ -structures $!_A: I \rightarrow A$ exists.

Given a Σ -structure A and an S -sorted family of variables X , *variable valuations* and *term evaluations* for $T_\Sigma(X)$ in A are, respectively, variable valuations and term evaluations for $T_\Sigma(X)$ in the many-sorted algebra underlying A . \diamond

Notice the difference between enriching a signature by a Boolean sort and some operations, as in the equational presentation of Example 3.1, and by predicates. Indeed, in the former case the models are *larger* than the models we are interested in, in the sense that sets and functions have been added to their structure. Instead, in the latter the models are *richer*, because they are the same algebras, as collections of sets and functions, but the language we use to handle them is more expressive (and correspondingly we need now to know how to interpret in them some more condition). Thus, for instance, we have the same number of elements, but we know each element better and are, hence, able to state the (un)truth of some property on them.

Consider for instance once again the Example 3.1. Then a model of the specification `Odd` is the algebra

$$\begin{aligned}
 &\mathbf{algebra\ } \mathbf{N}_{eq} = \\
 &\quad \mathbf{Carriers} \\
 &\quad \quad |\mathbf{N}_{eq}|_{\mathbf{nat}} = \mathbf{N} \\
 &\quad \quad |\mathbf{N}_{eq}|_{\mathbf{nat}/\equiv} = 2\mathbf{N} \cup \{\bar{1}\} \\
 &\quad \quad |\mathbf{N}_{eq}|_{\mathbf{bool}} = \{T, F\} \\
 &\quad \mathbf{Functions} \\
 &\quad \quad \mathbf{zero}_{\mathbf{N}_{eq}} = 0 \\
 &\quad \quad \mathbf{true}_{\mathbf{N}_{eq}} = T \\
 &\quad \quad \mathbf{false}_{\mathbf{N}_{eq}} = F \\
 &\quad \quad \mathbf{succ}_{\mathbf{N}_{eq}}(n) = n + 1 \\
 &\quad \quad \mathbf{plus}_{\mathbf{N}_{eq}}(n, m) = n + m \\
 &\quad \quad \mathbf{nat}\equiv_{\mathbf{N}_{eq}}(n) = \begin{cases} n, & \text{if there is } k \text{ s.t. } n = 2k \\ \bar{1}, & \text{otherwise} \end{cases} \\
 &\quad \quad \mathbf{cond}_{\mathbf{N}_{eq}}(b, n, m) = \begin{cases} n, & \text{if } b = T \\ m, & \text{otherwise} \end{cases} \\
 &\quad \quad \mathbf{odd}_{\mathbf{N}_{eq}}(n) = \begin{cases} F, & \text{if there is } k \text{ s.t. } n = 2k \\ T, & \text{otherwise} \end{cases}
 \end{aligned}$$

and \mathbf{N}_{eq} consists of the algebra we wanted, that is its $\Sigma_{\mathbf{Nat}}$ -reduct, plus a set and a bunch of functions. Instead, using predicates we get the first-order structure

$$\begin{aligned}
 &\mathbf{algebra\ } \mathbf{N}_p = \\
 &\quad \mathbf{Carriers} \\
 &\quad \quad |\mathbf{N}_p|_{\mathbf{nat}} = \mathbf{N} \\
 &\quad \quad |\mathbf{N}_p|_{\mathbf{nat}/\equiv} = 2\mathbf{N} \cup \{\bar{1}\} \\
 &\quad \mathbf{Functions}
 \end{aligned}$$

$$\begin{aligned}
\mathbf{zero}_{\mathbf{N}_p} &= 0 \\
\mathbf{succ}_{\mathbf{N}_p}(n) &= n + 1 \\
\mathbf{plus}_{\mathbf{N}_p}(n, m) &= n + m \\
\mathbf{nat}\equiv_{\mathbf{N}_p}(n) &= \begin{cases} n, & \text{if there is } k \text{ s.t. } n = 2k \\ \bar{1}, & \text{otherwise} \end{cases} \\
\mathbf{Predicates} & \\
\mathbf{is_odd}_{\mathbf{N}_p} &= \{n \mid \exists k \in \mathbf{N} \text{ s.t. } n = 2k + 1\}
\end{aligned}$$

that is exactly what we wanted, enriched by the information of which elements of its carrier are odd.

Notice that with positive conditional axioms and predicates, we cannot talk about falsehood of predicates, while the approach of enriching a signature by a Boolean sort and treating predicates as operations onto this sort does not have such a restriction. Thus, apparently the latter approach is richer. But the capability of using negative information has the drawback that, if the Boolean sort has to contain only the interpretations of `true` and `false`, the untruth has to be stated as well as the truth of each relation. Thus, in particular, semicomputable relations cannot be conditionally axiomatized, because their falseness cannot be recursively axiomatized.

Of course, it is possible to ensure that the Boolean sort has just two elements, using a more complex (first-order) axiom, without specifying the actual result of the application of the functions representing relations. However, in this case the simpler proof theory of positive conditional axioms cannot be used and the existence of free extensions is not guaranteed, that is, we may lose the capability of extending the models of some data type in a uniform way, because the new operations yield Boolean terms whose interpretation could be true as well as false.

Another merit of the approach using predicates is the easy specification of inductively defined relations using initial or free semantics (or initiality or freeness constraints). This is possible only with the predicate approach, which combines the property of the existence of initial models (and free extensions) with the flexibility of homomorphisms (which have to preserve truth, but not falsehood). With this, it is possible to specify the minimal relation satisfying some set of axioms. Using free extensions, for example transitive closure can be specified by just stating that the transitive closure contains the original relation and is transitive.¹ Note that initiality constraints are a second-order principle, so proof theory here gets more complex, since an induction principle is needed.

Let us now formally define conditional axioms and their validity.

¹ When trying to specify transitive closure in a purely functional style, we either get additional truth values in the free extension, or equate `true` and `false`. Adding a first-order axiom stating that there are exactly two truth values destroys the existence of free extensions. These problems have the origin that homomorphisms are not so flexible: they have to preserve both truth and falsehood.

Definition 3.5. Let Σ be the first-order signature (S, Ω, Π) and X be an S -sorted family of variables. The set of Σ -atoms on X is

$$At(\Sigma, X) = \{t = t' \mid t, t' \in |T_\Sigma(X)|_s\} \cup \{p(t_1, \dots, t_n) \mid p: s_1 \times \dots \times s_n \in \Pi \text{ and } t_i \in |T_\Sigma(X)|_{s_i}, i = 1, \dots, n\}$$

The set of Σ -conditional axioms on X is

$$Cond(\Sigma, X) = \{\forall X. \epsilon_1 \wedge \dots \wedge \epsilon_n \Rightarrow \epsilon_{n+1} \mid \epsilon_i \in At(\Sigma, X), i = 1, \dots, n+1\}$$

◇

In other words, conditional axioms are positive Horn-Clauses, built using the predicates in Π and the equality symbol. As for many-sorted algebras, quantification is explicit to avoid inconsistent deductions in the case of possibly empty carriers.

Definition 3.6. Given a Σ -structure A , we say that A *satisfies* a conditional axiom $\forall X. \varphi \in Cond(\Sigma, X)$ (denoted by $A \models_\Sigma \forall X. \varphi$) if all valuations v for X in A *satisfy* φ (denoted by $v \Vdash \varphi$), where satisfaction of a conditional axiom by a valuation is defined by the following rules:

- $v \Vdash t = t'$ iff $v^\#(t) = v^\#(t')$
- $v \Vdash p(t_1, \dots, t_n)$ iff $(v^\#(t_1), \dots, v^\#(t_n)) \in p_A$
- $v \Vdash \epsilon_1 \wedge \dots \wedge \epsilon_n \Rightarrow \epsilon_{n+1}$ iff $v \Vdash \epsilon_{n+1}$ or there is an ϵ_i s.t. $v \not\Vdash \epsilon_i$

A *presentation* consists of a first-order signature Σ and a set AX of Σ -conditional axioms. The class of models of a presentation $Sp = \langle \Sigma, AX \rangle$, denoted by $Mod(Sp)$, consists of all those Σ -structures satisfying the axioms in AX . ◇

Exercise 3.7. Generalize the notion of signature morphism, reduct and sentences translation and prove the satisfaction lemma for first-order structures with conditional axioms.

Most of the theory of many-sorted algebras carries over to Σ -structures smoothly, though the behavior of the model theory of many-sorted first-order structures resembles more that of the model theory for *partial* algebras/first-order structures (which is discussed in the next section) than that for total algebras.

The model-categories of total algebras are of a more “algebraic” flavor, while model categories of first-order structures and partial algebras have a more “topologically algebraic” flavor. In [AHS90], a precise mathematical background is given for this. The main point is that both for first-order structures and partial algebras, there are bijective homomorphisms that are not isomorphisms, while for total algebras, bijective homomorphisms are always isomorphisms. Thus the notions of full and closed homomorphism

and their interconnection with relative and closed substructures (Proposition 3.38) make already sense for first-order structures. But since first-order structures are a special case of partial first-order structures, we refer to Section 3.3.1 for details.

However, we present a sound and complete calculus for conditional axioms, that we will “borrow” for the partial case as well, and show how the calculus itself defines the initial (free) model for a presentation. Indeed, conditional axioms, as in the case without predicates, define *quasi varieties* and hence their model classes always admit initial models, which may be “computed” by a proof calculus.

Definition 3.8. Let $\Sigma = (S, \Omega, \Pi)$ be a first-order signature. The \vdash inference system consists of the following axioms and inference rules, where we assume that, as usual, ϵ and η , possibly decorated, are atoms over Σ , φ is a conditional axiom over Σ , Φ is a countable set of conditional axioms over Σ , X and Y are S -sorted family of variables, and t, t', t'', t_i, t'_i are Σ -terms.

Congruence Axioms

$$\begin{aligned} \Phi \vdash \forall X. t &= t \\ \Phi \vdash \forall X. t = t' &\Rightarrow t' = t \\ \Phi \vdash \forall X. t = t' \wedge t' = t'' &\Rightarrow t = t'' \\ \Phi \vdash \forall X. t_1 = t'_1 \wedge \dots \wedge t_n = t'_n &\Rightarrow f(t_1, \dots, t_n) = f(t'_1, \dots, t'_n) \\ \Phi \vdash \forall X. t_1 = t'_1 \wedge \dots \wedge t_n = t'_n \wedge p(t_1, \dots, t_n) &\Rightarrow p(t'_1, \dots, t'_n) \end{aligned}$$

Proper Axioms

$$\Phi \vdash \forall X. \varphi \quad \text{for } \forall X. \varphi \in \Phi$$

Substitution

$$\frac{\Phi \vdash \forall X. \varphi}{\Phi \vdash \forall Y. \varphi[\theta]} \quad \text{for } \theta: X \longrightarrow |T_\Sigma(Y)|$$

Here, $\varphi[\theta]$ is the formula φ with each term in it translated by $\theta^\#$.

Cut Rule

$$\frac{\Phi \vdash \forall X. \epsilon_1 \wedge \dots \wedge \epsilon_n \Rightarrow \eta_i \quad \Phi \vdash \forall Y. \eta_1 \wedge \dots \wedge \eta_k \Rightarrow \epsilon}{\Phi \vdash \forall X \cup Y. \eta_1 \wedge \dots \wedge \eta_{i-1} \wedge \epsilon_1 \wedge \dots \wedge \epsilon_n \wedge \eta_{i+1} \wedge \dots \wedge \eta_k \Rightarrow \epsilon} \quad \diamond$$

Proposition 3.9. *The calculus introduced in Definition 3.8 is sound, that is if $\Phi \vdash \forall X. \varphi$ and $M \models \Phi$, then $M \models \forall X. \varphi$.*

Exercise 3.10. Using the congruence axioms and the cut rule show that $\Phi \vdash \forall X. \epsilon \Rightarrow \epsilon$ for all atoms ϵ on X .

As for the equational case, also here the proposed calculus is complete w.r.t. equations without variables and the proof is done by building a reachable model satisfying exactly the deduced equations. Therefore, since the calculus is sound too, that model is initial, satisfying the *no-junk* \mathcal{E} *no-confusion* conditions.

Definition 3.11. Let $Sp = \langle \Sigma, AX \rangle$ be a presentation and $\Theta \subseteq At(\Sigma, X)$ be a finite set of atoms. Then the structure $F_{Sp}(X, \Theta)$ has as underlying algebra $T_\Sigma(X)/\equiv_\Theta$, where $t \equiv_\Theta t'$ if and only if $AX \vdash \forall X. \bigwedge \Theta' \Rightarrow t = t'$ for some $\Theta' \subseteq \Theta$, and a predicate $p: s_1 \times \cdots \times s_n \in \Pi$ is interpreted as

$$\{ ([t_1]_{\equiv_\Theta}, \dots, [t_n]_{\equiv_\Theta}) \mid AX \vdash \forall X. \bigwedge \Theta' \Rightarrow p(t_1, \dots, t_n) \text{ for some } \Theta' \subseteq \Theta \}$$

where we denote by $[t]_{\equiv_\Theta}$ the equivalence class w.r.t. \equiv_Θ of any term t . \diamond

It is immediate to verify that \equiv_Θ is a many-sorted congruence, because of the first four axioms and the Cut Rule. Moreover, because of the fifth axiom and the Cut Rule, the interpretation of p in $F_{Sp}(X, \Theta)$ is well defined.

Lemma 3.12. *Each valuation $v: Y \rightarrow F_{Sp}(X, \Theta)$ can be factorized (in general not uniquely) through $[_]_{\equiv_\Theta}: T_\Sigma(X) \rightarrow F_{Sp}(X, \Theta)$ as follows*

$$\begin{array}{ccc} Y & \xrightarrow{v} & F_{Sp}(X, \Theta) \\ & \searrow \bar{v} & \nearrow [_]_{\equiv_\Theta} \\ & T_\Sigma(X) & \end{array}$$

Moreover, for any \bar{v} s.t. $[_]_{\equiv_\Theta} \circ \bar{v} = v$ and any atom $\epsilon \in At(\Sigma, Y)$, we have $v \Vdash \epsilon$ if and only if $AX \vdash \forall X. \bigwedge \Theta' \Rightarrow \bar{v}^\#(\epsilon)$ for some $\Theta' \subseteq \Theta$.

Theorem 3.13. *Using the notation of Definition 3.11,*

1. $F_{Sp}(X, \Theta)$ is a Sp -algebra.
2. The valuation $\iota: X \rightarrow F_{Sp}(X, \Theta)$ given by $\iota(x) = [x]_{\equiv_\Theta}$ satisfies Θ and is universal w.r.t. this property, i.e. for any valuation $v: X \rightarrow A$ into a Sp -algebra A satisfying Θ , there exists exactly one homomorphism $\tilde{v}: F_{Sp}(X, \Theta) \rightarrow A$ with $\tilde{v} \circ \iota = v$

$$\begin{array}{ccc} X & \xrightarrow{\iota} & F_{Sp}(X, \Theta) \\ & \searrow v & \nearrow \tilde{v} \\ & A & \end{array}$$

3. If $\Theta = \emptyset$ then $F_{Sp}(X, \Theta)$ and ι are the free object in the model class of AX .

The calculus proposed can deduce all conditional formulae valid in all models in a stronger form, that is with possibly less premises. We call this property *practical completeness*. Obviously any practically complete system can be made complete in the usual sense by adding a weakening rule of the form

$$\frac{\Phi \vdash \forall X. \Theta \Rightarrow \epsilon}{\Phi \vdash \forall X. \Theta' \Rightarrow \epsilon} \quad \text{for } \Theta \subseteq \Theta'$$

that we do not include, as the resulting system would be less efficient.

Theorem 3.14. *The calculus is practically complete, that is if*

$$\Phi \models \forall X. \bigwedge \Theta \Rightarrow \epsilon,$$

then

$$\Theta' \subseteq \Theta \text{ exists s.t. } \Phi \vdash \forall X. \bigwedge \Theta' \Rightarrow \epsilon.$$

If $\Theta = \emptyset$, $F_{Sp}(X.\Theta)$ is called the *free Sp-structure over X*, written $F_{Sp}(X)$. If, moreover, $X = \emptyset$, $F_{Sp}(X)$ is called the *initial Sp-structure*, written I_{Sp} .

Let us see an incremental use of conditional specifications to describe a data type as initial model of a presentation.

Example 3.15. Let us consider the problem of the definition of a very primitive dynamic data type that is a subset of the CCS language. The *Calculus of Communicating Systems* has been introduced to study in isolation the problems due to concurrency and to describe reactive systems. It provides primitives to express the interactions between the components of complex systems. The intuition is that there is a set of *agents*, who can perform *actions* either individually or in cooperation with each other. The description of the activity of an agent at a fixed instant cannot be given by a function, because our agents are able to perform non-deterministic choices. Therefore, the activity is defined as a *transition* predicate stating how an agent, performing an action, evolves in another.

The starting point is the specification of the possible actions, that we assume given by the specification **Action**². Although such a specification can be as complex as needed by the concrete problem of concurrency we want to describe, at this level the only interesting feature is that each action determines a *complementary* action, representing the subject/object viewpoint of two interacting agents. For instance, the complementary action of *sending* a message is *receiving* it (and vice versa). Moreover, there is an *internal* action τ given by the composition of an action with its complement, representing the abstraction of a system, composed by two agents interacting between them, that perform a change of its internal state without effects on the external world. Here and in the sequel we use the notation $_$ to denote the place of operands in an infix notation.

² In this case, as in most complex examples, the data type we are interested in is defined using simpler data types, that, of course, should be not modified. The elementary data preservation is guaranteed, in this and in the following examples, by two stronger properties: no operations with an elementary type as result are added and all the equations (and predicate assertions) in the consequences of the axioms are of non-elementary sort (involve non-elementary predicates).

```

spec Action_ =
  sorts   Act
  opns    $\tau: \rightarrow \text{Act}$ 
          $\text{bar}: \text{Act} \rightarrow \text{Act}$ 
  vars    $a: \text{Act}$ 
  axioms  $\text{bar}(\tau) = \tau$ 
          $\text{bar}(\text{bar}(a)) = a$ 

```

As an example of actions, we can think of instructions like `send` or `receive`.

Now we add the sort `Agents` with the idle agent, that cannot perform any action, operations for prefixing an action, parallel composition and non-deterministic choice. The dynamic aspects of the data type are captured by a *transition* predicate.

```

spec CCS_ = enrich Action_ by
  sorts   Agents
  opns    $\lambda: \rightarrow \text{Agents}$ 
          $\_ \_ : \text{Act} \times \text{Agents} \rightarrow \text{Agents}$ 
          $\_ | \_ , \_ + \_ : \text{Agents} \times \text{Agents} \rightarrow \text{Agents}$ 
  preds   $\_ \Rightarrow \_ : \text{Agents} \times \text{Act} \times \text{Agents}$ 
  vars    $a: \text{Act}; p, q, p', q' : \text{Agents}$ 
  axioms  $p + \lambda = p$ 
          $p | \lambda = p$ 
          $p + q = q + p$ 
          $p | q = q | p$ 
          $a.p \Rightarrow p$ 
          $p \xrightarrow{a} p' \Rightarrow p + q \xrightarrow{a} p'$ 
          $p \xrightarrow{a} p' \Rightarrow p | q \xrightarrow{a} p' | q$ 
          $p \xrightarrow{a} p' \wedge q \xrightarrow{\text{bar}(a)} q' \Rightarrow p | q \xrightarrow{\tau} p' | q'$ 

```

The axioms stating equalities between agents capture the properties of the operations between agents, but notice that there are agents having the same transition capability that are not identified, as, for instance, $(a.\lambda + b.\lambda) | c.\lambda$ and $(a.\lambda | c.\lambda) + (b.\lambda | c.\lambda)$, that both can perform either a or b and then are in a situation where c is the only move available.

Thus, the given axioms leave open many different semantics, defined at a meta-level in terms of the action capabilities of agents. But more restrictive axioms could be imposed as well to describe more tightly the operations on agents, for example a distributive law $(p+p') | q = (p | q) + (p' | q)$ would impose the equivalence of terms disregarding the level where the non-deterministic choice has taken place.

In our specification there is no means to force an action to be performed instead of another in some context, if both choices are available for that component. This capability is usually achieved by *hiding* some action in an agent, so that it cannot be individually performed but is activated only by a parallel interaction with an agent capable of its complementary action. To axiomatize this construct we must be able to say if two actions are equal or not, in order to allow all actions but the restricted one. Notice that the use of equality is

not sufficient, because we want to express properties with inequalities in the premises like $a \neq b \wedge p \xrightarrow{a} p' \Rightarrow p|_b \xrightarrow{a} p'_b$, saying that if p has the capability of making an action a and a is not the action b we want to hide, then the restriction of p can perform a as well. This is a limitation of conditional axioms: whenever the negation of a property is needed in the premises of an axiom, it must be introduced as a new symbol and axiomatized. Equivalently, the property (in this case the equality) must be expressed not as a predicate, but as a Boolean function, introducing Boolean sort and operations as well.

Thus, let us assume that the specification of actions is actually richer than the first proposed and includes a predicate `different` : $\text{Act} \times \text{Act}$. Then we can enrich the agent specification

```
spec CCS = enrich CCS_ by
  opns   _|- : Agents × Act → Agents
  vars   a, b : Act; p, p' : Agents
  axioms p  $\xrightarrow{a}$  p' ∧ different(a, b) ⇒ p|_b  $\xrightarrow{a}$  p'_b
```

Negation of equality (and more in general of atomic sentences) is not the only kind of logical expression that we may want to express but that is not allowed by the conditional framework.

For instance, let us suppose that we want to define a predicate describing that an agent is allowed to perform, if any, just one action. Then we basically would like to give the following specification.

```
spec CCS1 = enrich CCS by
  preds  _must do_ : Agents × Act
  vars   a, b : Act; p, p' : Agents
  axioms p must do a ⇔ (∀ p' : Agents. ∀ b : Act. p  $\xrightarrow{b}$  p' ⇒ a = b)
```

But this is not, nor can be reduced to a conditional specification. Indeed, it has no initial model, because the minimality of the transition predicate conflicts with the minimality of the predicate `must do`.

The lack of an initial model, as well as the need for a more powerful logic, is quite common whenever functions and predicates are described through their properties instead than by an algorithm computing them. This situation is unavoidable in the phase of the *requirement* specification of a data type, when the implementation details are still left underspecified. (The fixing of details is usually done in the *design* phase.) Indeed, for instance the given description of the predicate `must do` does not depend on the structure of the agents, that could still be changed leaving the specification unaffected, nor suggests a way to compute/verify if it holds on given agent and action. But, exploiting the information we have on the definition of the transition predicate, we can as well specify the same predicate using conditional axioms:

```
spec CCS2 = enrich CCS by
  preds  _must do_ : Agents × Act
  vars   a : Act; p, q : Agents
  axioms a.p must do a
         p must do a ∧ q must do a ⇒ p||q must do a
         p must do a ∧ q must do a ⇒ p + q must do a
```

The latter specification has the expected initial model. However, it also has models that do not satisfy the specification CCS_1 (as some agent in them has more transitions than those strictly required by the specification CCS). Moreover, the definition of the predicate `must do` is correct only for this particular description of the agents, but should be updated if, for instance, a new combinator would be added for agents. Therefore, CCS_1 is much more flexible and can be used during the requirement phase, while CCS_2 can be adopted as a solution only for the design phase. ■

Let us see one more example of an (initial) specification of data types: the specification of finite maps. Since finite maps are the basis for the abstract description of stores and memories, this data type is, obviously, crucial for the description of each imperative data type.

Example 3.16. Let us assume given specifications of locations (Sp_L , with main sort `loc`) and values (Sp_V , with main sort `value`) for a given type of our imperative language. We want to define the specification of the *store* data type. Since we want to update a store introducing a new value at a given location, we need the capability of looking whether two locations are equal or not, as in Example 3.15. Therefore, we assume that a Boolean function representing equality is implemented in our specification of locations.

```
spec Sp_L =
  sorts   loc, bool ...
  opns   T, F: → bool
         eq: loc × loc → bool ...
```

Notice that the axioms describing *eq* cannot be given, without further assumption on the form of locations. Indeed, it is quite easy to guarantee that *eq* yields *T* over equal elements, by requiring the axioms $x = y \Rightarrow eq(x, y) = T$ and $eq(x, y) = T \Rightarrow x = y$. But, in order to get that *eq* yields *F* over distinct elements, we should add an axiom of the form $(\neg x = y) \Rightarrow eq(x, y) = F$, which is not positive conditional. Another possibility is to add the axiom³ $\forall b : \text{bool}. b = T \vee b = F$, that, in connection with the sentences axiomatizing the truth of *eq*, suffices to identify to *F* the result of *eq* over distinct elements, but this axiom is not conditional, either. Moreover, in either case we need to state that *T* and *F* represent distinct elements of sort `bool`, that is, we need an axiom of the form $\neg T = F$, that is not conditional, either. There is no general way of fully axiomatizing equality using conditional axioms, though it is possible to do it in many particular cases taking advantage of the structure of the terms of the argument sort.

Using *eq*, we can impose the extensional equality on stores, requiring that the order of updates of different locations is immaterial and that only the last update for each variable is recorded.

³ Adding the axiom is equivalent to restricting the model class to those models having (at most) a two valued boolean carrier set and developing the theory of conditional specifications for those model classes.

```

spec Stores1 = enrich SpL, SpV by
  sorts   store
  opns   empty: → store
           update: store × loc × value → store
  vars   x, y : loc; v1, v2 : value; s : store
  axioms update(update(s, x, v1), x, v2) = update(s, x, v2)
           eq(x, y) = F ⇒ update(update(s, x, v1), y, v2) =
           update(update(s, y, v2), x, v1)

```

The initial model of Stores₁ has the intended stores as elements of sort **store**, but no tools to retrieve the stored values. In order to introduce an operation **retrieve**: **store** × **loc** → **value**, we should first fix our mind about the result of retrieving a value from a location that has not been initialized in that store. Indeed if we simply give the specification

```

spec Stores2 = enrich Stores1 by
  opns   retrieve: store × loc → value
  vars   x : loc; v : value; s : store
  axioms retrieve(update(s, x, v), x) = v

```

then in its initial model the application **retrieve**(*s*, *x*) to stores *s* where *x* has never been updated, for instance if *s* = **empty**, cannot be reduced to a primitive value, but is a new element of sort **value**. This is a patent violation of any elementary principle of modularity and unfortunately does not depend on the initial approach or the particular specification.

Indeed, in all (total) models of Stores₂, a value for (the interpretation of) **retrieve**(**empty**, *x*) must be supplied, that logically should represent an error. Hence if Sp_V defines only “correct” values, either a new “error” value is introduced, violating the modularity principle, or an arbitrary correct value is given as result of **retrieve**(**empty**, *x*), against the logic of the problem. A (quite unsatisfactory) solution is requiring that all sorts of all specifications provide an “error”, so that when modularly defining a function on already specified data types, if that function is incorrect on some input, the result can be assigned to the “error” element. But this solution has two main limitations. Indeed, possibly simple and perfectly correct specifications have to be made far more complex. This is caused by the introduction of error elements that can appear as argument of the specification operators, requiring axioms for error propagation and messing up with the axioms for “correct” values. Moreover, if the errors are provided by the basic specifications, then they are classified depending on the needs of the original specifications; hence they do not convey any distinction among different errors due to the newly introduced operators. Therefore, the different origins of errors get confused and it is a complex task to define a sensible system of “error messages” for the user.

The point is that stores are inherently *partial* functions and hence the specification of their application should be partial as well. In the following sections we will see how, relaxing the definition of Σ-structure by allowing

the interpretation of some function symbols to be partial, the specification of most partial data types is simplified. ■

Bibliographical notes. After proposing the equational specification of abstract data types [GTW78] (see also [EM85]), the famous ADJ-group soon recognized the need for conditional axioms [TWW81]. Conditional axioms with predicates (but without full equality) are also used in logic programming and Prolog [Llo87]. A combination of both points of view, that is conditional axioms with equations *and* predicates, is done in the Eqlog language [GM86b]. The proof theory of this is studied in [Pad88].

3.2 Partial data types

The need for a systematic treatment of partial operations is clear from practice. One must be able to handle *errors* and *exceptions*, and account for *non-terminating operations*. There are several approaches to deal with these in literature, none of which appears to be fully satisfactory. *S. Feferman* [Fef92]

3.2.1 Different motivations for partiality

Partial operations, besides being a useful tool to represent not yet completely specified functions during the design refinement process, are needed to represent partial recursive functions. In the practice, partiality arises from situations that can be roughly partitioned into the following categories:

- a usual total abstract data type, like the positive natural numbers, is enriched by a partial function, like the subtraction. A canonical case is the axiomatic introduction of the inverse of some constructor. Most of the examples take place in this category, like the famous case of the *stacks*, where the stacks are built by the total functions *empty* and *push*. Then, *pop* and *top* are defined on them (i.e. the result of the application of these operations is either an “error” or a term built from the primitive operations);
- the partial functions that have to be specified are the “constructors” of their image set; consider for example the definition of *lists* without repetitions of elements, or of *search trees*; in both cases the new data type is built by partial inserting operations. This is not uncommon especially for hardware design or at a late stage of projects, when *limited* or *bounded* data types have to be defined, as, for instance, *integer* subranges (where *successor* and *predecessor* operations are the constructors; they result in an error when applied to the bounds of the subrange), or bounded *stacks* (where *pushing* an element on a full *stack* yields an error).

- a partial recursive function with non-recursive domain, for example an interpreter of a programming language, has to be specified;
- a semidecidable predicate p has to be specified, like in concurrency theory the *transition* relation on processes, or the *typing* relation for higher-order languages. Thus, representing p as a Boolean function f_P , it is possible to (recursively) axiomatize the truth, but not whether f_P yields *false* on some inputs and hence f_P is partial (or its image is larger than the usual Boolean values set);

The last case has already been solved, by explicitly adding predicates to our signature, as in the last section. Thus, let us focus on the others.

Programming on data types. Consider the following situation: we have defined a data type by a minimal set of (total) functions providing a way of construction for each element of our data type, that hence are called *constructors*. Then we want to enrich it by some (possibly partial) functions that are *programmed* in terms of the constructors, in the sense that their application to primitive elements either reduces to a term built by the constructors too, or is an error.

A particular case is the specification of the constructor inverses, from now on called *selectors*. Consider, for instance, as running example, the (Peano's style) specification of the natural numbers, by *zero* and *successor*.

```
spec Nat =
  sorts   nat
  opns    zero: → nat
          succ: nat → nat
```

Suppose that now we want to define the inverse of `succ`, that is the *predecessor* `prec`. Then the unique problem comes from the application of `prec` to elements that are not in the `succ` image, that is to `zero`.

Within a total approach, we have the following possibilities to cope with this. Either we introduce a new sort, representing the domain of `prec`, that does not contain `zero`, so that `prec(zero)` is not a well-formed term any more. Or we let `prec(zero)` denote an *erroneous* element `err`, and hence an error management mechanism has to be provided as well, for instance identifying all the application of the operations of the data type to the error(s) with `err`.

Partial constructors. A quite different problem from that introduced in the last subsection is the definition of data types whose constructors themselves are partial functions.

A paramount example of this case can be found in the formal languages field. Indeed, each production of a context-free grammar, with the form $s ::= w_1 s_1 \dots s_n w_{n+1}$, where the possibly decorated s 's are non-terminal symbols and each w_i is a string of terminal symbols, corresponds to a total function

from $s_1 \times \dots \times s_n$ into s . Thus, context-free grammars can be represented by total signatures. But attributed grammars cannot, because the applicability of production rules, i.e. of constructors, may be partial. The same applies also to grammars for languages whose operators are assigned a priority. Indeed, for instance in the case of plus and times on integers, to have that a string $x + y \times z$ unambiguously represents $x + (y \times z)$, the rule for times cannot apply to terms having some plus in the outmost position. Therefore, the interpretation on the times operator must be partial.

Other very relevant examples may be found, for instance, during the implementation phase, where, due to the machine limits, data types are limited. Indeed, in these cases, the constructors of infinite data type sets, like *successor* and *predecessor* for integers, or *push* for stacks, are not defined on the extreme values and become, hence, partial.

Partial recursive functions with non-recursive domains. Examples for this typically in the specification of system programs. When specifying an interpreter of a programming language that is expressive enough to describe all Turing computable functions, we get a partial recursive function. Indeed, the interpreter is recursive, but it loops for ever on some inputs. Since the Halting Problem is undecidable, it is undecidable whether the interpreter eventually yields an output or not. Thus, the interpreter is a partial recursive function with non-recursive domain.

Another example is an automatic theorem proving system that, given a proof goal, searches for a proof tree proving that goal. Now for a Turing complete logic, such as first-order logic, provability is undecidable in general. So our automatic theorem proving system will be a partial recursive function with non-recursive domain.

3.2.2 Capturing partiality within total frameworks

We now discuss how good the above listed occurrences of partiality can be treated within a total framework.

Static elimination of errors with order-sorted algebra. Let us consider again the example of natural numbers. Basically we want to enrich \mathbf{Nat} by a new sort \mathbf{pos} (representing the domain of \mathbf{prec}) and the function \mathbf{prec} itself. But we also have the intuition that the domain of \mathbf{prec} is a subset of \mathbf{nat} . In particular, we would like to be able to apply \mathbf{prec} to all strictly positive elements of sort \mathbf{nat} . Thus, in a pure many-sorted style we should also add the embedding of \mathbf{pos} into \mathbf{Nat} , producing the following specification.

```
spec Natpos =
  sorts   nat, pos
  opns   zero: → nat
         succ: nat → pos
```

```

prec: pos → nat
e: pos → nat
axioms  $\forall x : \mathbf{nat}. \mathbf{prec}(\mathbf{succ}(x)) = x$ 
          $\forall x, y : \mathbf{pos}. e(x) = e(y) \Rightarrow x = y$ 

```

Thus, for instance, the expression $0 + 1 + 1 - 1$ is represented by the term $\mathbf{prec}(\mathbf{succ}(e(\mathbf{succ}(\mathbf{succ}(\mathbf{zero}))))$, while we would expect $\mathbf{prec}(\mathbf{succ}(\mathbf{succ}(\mathbf{zero})))$. Therefore, it is much preferable to enrich the theory by explicitly allowing the *subsorts*, that is having sorts that must be interpreted in each model as subsets of the interpretation of other sorts. Thus, for instance, the size of each model it is not unduly enlarged by a subsort declaration. This can be relevant in the implementation phase. Accordingly, the rules for term formation are relaxed, so that any function requiring an argument of the supersort can also accept an argument of the subsort.

The resulting theory of *order-sorted* algebras, that will be more extensively used in the next subsection, is much deep and complex. We refer to [GM87,GM92,MG93,Yan93,Mos93,Poi90] for a formal presentation of the order-sorted approach. Here, we informally use the following specification

```

spec Natosa =
  sorts   pos ≤ nat
  opns   zero: → nat
          succ: nat → pos
          prec: pos → nat
  axioms  $\forall x : \mathbf{nat}. \mathbf{prec}(\mathbf{succ}(x)) = x$ 

```

with the convention that if t is a term of sort **pos** then it is a term of sort **nat** as well. In other words the above specification is a more convenient presentation of $\mathbf{Nat}^{\mathbf{pos}}$ but it has an equivalent semantics.

Then two main problems can be seen. First of all, as the domain of **prec** is *statically* described by means of the signature, it cannot capture our intuitive identifications of terms as different representations of the same value, that depend on the deductive mechanism of the specification and hence are, so to speak, *dynamic*. Thus, the term $\mathbf{prec}(\mathbf{prec}(\mathbf{succ}(\mathbf{succ}(\mathbf{zero}))))$ is incorrect, because the result type of **prec** is **nat**, while **prec** expects an argument of sort **pos**, even if $\mathbf{prec}(\mathbf{succ}(\mathbf{succ}(\mathbf{zero})))$ can be deduced equal to $\mathbf{succ}(\mathbf{zero})$. Hence, intuitively it should have sort **pos**. This problem cannot be avoided by any approach based on a static elimination of error elements, that is on a refinement of the typing of functions at the signature definition level.

Consider now the problem of specifying a partial constructor, say, the **push** operation for bounded stacks. This specification is, of course, parametric on the definition of the element data type, that we assume given by the specification **Elem**, with a sort **elem**, and is based also on a specification of natural numbers with an order relation \leq on them, in order to define the depth of a stack.

```

spec Nat≤ =
  sorts   nat

```

```

opns  zero:  $\rightarrow$  nat
        succ: nat  $\rightarrow$  nat
preds  $- \leq \_ :$  nat  $\times$  nat ...
vars   $x, y :$  nat
axioms zero  $\leq$  x
         $x \leq y \Rightarrow$  succ( $x$ )  $\leq$  succ( $y$ )

```

The problem is that only a fixed number of `push` of iterations should be allowed. Of course it is always possible, although awkward, to introduce ad hoc constructors, for instance $n - 1$ constructors each one representing the creation of a stack with exactly i elements for $i = 1 \dots n - 1$, where n is the maximal allowed depth of a stack. This approach would be not only unnatural, but also non-parametric w.r.t. the maximum n ; indeed if the value n would be changed in a further stage of design, functions should be added to the specification as well as axioms.

A more promising approach is to specify the domain of the partial constructor `push`:

```

spec BoundedStacks = enrich Nat $_{\leq}$ , Elem by
sorts  nonfullbstack  $\leq$  bstack
opns   max:  $\rightarrow$  nat
opns   empty:  $\rightarrow$  bstack
        push: elem  $\times$  nonfullbstack  $\rightarrow$  bstack
        depth: bstack  $\rightarrow$  nat
axioms max = ...
        depth(empty) = zero
         $\forall x : \text{elem}, s : \text{nonfullbstack. depth}(\text{push}(x, s)) = \text{succ}(\text{depth}(s))$ 

```

But now we have the problem similar to the problem with nested application of `succ` and `prec`: A term like

$$\text{push}(x, \text{push}(y, \text{empty}))$$

is not well-formed, because `push` expects a `nonfullbstack`, but only delivers a `bstack`.

The situation becomes even worse when considering partial recursive functions with non-recursive domains: the specification of a non-recursive subsort seems to be difficult, if not impossible, in the order-sorted approach.

Dynamic treatment of errors: retracts. In OBJ3 [GMW⁺93], the problem of ill-formedness of `prec(prec(succ(succ(zero))))` is solved by automatically adding retracts $r : \text{nat} > \text{pos}$ which can be removed using retract equations

$$\forall s : \text{pos. } r : \text{nat} > \text{pos}(s) = s$$

and which are irreducible in case of ill-typed terms. Then in a term like `prec(prec(succ(succ(zero))))`, a retract is added:

$$\text{prec}(r : \text{nat} > \text{pos}(\text{prec}(\text{succ}(\text{succ}(\text{zero}))))))$$

which, by the retract equation, has the intended semantics. Now terms like `prec(zero)` are parsed as an irreducible term:

$$\text{prec}(r : \text{nat} > \text{pos}(\text{zero}))$$

which can be seen as an error message. In a similar vein, by adding a conditional retract equation

$$\forall s : \text{bstack.succ}(\text{depth}(s)) \leq \text{max} \Rightarrow r : \text{bstack} > \text{nonfullbstack}(s) = s$$

multiple pushes on a bounded stack are allowed, provided that they do not exceed the bound. Otherwise, we get an irreducible term serving as an error message.

Discussion. The main problem with the retract approach is the following: terms involving irreducible retracts introduce useful new error elements, but thereby change the semantics of the specification. In [MG93] it is shown that specification with retracts have an initial semantics given by an injective homomorphism from the initial algebra of the specification without retracts to the initial algebra of the specification with retracts⁴. This hiatus between the signature of the models, and in particular of the initial one, that has no retracts in it, and the signature of the terms used in the language can be eliminated, if retracts are allowed to be truly *partial* functions. This solution needs a framework where order-sorted algebra is combined with partiality, as for instance in the language CASL in course of definition within the CoFI initiative (see Section 3.3.5).

Dynamic treatment of errors: sort constraints. An alternative approach uses (conditional) sort constraints [MG93,Yan93]. A sort constraint expresses that some term, which syntactically belongs to some sort s , is always interpreted in such a way that it already belongs to a sort $s' \leq s$ ⁵.

Now let us add a sort constraint⁶

$$\forall n.\text{nat}.\text{prec}(\text{succ}(\text{succ}(n))) : \text{pos}$$

to the above specification.

If now well-typedness of terms is defined by also taking sort constraints into account [GMJ85], the term

$$\text{prec}(\text{prec}(\text{succ}(\text{succ}(\text{zero}))))$$

⁴ Likewise, there is a free construction embedding any algebra of a specification without retracts into an algebra of the same specification with retracts.

⁵ Actually, s and s' only need to belong to the same connected component.

⁶ In CASL there are *membership* axioms that can be used to express sort constraints, but that do not have influence on parsing.

is well-typed because $\text{prec}(\text{succ}(\text{succ}(n)))$ is of sort `pos` by the sort constraint.

Similarly, by adding a conditional sort constraint

$$\forall s : \text{bstack}.\text{succ}(\text{depth}(s)) \leq \text{max} \Leftrightarrow s : \text{nonfullbstack}$$

we again get the possibility of multiple pushes on a bounded stack (provided that they do not exceed the bound).

If the partial function to be introduced is not the inverse of one of the constructors, then the domain of the partial function has to be introduced and axiomatized explicitly. This can be done with sort constraints, which allow to specify subsorts to consist of all those values that satisfy a given predicate. As an artificial, but simple, example let us consider the division by 2, that is well defined only for even numbers, corresponding to the following specification in an order-sorted simplified notation.

```
spec Nat2 =
  sorts   even ≤ nat
  opns   _ mod 2 : nat → nat
         _ div 2 : even → nat
  axioms zero mod 2 = zero
         succ(zero) mod 2 = succ(zero)
         ∀ x : nat. succ(succ(x)) mod 2 = x mod 2
         ∀ x : nat. x : even ⇔ x mod 2 = zero
         zero div 2 = zero
         ∀ x : even. succ(succ(x)) div 2 = succ(x div 2)
```

Note that the equivalence defining `even` can be expressed with a conditional sort constraint

$$\forall x : \text{nat}. x \text{ mod } 2 = \text{zero} \Rightarrow x : \text{even}$$

together with an ordinary axiom

$$\forall x : \text{even}. x \text{ mod } 2 = \text{zero}$$

With this method we can even specify non-recursive domains of partial recursive functions.

If the partial function to be introduced is not unary, the order-sorted approach does not immediately apply, because the domain should be a subsort of a *product* sort, but usually the sort set does not include products. One then has to define products explicitly by a tupling operation together with projections, and specify the domain of the operation to be a subsort of the product, using sort constraints.

Discussion. Allowing sort constraints to have influence on well-formedness, as proposed in [GMJ85], means that type checking can generate proof obligations, which in general can be resolved only dynamically by doing some

theorem proving.⁷ So parsing of terms becomes quite complex. Some work in this direction has been done in [Yan93], but there is no fully worked out theory yet.

The contribution of order-sorted algebra with sort constraints is to allow a separation of those parts of type-checking which can be done statically from those which can be done only dynamically. This distinction is lost in the approach of partial algebras introduced below. Therefore, it is worthwhile to combine the order-sorted and the partial approach, as it is done in the CASL language (see Section 3.3.5).

Error elements. Another possibility to deal with `prec(zero)` is using this term as a denotation for error. This is clearly inconsistent with a modular approach, as one or more new element(s) interpreting the error(s) have to be added to the models of the original specification (see [Poi87] for an argumentation against the phenomenon that error elements in basic types are introduced by hierarchically building more complex specifications). Moreover, having added (at least) one new element, the application of the data type functions to it has to be specified as well. In the pioneering *Error algebras*, introduced by the ADJ group in [GTW78], to achieve a reasonable uniformity, one constant symbol for each sort is added and all the errors have to reduce to it by introduction and propagation axioms. Of course the naive application of error propagation can produce a lot of troubles. Indeed, consider, for instance, the definition of natural numbers with product. Then, by instantiating the error propagation axiom for the product, $\mathbf{err} * x = \mathbf{err}$, with `zero` and the standard basis of the inductive product definition, $x * \mathbf{zero} = \mathbf{zero}$, with `err`, we deduce $\mathbf{zero} = \mathbf{err}$.

Thus, in [GTW78] a uniform technique to avoid these inconsistencies is introduced, consisting basically of a distinction of axioms into *axioms for correct elements* and *error propagation axioms*. Since error algebras are described as equational specifications of many-sorted algebras, the resulting specifications are quite heavy. However, by using predicates and conditional formulae, their usage is improved, because the implementation of the Boolean type, with its connectives, is not needed anymore. Still, for each function of n arguments, n error propagation axioms have to be stated, each constructor requires a correctness propagation axiom and each error introduction must be detected by an appropriate axiom. Thus, specifications in this style cannot be concise. Moreover, each axiom stating properties on the correct elements, that is the proper axioms of the data type, must be *guarded* by the predicates stating the correctness of their input in the premises.

⁷ But we cannot expect all definedness conditions to be resolved statically, because definedness is undecidable in general. And indeed, within usual approaches to partial algebras, terms may not denote and their definedness can be checked only dynamically with theorem proving.

For instance one of the more classical examples, that is the specification of natural numbers with sum and product, using a predicate to state that a term is correct, would be the following.

```

spec Nat* =
  sorts   nat
  opns    zero, err: → nat
          succ: nat → nat
          plus, minus, times: nat × nat → nat
  preds  OK, IsErr: nat
  vars   x, y: nat
  axioms OK(zero)
          OK(x) ⇒ OK(succ(x))
          IsErr(x) ⇒ IsErr(succ(x))
          IsErr(err)
          OK(x) ⇒ plus(x, zero) = x
          OK(x) ∧ OK(y) ⇒ plus(x, succ(y)) = succ(plus(x, y))
          IsErr(x) ⇒ IsErr(plus(x, y))
          IsErr(x) ⇒ IsErr(plus(y, x))
          OK(x) ⇒ times(x, zero) = zero
          OK(x) ∧ OK(y) ⇒ times(x, succ(y)) = plus(x, times(x, y))
          IsErr(x) ⇒ IsErr(times(x, y))
          IsErr(x) ⇒ IsErr(times(y, x))
          OK(x) ⇒ minus(x, zero) = x
          OK(x) ∧ OK(y) ⇒ minus(succ(x), succ(y)) = minus(x, y)
          IsErr(minus(zero, succ(x)))
          IsErr(x) ⇒ IsErr(minus(x, y))
          IsErr(x) ⇒ IsErr(minus(y, x))
          IsErr(x) ⇒ x = err

```

Notice that removing the last axiom, the incorrect terms are all distinct and can serve, then, as very informative error messages.

The error algebra approach can also be used to specify partial constructors. In the bounded stack example, pushing too many elements on the same stack results in one *error* element:

```

spec BoundedStacks = enrich Nat≤, Elem by
  sorts   bstack
  opns    max: → nat
          empty, err: → bstack
          Bpush: elem × bstack → bstack
          depth: bstack → nat
  preds  OK, Is_Err: bstack
  vars   x: elem; s: bstack
  axioms max = ...
          OK(empty)
          depth(empty) = zero
          depth(err) = succ(max)
          succ(depth(s)) ≤ max ⇒ OK(Bpush(x, s))
          OK(Bpush(x, s)) ⇒ depth(Bpush(x, s)) = succ(depth(s))

```

$$\begin{aligned} \max \leq \text{depth}(s) &\Rightarrow \text{Is_Err}(\text{Bpush}(x, s)) \\ \text{Is_Err}(s) &\Rightarrow s = \text{err} \end{aligned}$$

The first axiom fixes the actual maximal size of the bounded stacks, while the last one identify all errors.

The specification of partial recursive functions with non-recursive domain is possible within the error algebra approach, but in semi-computable models it inevitably leads to infinitely many error values.⁸

Discussion. Many other approaches flourished from the original error algebras, refining the basic idea of cataloging the elements of the data type but using more powerful algebraic framework to express the specifications. See e.g. the *exception algebras* in [BBC86], where both the elements of algebras and the terms are labeled to capture the difference between errors and exceptions, or the *clean algebras* in [Gog87], where an order-sorted approach is adopted to catalogue the elements of algebras. Among the approaches that use error elements in some way to model partiality within a total framework, there are Equational Type Logic [MSS90,MSS92], unified algebras [Mos89] and based algebras [Kre87,KM95]. In spite of the potentiating and the embellishments, these approaches share with the original one the difficulties of interaction with the modular definition of data types. Indeed the *non-ok* elements of basic types have to be designed *a priori* to support error messages, or exceptions caused by other modules that use the basic ones.

Thus, error algebras (and variations on the theme) are more suitable for specifying a completely defined system than for refining a project or represent (parts of) a library of specifications *on the shelf*. Quite often in the last stages of the refinement process even functions that are partial from a philosophical point of view (for instance the constructors of bounded stacks, bounded integer, search trees, ordered lists and other bounded resources or finite domains) are implemented as total functions, identifying incorrect applications with error messages, that are special algebra elements, or using more sophisticated approaches, where there is a notion of state and error handling means that a special state is entered (see Chapter 14, in this book). However, we cannot delay the semantics of the data until every detail has been decided, because of methodological reasons: we want to separate *requirement specifications* that are close to the informal understanding of human beings, from *design specifications* that are close to implementations.

Therefore, we have to find a way of dealing with the requirement specification of partial functions and in particular of partial constructors. In the above example we have seen the *design* specification of bounded stacks, that cannot be further refined. For instance all errors have been identified and

⁸ If there were only finitely many errors, one could use them to recursively enumerate the complement of the domain. The domain itself can be recursively enumerated as well. Both recursive enumerations lead to a decision procedure for the domain, which contradicts the assumption that the domain is non-recursive.

cannot be anymore distinguished in order to get a more informative error message system. In a total approach it is impossible (or, better, unnatural and inconvenient) to give the *requirement* specification of bounded stacks. Therefore, in the next section we will introduce a more powerful framework, based on the (possibly) *partial* interpretation of function symbols and see how it can be used to easily describe this and other specifications.

3.3 Partial first-order logic

When developing a model theory for partial first-order structures, it is not obviously clear in which way to proceed, as there are possibilities for different choices at various points. Sometimes different choices have severe technical implications, sometimes they are more or less only a matter of taste. See [CJ91, Far91] for an overview over different approaches.

The introduction of symbols denoting partial functions has the effect that not all terms can be interpreted in each model (unless well-formedness of terms is made dependent on the model where they have to be interpreted in, which seems to be not very useful). Thus, the valuations of terms is inherently partial, but it is still the question whether such partiality should propagate to formulae built over terms.

We here follow the two-valued approach developed and motivated by Burmeister [Bur82] and others [Bee85, Far91, Far95, Fef95, Par95]. In a two-valued logic of partial functions, formulae which contain some non-denoting term are interpreted as either true or false. The main motivation for this principle is that for *specification* of software systems, we need definiteness, i.e. we want to know whether some property does hold or does not hold, and do not want to have a “may-be”. Moreover, the logic we will develop is a so-called *negative logic* [GL97], that is, atomic formulae containing some non-denoting term are interpreted as false.⁹ Later on, we briefly sketch a three-valued logic [CJ91, Jon90] where the valuations for formulae may be partial as well, that is, a formula containing some non-denoting term is neither interpreted as true nor as false, but some third truth-value is assigned to it.

3.3.1 Model theory

This section is devoted to the study of categories of partial first-order structures. Since many definitions and results can be stated in a uniform language, using category theory, and hold for many algebraic frameworks, we will try to clarify the basic nature of these categories, and discuss the existence of very simple constructions, to allow an experienced reader to apply the available theories to our framework. However, as the proposed constructions have a quite natural and intuitive counterpart, basically generalizing analogous

⁹ Strictly speaking, this holds only if one counts strong equations as non-atomic formulae. But strong equations are a derived notion in our formalism.

constructions in (indexed) set theory, even those who are not interested in categories and their applications, can find useful our theory, simply ignoring the categorical terminology.

Partial first-order structures differ from (total) structures in the interpretation of function symbols being possibly *partial* functions, i.e. they are not required to yield a result on each possible input. Thus, total functions are a particular case of partial functions, that happen to be defined on all the elements of their source. It is anyway convenient to discriminate as soon as possible between total and partial functions of a data type, because knowing a function (symbol) to be (interpreted as) total simplifies its treatment not only from an intellectual point of view designing the data type, but also, for example, applying rewriting techniques or proof deductions. Therefore, we distinguish already at the signature level between function symbols that must be interpreted as total and function symbols that are allowed to denote partial operations (but can, obviously, be total as well in some model).

Definition 3.17. A *partial signature* $\Sigma = \langle S, \Omega, P\Omega, \Pi \rangle$ consists of a set S , denoted by $sorts(\Sigma)$, of *sorts*, two componentwise disjoint $S^* \times S$ -sorted families Ω and $P\Omega$, respectively denoted by $opns(\Sigma)$ and $popns(\Sigma)$, of *total* and *partial function symbols* and an S^* -sorted family Π , denoted by $preds(\Sigma)$, of *predicate symbols*.

Given a partial signature $\Sigma = \langle S, \Omega, P\Omega, \Pi \rangle$ and an S -sorted family X of variables, the Σ -*terms* are the terms on the first-order signature $(S, \Omega \cup P\Omega, \Pi)$, as introduced in Definition 3.2. \diamond

Thus, terms on a partial signature are defined as usual, disregarding the distinction between total and partial functions. Therefore, the same symbol cannot be used for a total and a partial function with the same arity, as such an overloading would introduce a semantic ambiguity.

Example 3.18. Let us see a signature for non-negative integers, with partial operations, like predecessor, subtraction and division, and a predicate stating if a number is a multiple of another.

```
sig  $\Sigma_{\mathbb{N}at} =$ 
  sorts   nat
  opns    zero:  $\rightarrow$  nat
          succ: nat  $\rightarrow$  nat
          plus, times: nat  $\times$  nat  $\rightarrow$  nat
  popns   prec: nat  $\rightarrow$  nat
          minus, div, mod: nat  $\times$  nat  $\rightarrow$  nat
  preds   multiple: nat  $\times$  nat
```

Analogously, a signature for stacks with possibly partial interpretation of top and pop on an empty stack, based on a signature $\Sigma_{\mathbb{E}lem}$, describing the type $elem$ of the elements for the stack, is the following.

```
sig  $\Sigma_{\mathbb{S}tack} = \text{enrich } \Sigma_{\mathbb{E}lem} \text{ by}$ 
```

```

sorts   stack
opns   empty:  $\rightarrow$  stack
          push:  $\text{elem} \times \text{stack} \rightarrow \text{stack}$ 
popns  pop:  $\text{stack} \dashrightarrow \text{stack}$ 
          top:  $\text{stack} \dashrightarrow \text{elem}$ 
preds  is_in:  $\text{elem} \times \text{stack}$ 
          is_empty:  $\text{stack}$ 

```

■

Since function symbols are partitioned into total and partial ones and both families are classified depending on their input/output types, the same symbol can appear many times in the same signature, possibly making the term construction ambiguous. Here and in the sequel we assume that terms are not ambiguous, i.e. that the overloading of function symbols is not introducing troubles (or that, if the overloading *is* problematic, that a different, unambiguous notation for terms has been adopted, for instance substituting for each function symbol a pair consisting of its name and its type).

Notation 3.19. For each partial function $pf: X \dashrightarrow Y$, we will denote its *domain* by $\text{dom } pf$, that is the subset of X defined by

$$\text{dom } pf = \{x \mid x \in X \text{ and } pf(x) \in Y\}.$$

In the total case, signature morphisms map sorts to sorts, operation symbols to operation symbols and predicate symbols to predicate symbols. In the partial case, having partitioned the operation symbols in total and possibly partial ones, total operation symbols must be mapped to total operation symbols, in order to be able to define reduct functors on models. For partial operation symbols, we have two possibilities: they can be either required to be mapped to partial operation symbols or to the union of total and partial operation symbols. The latter choice is more adequate for using signature morphisms to represent refinement and has, hence, been adopted in CASL.

Exercise 3.20. Generalize the notion of signature morphism for partial signatures, both for the case that partial operation symbols are required to be mapped to partial operation symbols and for the case that partial operation symbols are allowed to be mapped to either partial or total operation symbols.

As the interpretations of function symbols in $P\Omega$ can be undefined on some input, not all (meta)expressions denote values in the carriers of a partial first-order structure. Thus, the meaning of an equality between expressions that can be non-denoting becomes ambiguous; indeed it is arbitrary to decide if an equality implicitly states the existence of the denoted element, or holds also if both sides are undefined (assuming the viewpoint that (non-existing) undefined elements are indistinguishable), or is satisfied whenever the two sides do not denote different elements. Therefore, in the sequel we will use

different equality symbols for the different concepts and in particular we will use $e \stackrel{\text{E}}{=} e'$ (*existential equality*) to state that both sides denote the same value, $e \stackrel{\text{W}}{=} e'$ (*weak equality*) to state that if both sides denote a value, then the two values coincide, and $e \stackrel{\text{S}}{=} e'$ (*strong equality*) to state that either both sides denote the same value or both do not denote any value. Thus, in particular $e \stackrel{\text{E}}{=} e$, is equivalent to e denoting a value, and hence is usually represented by $e \downarrow$ (and its negation becomes $e \uparrow$).

It is interesting to note that assuming as primitive either existential or strong equality, the other notions can be derived; indeed $e \downarrow$ for an expression e of sort s can be expressed simply as $e \stackrel{\text{E}}{=} e$ or as “there exists some x with $e \stackrel{\text{S}}{=} x$ ” with x of sort s non-free in e . Then, using $e \downarrow$ as syntactic sugar for the above mentioned formulae of our meta-language, we have the following table.

$$\begin{array}{l} \text{using: } \stackrel{\text{E}}{=} \text{ becomes } \quad \stackrel{\text{W}}{=} \text{ becomes } \quad \stackrel{\text{S}}{=} \text{ becomes} \\ e \stackrel{\text{E}}{=} e' \quad e \stackrel{\text{E}}{=} e' \quad (e \downarrow \wedge e' \downarrow) \supset e \stackrel{\text{E}}{=} e' \quad (e \downarrow \vee e' \downarrow) \supset e \stackrel{\text{E}}{=} e' \\ e \stackrel{\text{S}}{=} e' \quad e \downarrow \wedge e \stackrel{\text{S}}{=} e' \quad (e \downarrow \wedge e' \downarrow) \supset e \stackrel{\text{S}}{=} e' \quad e \stackrel{\text{S}}{=} e' \end{array}$$

On the other hand, weak equality is too weak, indeed, to describe the other kinds of equality, because in particular it is not possible to state the definedness of an expression using only weak equalities.¹⁰ But using weak equality and definedness assertions it is possible to represent both existential and strong equalities, as follows.

$$\begin{array}{l} \stackrel{\text{E}}{=} \text{ becomes } \quad \stackrel{\text{W}}{=} \text{ becomes } \quad \stackrel{\text{S}}{=} \text{ becomes} \\ e \downarrow \wedge e' \downarrow \wedge e \stackrel{\text{W}}{=} e' \quad e \stackrel{\text{W}}{=} e' \quad (e \downarrow \vee e' \downarrow) \supset (e \downarrow \wedge e' \downarrow \wedge e \stackrel{\text{W}}{=} e') \end{array}$$

Definition 3.21. Given a partial signature $\Sigma = \langle S, \Omega, P\Omega, \Pi \rangle$, a *partial Σ -structure* A is a triple consisting of

- an S -sorted family $|A|$ of *carriers*;
- a family $\{\mathcal{I}_A^{w,s} : \Omega_{w,s} \cup P\Omega_{w,s} \longrightarrow \text{PFun}_{w,s}\}_{(w,s) \in S^* \times S}$ of *function interpretations*, where $\text{PFun}_{s_1, \dots, s_n, s}$ is the set of all partial functions from $|A|_{s_1} \times \dots \times |A|_{s_n}$ into $|A|_s$, s.t. $\mathcal{I}_A^{w,s}(f)$ is total for each $f : s_1 \times \dots \times s_n \longrightarrow s \in \Omega$. In the sequel, if no ambiguity will arise, $\mathcal{I}_A^{w,s}(f)$ will be denoted by f_A for each $f \in \Omega_{w,s} \cup P\Omega_{w,s}$.
- a family $\{\mathcal{J}_A^{s_1, \dots, s_n} : \Pi_{s_1, \dots, s_n} \longrightarrow \wp(|A|_{s_1} \times \dots \times |A|_{s_n})\}_{s_1, \dots, s_n \in S^*}$ of *predicate interpretations*. In the sequel, if no ambiguity will arise, $\mathcal{J}_A^w(p)$ will be denoted by p_A for each $p : s_1 \times \dots \times s_n \in \Pi$.

¹⁰ For the object level (we introduce $\stackrel{\text{W}}{=}$ for weak equality in our formal logical language later on), this can be seen as follows. In any trivial first-order structure with singleton carriers, all weak equalities are true, disregarding the definedness of the involved expressions.

Moreover, given partial Σ -structures A and B , a *homomorphism* of partial Σ -structures from A into B is an S -sorted family h of (total) functions $h_s: |A|_s \longrightarrow |B|_s$ s.t.

- $h_s(f_A(a_1, \dots, a_n)) \stackrel{E}{=} f_B(h_{s_1}(a_1), \dots, h_{s_n}(a_n))$ for all $f: s_1 \times \dots \times s_n \longrightarrow s \in \Omega$ and all $a_i \in |A|_{s_i}$ for $i = 1, \dots, n$;
- if $pf_A(a_1, \dots, a_n) \downarrow$ then $h_s(pf_A(a_1, \dots, a_n)) \stackrel{E}{=} pf_B(h_{s_1}(a_1), \dots, h_{s_n}(a_n))$ for all $pf: s_1 \times \dots \times s_n \dashrightarrow s \in P\Omega$ and all $a_i \in |A|_{s_i}$ for $i = 1, \dots, n$;
- if $(a_1, \dots, a_n) \in p_A$, then $(h_{s_1}(a_1), \dots, h_{s_n}(a_n)) \in p_B$ for all $p: s_1 \times \dots \times s_n \in \Pi$ and all $a_i \in |A|_{s_i}$ for $i = 1, \dots, n$.

The category $Mod(\Sigma)$ has partial Σ -structures as objects and homomorphisms of partial Σ -structures as arrows, with the obvious composition and identities. \diamond

Therefore, in order to define a partial Σ -structure, we must provide:

- the S -sorted family $|A|$ of carriers;
- for each $f: s_1 \times \dots \times s_n \longrightarrow s \in \Omega$, the *interpretation* of f in A , that is a total function $f_A: |A|_{s_1} \times \dots \times |A|_{s_n} \longrightarrow |A|_s$;
- for each $pf: s_1 \times \dots \times s_n \dashrightarrow s \in P\Omega$, the *interpretation* of pf in A , that is a partial function $pf_A: |A|_{s_1} \times \dots \times |A|_{s_n} \dashrightarrow |A|_s$;
- for each $p: s_1 \times \dots \times s_n \in \Pi$, a subset p_A of $|A|_{s_1} \times \dots \times |A|_{s_n}$, representing the *truth values* of p in A .

Notice that, although the interpretation function for partial and total function symbols is one, it is often convenient to distinguish between partial and total function symbols, as in the above definition of homomorphism, where the definedness condition can be dropped for the total symbols.

Let us see a couple of examples of partial Σ -structures on the signatures introduced by the previous Example 3.18.

Example 3.22. The natural numbers with the “obvious” interpretation of function symbols is a partial Σ_{Nat} -structure.

algebra $\mathbf{N} =$

Carriers

$$|\mathbf{N}|_{\text{nat}} = \mathbf{N}$$

Functions

$$\text{zero}_{\mathbf{N}} = 0$$

$$\text{succ}_{\mathbf{N}}(n) = n + 1$$

$$\text{plus}_{\mathbf{N}}(n, m) = n + m$$

$$\text{times}_{\mathbf{N}}(n, m) = n * m$$

$$\text{prec}_{\mathbf{N}}(n) = \begin{cases} n - 1, & \text{if } n > 0 \\ \text{undefined}, & \text{otherwise} \end{cases}$$

$$\text{minus}_{\mathbf{N}}(n, m) = \begin{cases} n - m, & \text{if } n \geq m \\ \text{undefined}, & \text{otherwise} \end{cases}$$

$$\text{div}_{\mathbf{N}}(n, m) = \begin{cases} n \text{ div } m, & \text{if } m \neq 0 \\ \text{undefined}, & \text{otherwise} \end{cases}$$

$$\text{mod}_{\mathbf{N}}(n, m) = \begin{cases} n \bmod m, & \text{if } m \neq 0 \\ \text{undefined}, & \text{otherwise} \end{cases}$$

Predicates

$$\text{multiple}_{\mathbf{N}} = \{(n, m) \mid n \bmod m \stackrel{\text{E}}{=} 0\} \cup \{(0, 0)\}$$

Another first-order structure on the same signature is, for instance, the following, where “error messages” have been added to the carrier.

algebra $\mathbf{N}_E =$

Carriers

$$|\mathbf{N}_E|_{\text{nat}} = \mathbf{N} \cup E \quad \text{for } E = \{\text{underflow}, \text{err_minus}, \text{division_by_0}\}$$

Functions

$$\text{zero}_{\mathbf{N}_E} = 0$$

$$\text{succ}_{\mathbf{N}_E}(n) = \begin{cases} n + 1, & \text{if } n \in \mathbf{N} \\ n, & \text{otherwise} \end{cases}$$

$$\text{plus}_{\mathbf{N}_E}(n, m) = \begin{cases} n + m, & \text{if } n, m \in \mathbf{N} \\ n, & \text{if } n \in E \\ m, & \text{otherwise} \end{cases}$$

$$\text{times}_{\mathbf{N}_E}(n, m) = \begin{cases} n * m, & \text{if } n, m \in \mathbf{N} \\ n, & \text{if } n \in E \\ m, & \text{otherwise} \end{cases}$$

$$\text{prec}_{\mathbf{N}_E}(n) = \begin{cases} n - 1, & \text{if } n \in \mathbf{N}, n > 0 \\ \text{underflow}, & \text{if } n = 0 \\ n, & \text{otherwise} \end{cases}$$

$$\text{minus}_{\mathbf{N}_E}(n, m) = \begin{cases} n - m, & \text{if } n, m \in \mathbf{N}, n \geq m \\ n, & \text{if } n \in E \\ m, & \text{if } n \in \mathbf{N}, m \in E \\ \text{err_minus}, & \text{otherwise} \end{cases}$$

$$\text{div}_{\mathbf{N}_E}(n, m) = \begin{cases} n \text{ div } m, & \text{if } n, m \in \mathbf{N}, m \neq 0 \\ n, & \text{if } n \in E \\ m, & \text{if } n \in \mathbf{N}, m \in E \\ \text{division_by_0}, & \text{otherwise} \end{cases}$$

$$\text{mod}_{\mathbf{N}_E}(n, m) = \begin{cases} n \bmod m, & \text{if } n, m \in \mathbf{N}, m \neq 0 \\ n, & \text{if } n \in E \\ m, & \text{if } n \in \mathbf{N}, m \in E \\ \text{division_by_0}, & \text{otherwise} \end{cases}$$

Predicates

$$\text{multiple}_{\mathbf{N}_E} = \{(n, m) \mid n, m \in \mathbf{N} \text{ and } n \bmod m \stackrel{\text{E}}{=} 0\} \cup \{(0, 0)\}$$

It is immediate to see that the embedding of the Σ_{Nat} -structure \mathbf{N} into \mathbf{N}_E is a homomorphism. \blacksquare

Exercise 3.23. Following the guideline of the previous example, define several Σ_{Stack} -algebras and relate them by homomorphisms, when possible.

Homomorphisms of partial first-order structures are truth preserving *weak* homomorphisms using the notation of Definition 2.7.28. There are several other possible notions of homomorphism, basically due to the combinations of choices for the treatment of predicates (truth-preserving, truth-reflecting or

both) and partial functions (the condition $pf_A(a_1, \dots, a_n) \downarrow$ can be dropped or substituted by $pf_B(h_{s_1}(a_1), \dots, h_{s_n}(a_n)) \downarrow$), that are used in literature (see e.g. [Bur82, Rei87]). The definition adopted here guarantees that initial (free) models (if any) in a class are *minimal*, following the *no-junk & no-confusion* principle from [MG85].

Exercise 3.24. Generalize the notion of reduct for partial first-order structures for both kinds of signature morphisms as defined in Exercise 3.20.

In each algebraic formalism, like group or ring theories or total algebras, there is a notion of subobject that is (up to isomorphism) a subset of the set underlying the algebraic structure, inheriting the interpretation of the operations from its superobject, that is the interpretation of each operation in the subobject is a function of the required class (e.g. the binary operation of a group is associative) and its graph is a subset of the graph of the interpretation of the same operation in the superobject. In the case of partial first-order structures, the same intuition applies; but since some operation symbols are interpreted as (possibly) partial functions, there are different possible generalizations. Let us, indeed, compare the interpretation pf_B of a partial symbol pf in a subobject B of a partial first-order structure A with the interpretation pf_A of the same partial symbol in A . The weakest possible requirement is that pf_B is a partial function and $graph(pf_B) \subseteq graph(pf_A)$. Thus the application of pf_B to an appropriate tuple of elements from the carrier sets of B may be undefined, though the application of pf_A to the same tuple results in a value and this, in turn, may cause some element, that in A is the interpretation of a term, to become “junk”.

Several possible definitions of subobject have been explored and used in literature. For model theory the most interesting are the *weak* and *closed* substructures, that we present and investigate here. A few more notions are shortly presented in Section 3.3.1 and their applications are sketched there and in the following.

Definition 3.25. Let A be a partial Σ -structure; then a partial Σ -structure A_0 is a *weak substructure* (a *subobject*) of A iff

- $|A_0| \subseteq |A|$;
- $f_A(a_1, \dots, a_n) \stackrel{E}{=} f_{A_0}(a_1, \dots, a_n)$ for all $f: s_1 \times \dots \times s_n \longrightarrow s \in \Omega$ and all $a_i \in |A_0|_{s_i}$ for $i = 1, \dots, n$;
- if $pf_A(a_1, \dots, a_n) \downarrow$, then $pf_{A_0}(a_1, \dots, a_n) \stackrel{E}{=} pf_A(a_1, \dots, a_n)$ for all $pf: s_1 \times \dots \times s_n \dashrightarrow s \in P\Omega$ and all $a_i \in |A_0|_{s_i}$ for $i = 1, \dots, n$;
- $p_{A_0} \subseteq p_A$ for all $p: s_1 \times \dots \times s_n \in \Pi$.

A weak substructure Σ -structure A_0 of A is a *closed substructure* (see [Bur82]) of A iff

- $pf_{A_0}(a_1, \dots, a_n) \stackrel{S}{=} pf_A(a_1, \dots, a_n)$ for all $pf: s_1 \times \dots \times s_n \dashrightarrow s \in P\Omega$ and all $a_i \in |A_0|_{s_i}$ for $i = 1, \dots, n$;

- $(a_1, \dots, a_n) \in p_{A_0}$ iff $(a_1, \dots, a_n) \in p_A$ for all $p: s_1 \times \dots \times s_n \in \Pi$ and all $a_i \in |A_0|_{s_i}$ for $i = 1, \dots, n$.

The *embedding* $e: A_0 \hookrightarrow A$ of a weak substructure A_0 into A is the homomorphism of partial Σ -structures whose components are set embeddings.

A Σ -structure without proper closed substructures is said to be *reachable*. \diamond

Thus, many different weak substructures of one Σ -structure sharing the same carriers exist, because partial function and predicate interpretations can be weakened. Opposed to that, each subset of the carriers closed under functional application defines a closed substructure.

Example 3.26. Let us consider the following homogeneous signature

```

sig  $\Sigma =$ 
  sorts    $s$ 
  opns    $c: \rightarrow s$ 
            $f: s \rightarrow s$ 
  popns   $pc: \dashv\rightarrow s$ 
            $pf: s \times s \dashv\rightarrow s$ 
  preds   $p: s \times s \times s$ 

```

and the following Σ -structure A

```

algebra  $A =$ 
  Carriers
   $|A|_s = \{0, \dots, \max\}$ 
  Functions
   $c_A = 0$ 
   $f_A(x) = x$ 
   $pc_A = \max$ 
   $pf_A(x, y) = \begin{cases} x, & \text{if } x = y \\ \text{undefined}, & \text{otherwise} \end{cases}$ 
  Predicates
   $p_A = \{(x, y, z) \mid x = y \text{ or } x = z \text{ or } y = z\}$ 

```

Then any subset of the range $\{0, \dots, \max\}$ including 0 can be the carrier of several weak substructures. For instance, let us consider the singleton $\{0\}$, then the following Σ -structures A_1 and A_2 are both weak substructures of A :

```

algebra  $A_1 =$ 
  Carriers
   $|A_1|_s = \{0\}$ 
  Functions
   $c_{A_1} = 0$ 
   $f_{A_1}(x) = x$ 
   $pc_{A_1} = \text{undefined}$ 
   $pf_{A_1}(x, y) = \begin{cases} x, & \text{if } x = y \\ \text{undefined}, & \text{otherwise} \end{cases}$ 
  Predicates
   $p_{A_1} = \emptyset$ 

```

algebra $A_2 =$
Carriers
 $|A_2|_s = \{0\}$
Functions
 $c_{A_2} = 0$
 $f_{A_2}(x) = x$
 $pc_{A_2} = \text{undefined}$
 $pf_{A_2}(x, y) = \text{undefined}$
Predicates
 $p_{A_2} = \{(0, 0, 0)\}$

and moreover the two weak substructures are not related by homomorphisms in either way. There does not exist a closed substructure of A with $\{0\}$ as carrier, because the interpretation of pc in A is defined but its value does not belong to $\{0\}$. But each subset X of $\{0, \dots, \mathbf{max}\}$ including 0 and \mathbf{max} defines a *unique* closed substructure A_X of A , consisting of:

algebra $A_X =$
Carriers
 $|A_X|_s = X$
Functions
 $c_{A_X} = 0$
 $f_{A_X}(x) = x$
 $pc_{A_X} = \mathbf{max}$
 $pf_{A_X}(x, y) = \begin{cases} x, & \text{if } x = y \\ \text{undefined}, & \text{otherwise} \end{cases}$
Predicates
 $p_{A_X} = \{(x, y, z) \mid x = y \text{ or } x = z \text{ or } y = z, x, y, z \in X\}$

■

As expected in a categorical setting, there is an obvious relation between the notion(s) of subobject and the domain of special kinds of homomorphisms. Indeed, weak substructures are (standard) subobjects, that is domains of monomorphisms.

Proposition 3.27. *Let A and B be partial Σ -structures. A homomorphism of partial Σ -structures $h: A \rightarrow B$ is a monomorphism, that is $h_1; h = h_2; h$ implies $h_1 = h_2$ for all $h_1, h_2: A_0 \rightarrow A$, if and only if h_s is injective for all $s \in S$.*

Analogously, closed substructures are regular subobjects, that is domains of equalizers. Notice that the image of a homomorphism is not, in general, a closed substructure, because the interpretation of a partial function in the target can be more defined than in the source.

Exercise 3.28. Prove that the image of a homomorphism is a weak substructure of the homomorphism target and show an example of a homomorphism whose image is not a closed substructure of the target. Moreover, show that the image of a closed homomorphism is a closed substructure of the homomorphism target.

If a family of functions satisfies the homomorphism conditions for a Σ -structure, then it is a homomorphism into any closed substructure including its image.

Lemma 3.29. *Let A_0 be a closed substructure of a partial Σ -structure A and $h: B \rightarrow A$ be a homomorphism s.t. the image of B is an S -indexed subset of $|A_0|$. Then $h: B \rightarrow A_0$ is a homomorphism.*

The category of partial Σ -structures has equalizers. That is given two parallel homomorphisms $h, h': A \rightarrow B$ there exists a homomorphism $e: E \rightarrow A$ s.t. e equalizes h and h' , i.e. $e; h = e; h'$, and moreover e is *universal*, that is each other homomorphism equalizing h and h' factorizes in a unique way through e , i.e. $k; h = k; h'$ for some $k: K \rightarrow A$ implies that there exists a unique $k': K \rightarrow E$ s.t. $k = k'; e$. Indeed, it is possible to “restrict” the domain of such h and h' to the elements on which they yield the same result.

Proposition 3.30. *Let $h, h': A \rightarrow B$ be parallel homomorphisms of partial Σ -structures; then the equalizer of h and h' is the (embedding of the) closed substructure E of A whose carriers are defined by*

$$|E|_s = \{a \mid a \in |A|_s \text{ and } h_s(a) \stackrel{E}{=} h'_s(a)\}$$

for all $s \in S$ (into A).

Thus, the domains of equalizers are closed substructures¹¹, and hence closed substructures are “regular subobjects”.

While monomorphisms are all injective functions, as in set theory, epimorphisms are not required to be surjective.

Definition 3.31. A homomorphism $h: A \rightarrow B$ is called *generating* (or also *dense*), if the smallest closed substructure of B containing the image of h is B itself. \diamond

Proposition 3.32. *Let $h: A \rightarrow B$ be a homomorphism of partial Σ -structures. B is generated by h iff h is an epimorphism, that is $h; k = h; k'$ implies $k = k'$ for all $k, k': B \rightarrow C$.*

It is worth noting that bijective homomorphisms are not required to be isomorphisms, that is their inverse may not exist. Consider, indeed, the signature Σ with one sort, no total function, one partial constant and no predicates

¹¹ It is also true the converse, that is, for any given substructure E of a Σ -structure A there is a pair of homomorphisms whose equalizer is E itself. However, the result is unnecessary for the model theory we want to present here and the construction of such homomorphisms is not elementary. Indeed, they are basically the embedding of A into a Σ -structure B , built by duplicating the elements of A and then identifying the elements of E , and adding new values to denote the results of total functions on mixed inputs from both copies of A .

at all. Then let us call A the Σ -structure on such signature with a singleton carrier and the interpretation of the constant defined and let us call B its weak substructure, with the same carrier, but with the interpretation of the constant undefined. Then the embedding of B into A is a bijective homomorphism, but there does not exist any homomorphism from A into B , as the constant is defined in A but it is not in B . Therefore, in the category of partial Σ -structures, bimorphisms (i.e. monic epimorphisms) are not required to be isomorphisms.

As usual in most algebraic approaches, a notion of *congruence* is introduced to represent the kernels of homomorphisms.

Definition 3.33. Let A be a partial Σ -structure; a *congruence* on A is an S -sorted family \equiv of subsets \equiv_s of $|A|_s \times |A|_s$ satisfying the following conditions:

- if $a \equiv_s a'$, then $a' \equiv_s a$ (*symmetry*);
- if $a \equiv_s a'$ and $a' \equiv_s a''$, then $a \equiv_s a''$ (*transitivity*);
- if $a_i \equiv_{s_i} a'_i$ for $i = 1, \dots, n$, then $f_A(a_1, \dots, a_n) \equiv_s f_A(a'_1, \dots, a'_n)$ for all $f: s_1 \times \dots \times s_n \rightarrow s \in \Omega$ (*total function closure*);
- if $a_i \equiv_{s_i} a'_i$ for $i = 1, \dots, n$ and both $pf_A(a_1, \dots, a_n) \equiv_s pf_A(a_1, \dots, a_n)^{12}$ and $pf_A(a'_1, \dots, a'_n) \equiv_s pf_A(a'_1, \dots, a'_n)$, then $pf_A(a_1, \dots, a_n) \equiv_s pf_A(a'_1, \dots, a'_n)$ for all $pf: s_1 \times \dots \times s_n \rightarrow s \in P\Omega$ (*partial function weak closure*).

Given a congruence \equiv on a partial Σ -structure A the *domain* of \equiv is the S -family $\{a \mid a \in |A|_s \text{ and } a \equiv_s a\}$; if the domain of \equiv coincides with the whole carrier, then \equiv is called *total*.

Given a congruence \equiv on a partial Σ -structure A , the *quotient* of A by \equiv is the partial Σ -structure A/\equiv defined by:

- $|A/\equiv|_s = \{[a]_{\equiv} \mid a \equiv_s a\}$ for all $s \in S$;
- $f_{A/\equiv}([a_1]_{\equiv}, \dots, [a_n]_{\equiv}) = [f_A(a_1, \dots, a_n)]_{\equiv}$, for all $f: s_1 \times \dots \times s_n \rightarrow s \in \Omega$ and all $[a_i]_{\equiv} \in |A/\equiv|_{s_i}$ for $i = 1, \dots, n$;
- $pf_{A/\equiv}(x_1, \dots, x_n) \stackrel{E}{=} [pf_A(a_1, \dots, a_n)]_{\equiv}$ if $a_i \in x_i$ exist for $i = 1, \dots, n$ s.t. $pf_A(a_1, \dots, a_n) \equiv_s pf_A(a_1, \dots, a_n)$; otherwise it is undefined, for all $pf: s_1 \times \dots \times s_n \rightarrow s \in P\Omega$ and all $x_i \in |A/\equiv|_{s_i}$ for $i = 1, \dots, n$;
- $(x_1, \dots, x_n) \in p_{A/\equiv}$ iff $(a_1, \dots, a_n) \in p_A$ for some $a_i \in x_i$ for $i = 1, \dots, n$, for all $p: s_1 \times \dots \times s_n \in P$ and all $x_i \in |A/\equiv|_{s_i}$ for $i = 1, \dots, n$.

Given a total congruence \equiv on a partial Σ -structure A , we will denote by nat_{\equiv} the homomorphism from A into A/\equiv associating each element with its equivalence class in \equiv .

Let h be a homomorphism of partial Σ -structures from A into B . Then the *kernel* $K(h)$ of h is the total congruence on A defined by $a K(h) a'$ iff $h(a) = h(a')$ for all $a, a' \in |A|_s$ and all $s \in S$. \diamond

¹² Note that this implicitly implies $pf_A(a_1, \dots, a_n) \downarrow$.

It is immediate to verify that A/\equiv is actually a partial Σ -structure for any given congruence \equiv on a partial Σ -structure A . Indeed the conditions on weak partial and total functional closure ensure the unambiguous definition of function interpretation.

Moreover, $K(h)$ is symmetric, reflexive and transitive, by definition, and the conditions of functional closure are guaranteed by the definition of homomorphism. Therefore, $K(h)$ is a total congruence. The first homomorphism theorem holds for our definition of congruence.

Proposition 3.34. *Given a homomorphism of partial Σ -structures h from A into B , there exists a unique $h_K(h): A/K(h) \rightarrow B$ s.t. $h = \text{nat}_{K(h)}; h_K(h)$.*

Different notions of subobjects. Having introduced two definitions of substructure, it can be supposed that there are others as well. And indeed between weak and closed subobjects there are different reasonable possibilities that have been proposed for algebras in literature.

A first requirement that we can impose on a weak substructure B of a partial first-order structure A is that the application of pf_B to an appropriate tuple of elements from the carrier sets of B is defined whenever this is possible, that is if the application of pf_A to the same tuple results in a value within the carrier set of B . This observation leads to the notion of *relative* substructure, where pf_B (regarded as a graph) is the largest subset of pf_A included in the (appropriate cartesian product of the) carrier sets of B . This concept of substructure is particularly relevant, because it allows to regard a bounded implementation of a data type as a substructure of its unbounded abstraction (see e.g. [Grä79]). The important property here is that each subset of the carriers induces a unique relative substructure.

While closed substructures satisfy some upward closure principle w.r.t. partial function application, *normal* substructures satisfy some *downward* closure principle. Normal substructures are important for term evaluation (the downward closure here means that all subterms of an interpretable term are interpretable).

To help the experienced reader to get an intuition of how we are going to generalize the different notions of subalgebra to the case of first-order structures, the key idea is that predicates are thought of as partial functions in a singleton set, identifying definedness of such functions to predicate truth.

Definition 3.35. A weak substructure Σ -structure A_0 of A is a *relative substructure* (see [Bur82]) of A iff

- for all $pf: s_1 \times \dots \times s_n \dashrightarrow s \in P\Omega$ and all $a_i \in |A_0|_{s_i}$ for $i = 1, \dots, n$ if $pf_A(a_1, \dots, a_n) \in |A_0|_s$ then $pf_{A_0}(a_1, \dots, a_n) \stackrel{E}{=} pf_A(a_1, \dots, a_n)$, and
- $(a_1, \dots, a_n) \in p_{A_0}$ iff $(a_1, \dots, a_n) \in p_A$ for all $p: s_1 \times \dots \times s_n \in \Pi$ and all $a_i \in |A_0|_{s_i}$ for $i = 1, \dots, n$.

A relative substructure Σ -structure A_0 of A is a *normal substructure* (see [Sch70]) of A iff for all $pf: s_1 \times \cdots \times s_n \dashrightarrow s \in P\Omega$ and all $a_i \in |A|_{s_i}$ for $i = 1, \dots, n$ if $pf_A(a_1, \dots, a_n) \in |A_0|_s$ then $pf_{A_0}(a_1, \dots, a_n) \stackrel{E}{=} pf_A(a_1, \dots, a_n)$ ¹³. \diamond

As in the case of closed substructure, the carriers completely determine the relative (or normal) substructure having them. Moreover, each subset of the carriers determines a unique relative substructure.

Instead of classifying the different notions of subobject through more or less complex categorical constructions¹⁴, as we did for closed substructures that are domain of equalizers, we can as well see them as standard subobjects in a subcategory, where homomorphisms are restricted to those satisfying some extra requirement. We can also characterize the different kinds of substructures using different specializations of homomorphisms.

Definition 3.36. Let $\Sigma = \langle S, \Omega, P\Omega, \Pi \rangle$ be a partial signature, A and B be partial Σ -structures.

A homomorphism h from A into B is called *full* [Bur82] iff the following conditions hold:

- if $pf_B(h_{s_1}(a_1), \dots, h_{s_n}(a_n)) = b \in |B|_s$ and there exists $a \in |A|_s$ s.t. $h_s(a) = b$ then there exist $a'_i \in |A|_{s_i}$ for $i = 1, \dots, n$ s.t. $h_{s_i}(a_i) = h_{s_i}(a'_i)$ and $pf_A(a'_1, \dots, a'_n) \in |A|_s$ for all $pf \in P\Omega_{w,s}$, where $w = s_1 \times \cdots \times s_n$, and all $a_i \in |A|_{s_i}$ for $i = 1, \dots, n$;
- if $(h_{s_1}(a_1), \dots, h_{s_n}(a_n)) \in p_B$ then there exist $a'_i \in |A|_{s_i}$ for $i = 1, \dots, n$ s.t. $h_{s_i}(a_i) = h_{s_i}(a'_i)$ and $(a'_1, \dots, a'_n) \in p_A$.

A full homomorphism h from A into B is called *normal* iff the following condition holds:

if there exists $a \in |A|_s$ s.t. $h_s(a) \stackrel{E}{=} f_B(b_1, \dots, b_n)$ then there exist $a_i \in |A|_{s_i}$ for $i = 1, \dots, n$ s.t. $h_{s_i}(a_i) = b_i$ and $f_A(a_1, \dots, a_n) \downarrow$ for all $f \in \Omega_{w,s} \cup P\Omega_{w,s}$, where $w = s_1 \times \cdots \times s_n$, and all $b_i \in |B|_{s_i}$ for $i = 1, \dots, n$;

A homomorphism h from A into B is *closed* [Bur82] iff the following conditions hold:

- if $pf_B(h_{s_1}(a_1), \dots, h_{s_n}(a_n)) \downarrow$ then $pf_A(a_1, \dots, a_n) \downarrow$ for all $pf \in P\Omega_{w,s}$, where $w = s_1 \times \cdots \times s_n$, and all $a_i \in |A|_{s_i}$ for $i = 1, \dots, n$;
- if $(h_{s_1}(a_1), \dots, h_{s_n}(a_n)) \in p_B$, then $(a_1, \dots, a_n) \in p_A$ for all $p: s_1 \times \cdots \times s_n \in \Pi$ and all $a_i \in |A|_{s_i}$ for $i = 1, \dots, n$.

\diamond

¹³ Note that this also means that $a_i \in |A_0|_{s_i}$.

¹⁴ For instance, relative substructures are initial subobjects, that is, domains of initial monomorphisms [AHS90] (in a different terminology, they would be called cartesian monomorphisms [Bor94]) w.r.t. to the obvious forgetful functor.

It is worth noting that for any congruence \equiv on a partial first-order structure A , the homomorphism nat_{\equiv} is a full surjection. Moreover, in case that the congruence is not total, nat_{\equiv} can be defined to start from the relative subalgebra induced by $(\{a \mid a \equiv_s a\})_{s \in S}$.

Kind	partial operations	predicates
hom.	$h(\text{graph } pf_A) \subseteq \text{graph } pf_B$	$h(p_A) \subseteq p_B$
full	$h(\text{graph } pf_A) = \text{graph } pf_B \cap h(A)$	$h(p_A) = p_B \cap h(A)$
normal	$h(\text{graph } f_A) = \text{graph } f_B \cap B _{s_1, \dots, s_n} \times h_s(A _s)$	$h(p_A) = p_B \cap h(A)$
closed	$\text{dom } pf_A = h^{-1}(\text{dom } pf_B)$	$p_A = h^{-1}(p_B)$

Proposition 3.37. *Any closed homomorphism is full.*

Proposition 3.38. *A weak substructure is a relative (normal, closed) substructure iff the embedding is a full (normal, closed) homomorphism.*

Corollary 3.39. *Any closed substructure is relative.*

Term evaluation and initiality. Standard term algebras (as defined in the total case) can be endowed with (possibly infinite) choices of predicate interpretations in order to get partial first-order structures (with predicates). But, the usual inductive definition of term evaluation is not a homomorphism, disregarding the interpretation of predicates, because it is, in general, a *partial* function. However, the domain of term evaluation is, by definition, a normal substructure of the term algebra and term evaluation is the unique closed homomorphism extending variable evaluation with a normal substructure as domain (w.r.t. the signature where predicates, that do not play any role in term evaluation, are dropped¹⁵). For a similar presentation, factorizing term evaluation in two steps (the definition of the domain as a substructure satisfying appropriate conditions and the definition of term evaluation as a closed homomorphism) see also [Wol90].

A *term Σ -structure* over X consists of the (total) term algebra $T_{\Sigma}(X)$ and an interpretation $p_{T_{\Sigma}(X)} \subseteq |T_{\Sigma}(X)|_{s_1} \times \dots \times |T_{\Sigma}(X)|_{s_n}$ of each $p: s_1 \times \dots \times s_n \in \Pi$.

In particular we will denote the term Σ -structure with $p_{T_{\Sigma}(X)} = \emptyset$ for all $p: s_1 \times \dots \times s_n \in \Pi$ by $T_{\Sigma}(X)$, and if X is the empty family of variables, $T_{\Sigma}(X)$ will be simply denoted by T_{Σ} .

A *variable valuation* v for X in A is any S -sorted family of partial functions $v_s: X_s \dashrightarrow |A|_s$; given a variable valuation v for X in A , the *term evaluation* $v^{\#}: T_{\Sigma}(X) \dashrightarrow A$ is inductively defined by:

¹⁵ If we let predicates to persist in the signature, then, since closed homomorphism both preserve, as all homomorphisms, and reflect truth, we would need an *ad hoc* definition of predicate interpretation in the term structure for each algebra where we want to evaluate terms.

- $v_s^\#(x) \stackrel{S}{=} v(x)$ for all $x \in X_s$ and all $s \in S$;
- $v_s^\#(f(t_1, \dots, t_n)) \stackrel{S}{=} f_A(v_{s_1}^\#(t_1), \dots, v_{s_n}^\#(t_n))$ for all $f \in \Omega_{w,s} \cup P\Omega_{w,s}$, where $w = s_1 \times \dots \times s_n$, and all $t_i \in |T_\Sigma(X)|_{s_i}$ for $i = 1, \dots, n$;

Given a term t , we say that t is *v-interpretable*, if $t \in \text{dom } v^\#$ and in this case we call $v^\#(t) \in |A|_s$ the *value of t in A under the valuation v*.

In particular if X is the empty family of variables, then v is the empty map for each partial Σ -structure A and we will denote $v_s^\#$ by $eval_A$ and its application to a term t by t_A .

Whenever $v^\#$ is surjective, we say that the Σ -structure A is *generated by v*, and if X is the empty set (and hence v is the empty map), A is simply said *term-generated*.

Proposition 3.40. *A homomorphism h from A into B is generating iff B is generated by h (regarded as a valuation of the family |A| of variables).*

It is straightforward to verify, by induction on the definition of $v^\#$, the following technical lemma, whose proof is left as exercise to the reader.

Lemma 3.41. *Let $\Sigma = \langle S, \Omega, P\Omega, \Pi \rangle$ be a partial signature, A be a partial Σ -structures, X be an S -sorted family of variables, and v be a valuation for X in A .*

- For each term t , if $v^\#(t) \downarrow$, then $h(v^\#(t)) \stackrel{E}{=} (v; h)^\#(t)$ for all homomorphisms $h: A \rightarrow B$.
- For each valuation v' for X in A , if $v^\#(x) \stackrel{W}{=} v'^\#(x)$ for all variables $x \in X$, then $v^\#(t) \stackrel{W}{=} v'^\#(t)$ for all terms t .

Notation 3.42. In the sequel let us fix a partial signature $\Sigma = \langle S, \Omega, P\Omega, \Pi \rangle$.

Closed substructures are closed w.r.t. term evaluation, while weak substructures are not.

Exercise 3.43. Prove that if the valuation of a family of variables is contained in the carriers of a closed substructure, the evaluation of any term w.r.t. such a valuation in the Σ -structure and in the closed substructure are strongly equal (while for weak substructures, their are only weakly equal).

Corollary 3.44. *A Σ -structure is term-generated if and only if it is reachable (i.e. it does not have proper closed substructures).*

Definition 3.45. Let \mathcal{A} be a class of partial first-order structures over a partial first-order signature $\Sigma = \langle S, \Omega, P\Omega, \Pi \rangle$ and X be an S -sorted family of variables.

Then a partial first-order structure $F \in \mathcal{A}$ is *free* for X in \mathcal{A} iff there exists a total valuation e for X in F s.t. for all $A \in \mathcal{A}$ and all total valuations v for X in A a unique homomorphism $h_v: F \rightarrow A$ exists s.t. $v = e; h_v$.

A free partial first-order structure $F \in \mathcal{A}$ for the empty family of variables in \mathcal{A} is called *initial* in \mathcal{A} . \diamond

Lemma 3.46. *Let \mathcal{A} be a class of partial first-order structures over a partial first-order signature $\Sigma = \langle S, \Omega, P\Omega, \Pi \rangle$ closed under closed substructures. Then a free first-order structure for an S -sorted family X of variables in \mathcal{A} , if any, is generated by X .*

3.3.2 Partial logic

We now want to generalize the concept of describing classes of algebras by axioms introduced in Chapter 2 and extended at the beginning of this chapter, by allowing conditional axioms built starting not only from equalities, but also from predicate symbols applied to tuples of terms.

The presence of partial functions introduces the possibility of terms which do not denote in all structures. This phenomenon causes different possible generalizations of the concept of equation to the partial case. Thus, the proliferation of equality symbols that we introduced at the meta-level also reflects on formulae, having three different kinds of atomic formulae representing respectively existential, weak and strong equalities between terms, besides atomic predicate formulae.

Moreover, in Section 3.1, we only allowed universally quantified conditional axioms, since they have the nice properties that initial models and relatively fast theorem provers exist. Here, we pass over to full first-order logic. This increased expressiveness of axioms allows to write requirement specifications (to be interpreted loosely) which are more succinct and more related to informal requirements than specifications with conditional equations can be, as illustrated at the end of Example 3.15. Moreover, there are interesting data types that cannot be directly expressed within the conditional fragment; see, for instance, Section 3.4 below. Since the existence of initial models is needed sometimes (e.g. for initial constraints or for design specifications), we later identify those fragments of first-order logic which still have initial models.

We now want to use terms for building formulae, which will eventually serve as axioms in specifications.

Definition 3.47 (First-order formula). Let $\Sigma = \langle S, \Omega, P\Omega, \Pi \rangle$ be a signature. We inductively define for all S -sorted sets X in parallel the set $Form(\Sigma, X)$ of Σ -formulae in variables X . $Form(\Sigma, X)$ is the least S -sorted set containing

- $t_1 \stackrel{e}{=} t_2$ for $t_1, t_2 \in |T_\Sigma(X)|_s$
- $p(t_1, \dots, t_n)$ for $p: s_1 \times \dots \times s_n \in \Pi$ and $t_i \in |T_\Sigma(X)|_{s_i}$, $i = 1, \dots, n$
- F (read: false)
- $(\varphi \wedge \psi)$ and $(\varphi \Rightarrow \psi)$ for $\varphi, \psi \in Form(\Sigma, X)$
- $(\forall Y.\varphi)$ for $\varphi \in Form(\Sigma, X \cup Y)$, Y an S -sorted set

If there is no ambiguity, the brackets around $(\varphi \wedge \psi)$ etc. can be omitted.

We define the following abbreviations:

- $(\neg\varphi)$ stands for $(\varphi \Rightarrow F)$
- $(\varphi \vee \psi)$ stands for $\neg(\neg\varphi \wedge \neg\psi)$
- $(\varphi \Leftrightarrow \psi)$ stands for $(\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$
- $D(t)$ stands for $t \stackrel{e}{=} t$
- $t_1 \stackrel{s}{=} t_2$ stands for $(D(t_1) \vee D(t_2)) \Rightarrow t_1 \stackrel{e}{=} t_2$
- $t_1 \stackrel{w}{=} t_2$ stands for $(D(t_1) \wedge D(t_2)) \Rightarrow t_1 \stackrel{e}{=} t_2$
- $(\exists Y.\varphi)$ stands for $\neg(\forall Y.\neg\varphi)$ ◇

Definition 3.48 (First-order axiom). A *first-order axiom* over a signature Σ is a pair (X, φ) , written $X.\varphi$, where $\varphi \in \text{Form}(\Sigma, X)$.

A first-order axiom $X.\varphi$ is called *closed*, if $X = \emptyset$. ◇

The usual definition of free variables of a term or a formula now becomes easy.

Definition 3.49. Given a term $t \in T_\Sigma(X)$, the set $FV(t)$ of *free variables of t* is the least set $X' \subseteq X$ such that already $t \in T_\Sigma(X')$.

Likewise, given a formula $\varphi \in \text{Form}(\Sigma, X)$, the set $FV(\varphi)$ of *free variables of φ* is the least set $X' \subseteq X$ such that already $\varphi \in \text{Form}(\Sigma, X')$. ◇

It is common practice to leave out, in the definition of axioms, the family X of variables over which the formula is defined and just write φ , where X is recovered as $FV(\varphi)$. But, at least for a semantics based on total valuations, it is essential to allow also axioms $X.\varphi$ where $FV(\varphi)$ is a proper subset of X , which may behave different from $FV(\varphi).\varphi$, see Section 3.3.3.

To be able to formally understand what a model of a specification is, we now have to define satisfaction of first-order axioms by first-order structures. According to Feferman [Fef95], there are basically two different ways to define satisfaction: in *logics of existence*, satisfaction is defined using *partial* variable valuations, i.e. variables may be undefined. Opposed to that, in *logics of definedness*, satisfaction is defined using *total* variable valuations, i.e. variables are always defined. Note that in both cases, we quantify over the defined only – bound variables are always defined. Therefore, quantification is treated by extension of valuations which have to be defined for the quantified variables.

Definition 3.50 (Satisfaction). A partial Σ -structure A *satisfies* a first-order axiom $X.\varphi$ *w.r.t. total valuations* (written $A \models_\Sigma^t X.\varphi$), if all total valuations $v: X \rightarrow |A|$ satisfy φ .

A partial Σ -structure A *satisfies* a first-order axiom $X.\varphi$ *w.r.t. partial valuations* (written $A \models_\Sigma^p X.\varphi$), if all (partial or total) valuations $v: X \rightarrow |A|$ satisfy φ .

Satisfaction of a formula $\varphi \in \text{Form}(\Sigma, X)$ by a (possibly partial) valuation $v: X \rightarrow |A|$ is defined inductively over the structure of φ :

- $v \Vdash_\Sigma t_1 \stackrel{e}{=} t_2$ iff $v^\#(t_1) \stackrel{E}{=} v^\#(t_2)$

- $v \Vdash_{\Sigma} p(t_1, \dots, t_n)$ iff $v^{\#}(t_1) \downarrow$ and \dots and $v^{\#}(t_n) \downarrow$ and $(v^{\#}(t_1), \dots, v^{\#}(t_n)) \in p_A$
- not $v \Vdash_{\Sigma} F$
- $v \Vdash_{\Sigma} (\varphi \wedge \psi)$ iff $v \Vdash_{\Sigma} \varphi$ and $v \Vdash_{\Sigma} \psi$
- $v \Vdash_{\Sigma} (\varphi \Rightarrow \psi)$ iff $v \Vdash_{\Sigma} \varphi$ implies $v \Vdash_{\Sigma} \psi$
- $v \Vdash_{\Sigma} (\forall Y.\varphi)$ iff for all valuations $\xi: X \cup Y \rightarrow |A|$ which
 - extend v on $X \setminus Y$ (i.e. $\xi(x) \stackrel{S}{=} v(x)$ for all $x \in X_s \setminus Y_s, s \in S$) and
 - are defined on Y (i.e. $\xi(y) \downarrow$ for $y \in Y_s, s \in S$)
 we have $\xi \Vdash_{\Sigma} \varphi$.

◊

Thus, we treat quantification by extensions of valuations to the quantified variables. By requiring ξ to be an extension of v on $X \setminus Y$ only, variables in Y are treated as fresh variables: their value under v is disregarded within φ .

Satisfaction of arbitrary first-order axioms w.r.t. total valuations can be reduced to that of closed axioms, because the satisfaction of a quantified formula is equivalent to that of its universal closure.

Exercise 3.51. Prove that if a formula φ is satisfied by some valuation v then it is satisfied by all valuations coinciding with v on the free variables of φ .

Exercise 3.52. Prove that for a partial Σ -structure A and $\varphi \in \text{Form}(\Sigma, X)$

$$A \models_{\Sigma}^t X.\varphi \text{ if and only if } A \models_{\Sigma}^t \emptyset.\forall X.\varphi \text{ if and only if } A \models_{\Sigma}^p \emptyset.\forall X.\varphi$$

The counterexample $A \models_{\Sigma}^p \emptyset.\forall x : s.x \stackrel{e}{=} x$ but $A \not\models_{\Sigma}^p \{x : s\}.x \stackrel{e}{=} x$, as not $v \Vdash_{\Sigma} x \stackrel{e}{=} x$ for v the totally undefined partial valuation, shows that open formulae are interpreted by \models^p quite differently.

Proposition 3.53. *Let X and Y be two S -sorted variable systems and $\varphi \in \text{Form}(\Sigma, X \cap Y)$ be a first-order formula. Then for any partial Σ -structure A ,*

$$A \models_{\Sigma}^p Y.\varphi \text{ if and only if } A \models_{\Sigma}^p X.\varphi$$

while the corresponding property for \models^t does not hold, unless all carrier sets of A are non-empty.

Definition 3.54 (Semantical consequence). A first-order axiom $X.\varphi$ is said to *follow semantically w.r.t. total valuations* (resp. w.r.t. partial valuations) from a set of first-order axioms M , written $M \models_{\Sigma}^t \varphi$ (resp. $M \models_{\Sigma}^p \varphi$), if for all total (resp. total or partial) valuations $v: X \cup \bigcup_{Y.\psi \in M} Y \rightarrow |A|$ into partial Σ -structures A we have:

$$\text{if } v|_Y \Vdash_{\Sigma} \psi \text{ for all } Y.\psi \in M, \text{ then } v|_X \Vdash_{\Sigma} \varphi \quad \diamond$$

Proposition 3.53 can be easily extended to semantical consequence.

Proposition 3.55. *Let X and Y be to S -sorted variable systems and $M \cup \{\varphi\} \subseteq \text{Form}(\Sigma, X \cap Y)$ be first-order formulae. Then*

$$M \models_{\Sigma}^p Y.\varphi \text{ if and only if } M \models_{\Sigma}^p X.\varphi$$

while the corresponding property for \models^t does not hold.

The peculiarity of \models^t shown in Propositions 3.53 and 3.55 is not introduced by the extension of logical power, but it is already present in the total many-sorted equational fragment, where it leads to inconsistent calculi unless quantification is very carefully treated. For this reason in most part of the literature¹⁶ on total algebras empty carrier sets are not allowed or they are required to be unconnected to the non-empty carriers by any function symbol. In [HO80] syntactical conditions on signatures are given, guaranteeing that the empty carriers cannot introduce troubles. Not only such conditions are not significant anymore for the partial case, but in the context of specification, there may be very well the situation of some data set being empty, for instance during the requirement phase, before the decisions on some kind of elements have been completed. Thus, we do not require that the models of a specification have non-empty carriers.

Both Exercise 3.52 and Proposition 3.53 (the latter together with its companion 3.55) do hold for one-sorted total first-order logic. When generalizing to the partial many-sorted case, we cannot keep both true. So we have to choose between the equivalence of formulae to their universal closure (which holds for \models^t) and invariance under changes of the variable system (which holds for \models^p).¹⁷ While many treatments of partial logics [Bur82, Bee85, Rei87] are guided by the former, we prefer the latter, also because of the easier Substitution Lemma for \models^p (see Lemmas 3.62 and 3.63 below). The price for this preference is a slightly more complex manipulation of quantification. But the invariance under changes of the variable system of \models^p allows us now to drop the variable system.

Notation 3.56. Concerning \models_{Σ}^p , we may drop the variable system from formulae and understand φ as an abbreviation of $FV(\varphi).\varphi$.

As in Section 2.2, we define a presentation to be a pair $\langle \Sigma, AX \rangle$ where AX is a set of Σ -first-order axioms. A *model* of a presentation $\langle \Sigma, AX \rangle$ is

¹⁶ Indeed, the only references developing full many-sorted first-order logic with possibly empty carriers we found are [AR94, Hai53, KGS88] and [KN94].

¹⁷ Of course, we can keep both true if we define the semantics of quantification over partial extensions of valuations, so that quantified variables, as free variables, need not denote a value. But then, in practice, we have to add many definedness conditions to quantified axioms.

Another possibility to keep both conditions true, for total satisfaction, is to restrict the models to those with non-empty carriers, but this is too restrictive from a methodological point of view during the initial phases, as already mentioned.

a partial Σ -structure A such that $A \models_{\Sigma}^p AX$. $Mod_{\Sigma}(AX)$ is the class of all models of $\langle \Sigma, AX \rangle$.

Definition 3.57. A presentation $\langle \Sigma, AX \rangle$ is called *semantically inconsistent*, if $Mod_{\Sigma}(AX)$ is empty. Otherwise, it is called *semantically consistent*. \diamond

Proposition 3.58. For Σ -first-order formulae $\varphi_1, \dots, \varphi_n, \psi$, the following are equivalent:

1. $M \cup \{ \varphi_1, \dots, \varphi_n \} \models_{\Sigma}^p \psi$;
2. $M \models_{\Sigma}^p \varphi_1 \wedge \dots \wedge \varphi_n \Rightarrow \psi$;
3. $\langle \Sigma, M \cup \{ \varphi_1, \dots, \varphi_n, \neg\psi \} \rangle$ is inconsistent.

Example 3.59. An easy example of an inconsistent presentation is given, as usual, requiring $A \wedge \neg A$ for some formula A without free variables.

```
spec INCONSISTENT =
  sorts    s
  preds   p : s
  axioms  ( $\forall x : s. p(x) \wedge (\neg \forall x : s. p(x))$ )
```

■

It is interesting to note that the following, quite similar, specification is *not* inconsistent.

```
spec PECULIAR =
  sorts    s
  preds   p : s
  axioms   $\forall x : s. (p(x) \wedge \neg p(x))$ 
```

Indeed it has the empty structure as a model, because if the carrier of sort s is empty, there does not exist a total valuation for $\{x\}$ in it and hence $\forall x : s. A$ is satisfied disregarding the formula A .

3.3.3 Proof theory

Whereas model theory introduced in the previous section lays the foundation for specification of data types (understood as partial algebras), proof theory is essential for deriving in a syntactical, computable way the semantical consequences of a specification. The consequences may not only reveal wanted or unwanted behavior of the specified system, but possibly also the inconsistency of the specification.

We here present two natural deduction-style proof calculi for partial first-order logic. The first one was developed by Burmeister [Bur82] to capture \models^t . Burmeister's calculus covers only the one-sorted case. Here, in accordance with the previous sections, we generalize it to the many-sorted case (also allowing carriers to be empty), which forces us to carefully keep track of variables (cf. [GM85,GM86a]). The second calculus captures \models^p and follows

the ideas of Scott [Sco79]. In this calculus, because of Proposition 3.55, we can omit the variable system.

Both calculi are based on a notion of substitution. The usual notion (see Exercise 1.4.9) can be easily generalized to the partial case.

Definition 3.60. A function $\theta: X \rightarrow |T_\Sigma(Y)|$ is called a *substitution*. Given a substitution $\theta: X \rightarrow |T_\Sigma(Y)|$ and an S -sorted variable system Z , we denote by $\theta \setminus Z: X \cup Z \rightarrow T_\Sigma(Y \cup Z)$ the substitution being the identity on Z and being θ on $X \setminus Z$. \diamond

Substitutions can be applied to terms as well as to formulae, where in the case of formulae, the application is not defined in all cases because of possible name clashes of substituted with quantified variables.

Definition 3.61. The term $t[\theta] \in T_\Sigma(Y)$ resulting from applying the substitution θ to a term $t \in T_\Sigma(X)$ is defined by

$$t[\theta] = \theta^\#(t)$$

The formula $\varphi[\theta] \in Form(\Sigma, Y)$, which, if defined, results from applying the substitution θ to a formula $\varphi \in Form(\Sigma, X)$ is defined inductively over φ :

- $(t_1 \stackrel{e}{=} t_2)[\theta] \stackrel{E}{=} t_1[\theta] \stackrel{e}{=} t_2[\theta]$
- $p(t_1, \dots, t_n)[\theta] \stackrel{E}{=} p(t_1[\theta], \dots, t_n[\theta])$
- $F[\theta] \stackrel{E}{=} F$
- $(\varphi \wedge \psi)[\theta] \stackrel{S}{=} (\varphi[\theta]) \wedge (\psi[\theta])$
- $(\varphi \Rightarrow \psi)[\theta] \stackrel{S}{=} (\varphi[\theta]) \Rightarrow (\psi[\theta])$
- $(\forall Z.\varphi)[\theta] \stackrel{S}{=} \begin{cases} \forall Z.(\varphi[\theta \setminus Z]), & \text{if } \forall x \in X_s, s \in S. \\ & (x[\theta] \neq x \text{ and } x \in FV(\forall Z.\varphi) \\ & \text{implies } Z \cap FV(x[\theta]) = \emptyset) \\ \text{undefined,} & \text{otherwise} \end{cases}$

\diamond

The last case, causing $(\forall Z.\varphi)[\theta]$ to be undefined in the case of name clashes, prevents a free variable in $x[\theta]$ to get bound by the quantification over Z . This restriction is important to keep the intended semantics of substitutions. This semantics is reflected by the following Lemma from [Bur82].

Lemma 3.62 (Substitution Lemma for \models^t). *Let A be a partial Σ -structure, $v: Y \rightarrow A$ be a total valuation, $\theta: X \rightarrow |T_\Sigma(Y)|$ be a substitution and $\varphi \in Form(\Sigma, X)$ a formula. Under the conditions that*

- $v^\# \circ \theta: X \rightarrow A$ is a total valuation as well (i.e. for all $x \in X_s, s \in S$ we have $v \Vdash_\Sigma D(\theta(x))$) and
- $\varphi[\theta]$ is defined,

we have

$$v^\# \circ \theta \Vdash_\Sigma \varphi \text{ if and only if } v \Vdash_\Sigma \varphi[\theta]$$

Compared with the usual Substitution Lemma for total logics, we here have to make the additional assumption that the terms being substituted are defined. On the other hand, the Substitution Lemma for \models^p keeps the simplicity of substitution in the total case.

Lemma 3.63 (Substitution Lemma for \models^p). *Let A be a partial Σ -structure, $v: Y \rightarrow A$ be a (total or partial) valuation, $\theta: X \rightarrow |T_\Sigma(Y)|$ be a substitution and $\varphi \in \text{Form}(\Sigma, X)$ a formula. Under the condition that $\varphi[\theta]$ is defined, we have*

$$v^\# \circ \theta \Vdash_\Sigma \varphi \text{ if and only if } v \Vdash_\Sigma \varphi[\theta]$$

The more complicated Substitution Lemma for \models^t also complicates the rules of the calculus dealing with substitution, while the others rules can be taken directly from total first-order logic.

A calculus for \models^t . Let Φ, Φ_1, Φ_2 , and Φ_3 be finite sets of formulae in $\text{Form}(\Sigma, X)$. Application of substitution to such sets is understood element-wise. We introduce the following rules of derivation:

Assumption

$$\frac{}{\Phi \vdash_{\Sigma, X}^t \varphi \quad \varphi \in \Phi}$$

\wedge -introduction

$$\frac{\begin{array}{l} \Phi_1 \quad \vdash_{\Sigma, X}^t \varphi \\ \Phi_2 \quad \vdash_{\Sigma, X}^t \psi \end{array}}{\Phi_1 \cup \Phi_2 \vdash_{\Sigma, X}^t (\varphi \wedge \psi)}$$

\wedge -left elimination

$$\frac{\Phi \vdash_{\Sigma, X}^t (\varphi \wedge \psi)}{\Phi \vdash_{\Sigma, X}^t \varphi}$$

\wedge -right elimination

$$\frac{\Phi \vdash_{\Sigma, X}^t (\varphi \wedge \psi)}{\Phi \vdash_{\Sigma, X}^t \psi}$$

Tertium non datur

$$\frac{\begin{array}{l} \Phi_1 \cup \{\varphi\} \quad \vdash_{\Sigma, X}^t \psi \\ \Phi_2 \cup \{(\varphi \Rightarrow F)\} \quad \vdash_{\Sigma, X}^t \psi \end{array}}{\Phi_1 \cup \Phi_2 \quad \vdash_{\Sigma, X}^t \psi}$$

Absurdity

$$\frac{\Phi \vdash_{\Sigma, X}^t F}{\Phi \vdash_{\Sigma, X}^t \psi}$$

Cut

$$\frac{\begin{array}{l} \Phi_1 \quad \vdash_{\Sigma, X}^t \quad \varphi_1 \wedge \cdots \wedge \varphi_n \Rightarrow \psi_i \\ \Phi_2 \quad \vdash_{\Sigma, Y}^t \quad \psi_1 \wedge \cdots \wedge \psi_k \Rightarrow \epsilon \end{array}}{\Phi_1 \cup \Phi_2 \vdash_{\Sigma, X \cup Y}^t \psi_1 \wedge \cdots \wedge \psi_{i-1} \wedge \varphi_1 \wedge \cdots \wedge \varphi_n \wedge \psi_{i+1} \wedge \cdots \wedge \psi_k \Rightarrow \epsilon}$$

 \Rightarrow -introduction

$$\frac{\Phi \cup \{\varphi_1, \dots, \varphi_n\} \vdash_{\Sigma, X}^t \psi}{\Phi \vdash_{\Sigma, X}^t \varphi_1 \wedge \cdots \wedge \varphi_n \Rightarrow \psi}$$

 \forall -elimination

$$\frac{\Phi \vdash_{\Sigma, X}^t (\forall Y, \varphi)}{\Phi \vdash_{\Sigma, X \cup Y}^t \varphi}$$

 \forall -introduction

$$\frac{\Phi \vdash_{\Sigma, X \cup Y}^t \varphi}{\Phi \vdash_{\Sigma, X}^t (\forall Y, \varphi)} \quad \text{if } Y \cap FV(\Phi) = \emptyset$$

Reflexivity

$$\frac{}{\Phi \vdash_{\Sigma, X}^e x \stackrel{e}{=} x} \quad \text{for } x \in X_s$$

Congruence

$$\frac{\Phi \vdash_{\Sigma, X}^t \varphi}{\Phi \vdash_{\Sigma, X \cup Y}^t (\bigwedge_{x \in X_s} x \stackrel{e}{=} \theta(x)) \Rightarrow \varphi[\theta]} \quad \text{for } \theta: X \longrightarrow |T_\Sigma(Y)| \text{ with } \varphi[\theta] \downarrow$$

Substitution

$$\frac{\Phi \vdash_{\Sigma, X}^t \varphi}{\Phi[\theta] \vdash_{\Sigma, Y}^t (\bigwedge_{x \in X_s} D(\theta(x))) \Rightarrow \varphi[\theta]} \\ \text{for } \theta: X \longrightarrow |T_\Sigma(Y)| \text{ with } \varphi[\theta] \downarrow \text{ and } \Phi[\theta] \downarrow$$

Function Strictness

$$\frac{\Phi \vdash_{\Sigma, X}^t t_1 \stackrel{e}{=} t_2}{\Phi \vdash_{\Sigma, X}^t D(t)} \quad t \text{ some subterm of } t_1 \text{ or } t_2$$

Predicate Strictness

$$\frac{\Phi \vdash_{\Sigma, X}^t p(t_1, \dots, t_n)}{\Phi \vdash_{\Sigma, X}^t D(t_i)} \quad \text{for } p: s_1 \times \cdots \times s_n \in \Pi$$

Totality

$$\frac{\Phi \vdash_{\Sigma, X}^t \bigwedge_{i=1, \dots, n} D(t_i)}{\Phi \vdash_{\Sigma, X}^t D(f(t_1, \dots, t_n))} \quad \text{for } f: s_1 \times \cdots \times s_n \longrightarrow s \in \Omega$$

Here $D(t)$ is syntactical sugar for $t \stackrel{e}{=} t$.

A *derivation* is a finite sequence of judgments of form $\Phi \vdash_{\Sigma, X}^t \varphi$ such that each member of the sequence is either an axiom or obtained from previous members of the sequence by application of a rule. A *derivation of a formula* $X.\varphi$ is a derivation whose last member is the judgment $\emptyset \vdash_{\Sigma, X}^t \varphi$.

Burmeister states the following theorem in [Bur82].

Theorem 3.64. *The calculus is sound and complete, i.e.*

$$\Phi \models_{\Sigma}^t X.\varphi \text{ if and only if } \Phi \vdash_{\Sigma, X \cup \bigcup_{Y, \psi \in \Phi} Y}^t \varphi$$

Derived rules for the defined connectives and quantifiers can be found in [Her73]. If we want to have a calculus with definedness and strong equality as basic notions, we have to add the following derived rules:

Reflexivity

$$\frac{}{\Phi \vdash_{\Sigma, X}^t t \stackrel{s}{=} t} \text{ for } t \in |T_{\Sigma}(X)|_s$$

Equality1

$$\frac{\Phi \vdash_{\Sigma, X}^t t_1 \stackrel{e}{=} t_2}{\Phi \vdash_{\Sigma, X}^t t_1 \stackrel{s}{=} t_2}$$

Equality2

$$\frac{\begin{array}{l} \Phi \vdash_{\Sigma, X}^t D(t_1) \\ \Phi \vdash_{\Sigma, X}^t t_1 \stackrel{s}{=} t_2 \end{array}}{\Phi \vdash_{\Sigma, X}^t t_1 \stackrel{e}{=} t_2}$$

Equality3

$$\frac{\begin{array}{l} \Phi \vdash_{\Sigma, X}^t D(t_1) \Rightarrow F \\ \Phi \vdash_{\Sigma, X}^t D(t_2) \Rightarrow F \end{array}}{\Phi \vdash_{\Sigma, X}^t t_1 \stackrel{s}{=} t_2}$$

and replace $\stackrel{e}{=}$ by $\stackrel{s}{=}$ in **Congruence**. Moreover, the old **Reflexivity** can be dropped.

All rules of the calculus up to **Congruence** are taken from a calculus for total first-order logic [Her73] (except that we index judgments with signatures and variables here, which is necessary to cover the case of empty carriers). On the other hand, the last four rules are entirely new. **Function Strictness** and **Predicate Strictness** state that atomic formulae are interpreted “existentially strict”, that is, their truth entails definedness of all terms occurring in them. **Totality** states that the application of a total function to defined terms is defined.

Finally, the **Substitution** Rule also occurs in the calculus for total first-order logic, but has to be modified for the partial case: it contains additional

assumptions

$$\{ D(\theta(x)) \mid x \in X_s \}$$

in the derived judgment, which state that the terms to be substituted are defined. This is a syntactical version of the definedness condition in the Substitution Lemma for \models^t .

The calculus (especially when extended with suitable derived rules¹⁸) is useful for doing proofs whose structure follows the reasoning of a mathematician. But for automated theorem proving, more efficient proof calculi are used, like analytic tableaux, resolution and the connection structure method [G⁺93]. One crucial source of efficiency is the use of unification (i.e. finding a substitution under which two given formulae become equal). But in the above calculus, the rule of substitution is restricted to the case where the things being substituted are defined. This causes difficulties, at least, when using the well-known techniques and results based on unification.

A calculus for \models^p . This is a further strong argument in favor of \models^p , which can be also captured by a proof calculus. This calculus consists of the rules **Assumption**, **\wedge -introduction**, **\wedge -left elimination**, **\wedge -right elimination**, **Tertium non datur**, **Absurdity**, **Predicate Strictness**, **Function Strictness** and **Totality** which are obtained by the corresponding rules from the above calculus by dropping the variable system as an index for \vdash , and the following further rules:

\forall -elimination

$$\frac{\Phi \vdash_{\Sigma}^p (\forall Y. \varphi)}{\Phi \vdash_{\Sigma}^p (\bigwedge_{y \in Y_s, s \in S} D(y)) \Rightarrow \varphi}$$

\forall -introduction

$$\frac{\Phi \vdash_{\Sigma}^p (\bigwedge_{y \in Y_s, s \in S} D(y)) \Rightarrow \varphi}{\Phi \vdash_{\Sigma}^p (\forall Y. \varphi)} \quad \text{if } Y \cap FV(\Phi) = \emptyset$$

Symmetry

$$\frac{}{\Phi \vdash_{\Sigma}^p x \stackrel{e}{=} y \Rightarrow y \stackrel{e}{=} x}$$

Substitution

$$\frac{\Phi \vdash_{\Sigma}^p \varphi}{\Phi[\theta] \vdash_{\Sigma}^p \varphi[\theta]}$$

for $\theta: X \rightarrow |T_{\Sigma}(Y)|$ with $\varphi[\theta] \downarrow$ and $\Phi[\theta] \downarrow$

¹⁸ Note that the derived rule for introduction of existential quantifiers has to leave the variable system untouched. In particular, from $\vdash_{\Sigma, \{x:s\}} x \stackrel{e}{=} x$ we can only derive $\vdash_{\Sigma, \{x:s\}} \exists x : s. x \stackrel{e}{=} x$, but not $\vdash_{\Sigma, \emptyset} \exists x : s. x \stackrel{e}{=} x$.

Again, $D(t)$ is syntactical sugar for $t \stackrel{e}{=} t$.

Theorem 3.65. *The calculus is sound and complete, i.e.*

$$\Phi \models_{\Sigma}^p \varphi \text{ if and only if } \Phi \vdash_{\Sigma}^p \varphi$$

This calculus has a simple substitution rule **Substitution**, while the quantifier rules **\forall -elimination** and **\forall -introduction** now have to take care of definedness. Reflexivity of $\stackrel{e}{=}$ does no longer hold and has to be replaced by symmetry, which is strictly weaker because it follows from reflexivity together with **Congruence**. (Transitivity follows by **Congruence** in either case.)

Translating partial to total first-order logic. The success for total first-order logic is based on the fact, that it is expressive enough to be called a *universal logic* in [MT93] but just not too expressive, so there is a sound and complete calculus. First-order logic being universal means that there are translations from many-sorted, higher-order, dynamic, modal etc. logics to first-order logic. We will now describe a translation from partial first-order logic (denoted by PFOL) to total first-order logic, from now on denoted by FOL. This translation allows us to take any deductive system for total first-order logic and re-use it for PFOL (either with \models^t or \models^p). This is a particular case of the *borrowing* technique proposed in [CM97].

Of course, since PFOL is a superset of FOL, it shares the universal character with FOL. On the other hand, it becomes clear that no essential expressive power is added by the passage from FOL to PFOL, but, as we shall see, we gain much notational convenience.

We here use the standard FOL introduced in many textbooks. That is, a FOL-signature consists of a PFOL-signature with exactly one sort symbol and no partial operation symbols. Σ -structures have to have a non-empty carrier, in order to get a simple calculus. Such a calculus for FOL consists of the rules **Assumption**, **\wedge -introduction**, **\wedge -left elimination**, **\wedge -right elimination**, **Tertium non datur**, **Absurdity**, **\forall -elimination**, **\forall -introduction**, **Reflexivity**, **Substitution** and **Congruence**. For the rules **\forall -elimination**, **\forall -introduction**, **Reflexivity** and **Congruence**, the variable system has to be dropped and $\stackrel{e}{=}$ has to be replaced by $=$. This gives us an entailment relation \vdash_{Σ}^{FOL} .

The idea is now to translate PFOL to FOL and then re-use the easier calculus for FOL via this translation. In particular, the world of automated theorem provers for FOL can thus be adapted for PFOL.

The only translation from partial to FOL fully representing PFOL-structures and -homomorphisms is the representation of partial operations by their graph relations, sketched by Burmeister in [Bur82]. Substitution of terms containing partial operation symbols is avoided, but instead applications of partial operation symbols have to be expanded into long existentially quantified conjunctions: a term $pf(t_1, \dots, t_n)$ is translated to the formula

$$\begin{aligned} \alpha(pf(t_1, \dots, t_n)) = \\ \exists x_1 : s_1, \dots, x_n : s_n, x : s. \\ (R^{pf}(x_1, \dots, x_n, x) \wedge \alpha(x_1 \stackrel{e}{=} t_1) \wedge \dots \wedge \alpha(x_n \stackrel{e}{=} t_n) \wedge \alpha(x \stackrel{e}{=} t)) \end{aligned}$$

This makes the treatment of partial operations even more cumbersome than in Burmeister's calculus.

But there is another translation from PFOL to FOL along the lines of Scott's ideas [Sco79]. Though model categories are not represented faithfully, proof theory is, so it fits for our purposes here.

A PFOL-signature $\Sigma = \langle S, \Omega, P\Omega, \Pi \rangle$ is translated to a FOL-signature $\Phi(\Sigma) = (\{ * \}, \Omega' \uplus P\Omega', \Pi')$, where Ω' and $P\Omega'$ result from Ω and $P\Omega$ by replacing all sorts by $*$, and $\Pi' = \Pi \cup \{ \equiv_s : s \times s \mid s \in S \}$. To the translated signature, there has to be added the set of axioms $C(\Sigma)$ consisting of

- $x \equiv_s y \Rightarrow y \equiv_s x$ for $s \in S$
- $x \equiv_s y \wedge y \equiv_s z \Rightarrow x \equiv_s z$ for $s \in S$
- $x_1 \equiv_{s_1} y_1 \wedge \dots \wedge x_n \equiv_{s_n} y_n \Rightarrow f(x_1, \dots, x_n) \equiv_s f(y_1, \dots, y_n)$ for $f: s_1 \times \dots \times s_n \rightarrow s \in \Omega$
- $x_1 \equiv_{s_1} y_1 \wedge \dots \wedge x_n \equiv_{s_n} y_n \wedge pf(x_1, \dots, x_n) \equiv_s pf(x_1, \dots, x_n) \Rightarrow pf(x_1, \dots, x_n) \equiv_s pf(y_1, \dots, y_n)$ for $pf: s_1 \times \dots \times s_n \dashrightarrow s \in P\Omega$
- $x_1 \equiv_{s_1} y_1 \wedge \dots \wedge x_n \equiv_{s_n} y_n \wedge p(x_1, \dots, x_n) \Rightarrow p(y_1, \dots, y_n)$ for $p: s_1 \times \dots \times s_n \in \Pi$
- $f(x_1, \dots, x_n) \equiv_s f(x_1, \dots, x_n) \Rightarrow x_i \equiv_{s_i} x_i$ for $f: s_1 \times \dots \times s_n \rightarrow s \in \Omega$
- $pf(x_1, \dots, x_n) \equiv_s pf(x_1, \dots, x_n) \Rightarrow x_i \equiv_{s_i} x_i$ for $pf: s_1 \times \dots \times s_n \dashrightarrow s \in P\Omega$
- $p(x_1, \dots, x_n) \Rightarrow x_i \equiv_{s_i} x_i$ for $p: s_1 \times \dots \times s_n \in \Pi$

stating that \equiv is a strict partial congruence and partial operations and predicates are strict.

A Σ -axiom φ (in PFOL) is translated to the $\Phi(\Sigma)$ -axiom $\alpha_\Sigma(\varphi)$:

- $\alpha_\Sigma(t_1 \stackrel{e}{=} t_2) = t_1 \equiv_s t_2$ for $t_1, t_2 \in T_\Sigma(X)_s$
- $\alpha_\Sigma(p(t_1, \dots, t_n)) = p(t_1, \dots, t_n)$
- $\alpha_\Sigma(F) = F$
- $\alpha_\Sigma(\varphi \wedge \psi) = \alpha_\Sigma(\varphi) \wedge \alpha_\Sigma(\psi)$
- $\alpha_\Sigma(\varphi \Rightarrow \psi) = \alpha_\Sigma(\varphi) \Rightarrow \alpha_\Sigma(\psi)$
- $\alpha_\Sigma(\forall Y. \bigwedge_{\{y \in Y, s \in S\}} y \equiv_s y) \Rightarrow \alpha_\Sigma(\varphi)$

A $\langle \Phi(\Sigma), C(\Sigma) \rangle$ -structure B (in FOL) is translated to the partial Σ -structure $\beta_\Sigma(B)$ (in PFOL) with

$$\beta_\Sigma(B) = (B|_{\Sigma \rightarrow \Phi(\Sigma)}) / \equiv_B$$

where $B|_{\Sigma \rightarrow \Phi(\Sigma)}$ is the reduct of B along the obvious map $\Sigma \rightarrow \Phi(\Sigma)$, i.e. B interpreted as partial Σ -structure, and the quotient is taken as in Definition 3.33.

This translation now has the following crucial properties.

Proposition 3.66. *For each Σ -axiom φ and each total valuation $v: X \rightarrow B$ into a $\langle \Phi(\Sigma), C(\Sigma) \rangle$ -structure B we have*

$$\text{nat} \equiv_B \circ v \Vdash_{\Sigma}^{\text{PFOL}} \varphi \text{ if and only if } v \Vdash_{\Phi(\Sigma)}^{\text{FOL}} \alpha_{\Sigma}(\varphi)$$

Proposition 3.67. *For each partial Σ -structure A there is a $\langle \Phi(\Sigma), C(\Sigma) \rangle$ -structure B with $\beta_{\Sigma}(B) = A$, such that for each valuation $\rho: X \rightarrow A$ there exists a total valuation $v: X \rightarrow B$ with $\rho = \text{nat} \equiv_B \circ v$. If ρ moreover is total as well, then $v(x) \equiv_{s,B} v(x)$ for all $x \in X_s, s \in S$.*

Theorem 3.68 (Borrowing of proof calculus from FOL for PFOL).

$$\Phi \models_{\Sigma}^t X.\varphi \text{ iff } C(\Sigma) \cup \alpha_{\Sigma}(\Phi) \cup \{x \equiv_s x \mid x : s \in X \cup \bigcup_{Y: \psi \in \Phi} Y\} \vdash_{\Phi(\Sigma)}^{\text{FOL}} \alpha_{\Sigma}(\varphi)$$

$$\Phi \models_{\Sigma}^p \varphi \text{ iff } C(\Sigma) \cup \alpha_{\Sigma}(\Phi) \vdash_{\Phi(\Sigma)}^{\text{FOL}} \alpha_{\Sigma}(\varphi)$$

The translation of PFOL to FOL can also take advantage of special theorem prover for FOL coping also with the partial congruences very well. This translation generates partial congruence relations, which can be treated in a way similar to equality with the results of Bachmair and Ganzinger [BG94, CGW95].

3.3.4 Conditional logic with existential premises

Although having full first-order logic at hand to describe a specification allows in many cases to give a concise and close to the intuition axiomatization, there are several data types that are quite easily and naturally described within a far smaller fragment of PFOL, consisting of the conditional axioms.

The advantages in using such restricted language are basically two: on one side, if the form of the axioms used in the deduction is restricted, better theorem provers are available, taking advantage, for instance, of paramodulation (see [Pad88]) and conditional term-rewriting techniques (see Chapter 1 of this book by H. Kirchner and [DJ90, Klo92]).

On the semantic side, the existence of an initial model for such classes of specifications is guaranteed. Moreover, the initial model is characterized as the *minimal* first-order structure satisfying the axioms of the specification. Thus, using it corresponds to an economy of thought.

Let us first see an example of partial conditional specifications, before their formal definition and the proof of existence of initial (free) models for such a class of specifications.

Example 3.69. Let us see how, using the partial framework, the specification of *stacks* with their constructors and selectors becomes easy and elegant, indeed.

```

sig  $\Sigma_{\text{Stack}}$  = enrich  $\Sigma_{\text{Elem}}$  by
  sorts   stack
  opns   empty:  $\rightarrow$  stack
          push: elem  $\times$  stack  $\rightarrow$  stack
  popns  pop: stack  $\rightarrow$  stack
          top: stack  $\rightarrow$  elem

```

Then the minimal specification of stacks on this signature, is given by the axioms identifying the `pop` and `top` as (partial) inverse of the constructor `push`

```

spec Stack = enrich  $\Sigma_{\text{Stack}}$  by
  vars   e : elem; sstack
  axioms pop(push(e, s)) $\stackrel{s}{=}s$ 
          top(push(e, s)) $\stackrel{s}{=}e$ 

```

As we will see, all *positive conditional* specifications, i.e. specifications with axioms that are implications whose premises are (first-order equivalent to) a set of existential equalities and predicate applications, have an initial model, characterized by the *no-junk* & *no-confusion* properties. Therefore, in particular, the above specification of stacks have the following initial model I .

```

algebra I =
  Carriers
    |I|elem = X
    |I|stack = X*
  Functions
    emptyI =  $\lambda$ 
    pushI(x, s) = x · s
    popI(s) =  $\begin{cases} s', & \text{if } s = x \cdot s' \\ \text{undefined}, & \text{otherwise} \end{cases}$ 
    topI(s) =  $\begin{cases} x, & \text{if } s = x \cdot s' \\ \text{undefined}, & \text{otherwise} \end{cases}$ 

```

It is interesting to note that the specification `Stack` is the most abstract interpretation of stacks and can be further specialized to get an *implementation* where more details have been fixed. For instance the operation `pop` on the empty stack could be recovered on the empty stack, as in many standard total approaches, by enriching `Stack` with the following axiom

$$\text{pop}(\text{empty}) \stackrel{s}{=} \text{empty}$$

Since also this axiom is positive conditional, the enriched specification has an initial model too, that is the Σ_{Stack} -structure I with the interpretation of function `pop` modified into

$$\text{pop}_I = \begin{cases} s', & \text{if } s = x \cdot s' \\ \lambda, & \text{otherwise} \end{cases}$$

More refined error recovery (or detection) techniques can be implemented as well, by differently enriching `Stack`.

But in any case the (initial) models of the enrichments are already models of the specification **Stack** and hence any property proved for **Stack** holds for them too, allowing incremental tests. ■

Definition 3.70. A *positive conditional* formula is a well-formed first-order formula $\varphi \in \text{Form}(\Sigma, X)$ of the form

$$\varphi = \forall X. \epsilon_1 \wedge \cdots \wedge \epsilon_n \Rightarrow \epsilon$$

where ϵ is any atom and each ϵ_i is either a predicate application or an existential equality.

A specification Sp is called *positive conditional* if it has the same model class as a specification $Sp' = \langle \Sigma, AX \rangle$ and each $\varphi \in AX$ is a positive conditional formula. ◇

A particular case of partial positive conditional specifications are total conditional specifications. Indeed, a total first-order signature is a partial first-order signature with the empty family of partial function symbols $\text{popns}(\Sigma) = \emptyset$ and, moreover, each partial first-order structure is a total first-order structure too. Thus, the distinction among different kind of equalities is immaterial and hence the model class of a total conditional specification is the same as the model class of the partial positive conditional specification having the “same” axioms, where each $=$ symbol has been replaced by $\stackrel{e}{=}$.

Another important class of positive conditional specifications that can be easily recognized are those whose axioms are conditional and each strong equality in the premises is guarded by a definedness assertion on either side of the equality and, analogously, each weak equality in the premises is guarded by a definedness assertion on both sides of the equality, because such a formula has the same models as the given conditional formula where all equalities in the premises have been substituted by existential equalities.

Exercise 3.71. Show that a specification $Sp = \langle \Sigma, AX \rangle$ is positive conditional if each $\varphi \in AX$ has the form

$$\varphi = \forall X. \epsilon_1 \wedge \cdots \wedge \epsilon_n \Rightarrow \epsilon$$

where ϵ and all ϵ_i are atoms and the following two conditions are satisfied:

- if ϵ_i is the strong equality $t \stackrel{s}{=} t'$, then there exists $j \in 1, \dots, n$ s.t. ϵ_j is $D(t)$ or $D(t')$;
- if ϵ_i is the weak equality $t \stackrel{w}{=} t'$, then there exist $j, k \in 1, \dots, n$ s.t. ϵ_j is $D(t)$ and ϵ_k is $D(t')$.

Using initial semantics of specifications with positive conditional formulae to describe a data type intuitively corresponds to using inductive definition. A particular, but very common, case is the axiomatization of a data type where the carriers are built by some total functions, called *constructors*, and

then other, possibly partial, operations are defined on such elements simply imposing the equality of their applications to terms built by the constructors.

An instance of this methodology of data definition is, indeed, the previous example of the stacks, that are built by pushing elements on the empty stack, where the evaluation of a `pop (top)` reduces by the axioms to the evaluation of simpler terms, without `pop (top)`. Let us see another example, that is the specification of the minus between non-negative integers.

Example 3.72. The basic specification of non-negative integers is the usual (total) one, given by the absolutely free constructors “zero” and “successor”.

```
sig SpNatT =
  sorts  nat
  opns   zero: → nat
         succ: nat → nat
```

Then on this signature we want to define, for example, the predecessor and the minus operations.

```
spec SpNat = enrich SpNatT by
  popns prec: nat →→ nat
        minus: nat × nat →→ nat
  vars  x, y: nat
  axioms prec(succ(x))  $\stackrel{s}{=} x$ 
         minus(x, zero)  $\stackrel{s}{=} x$ 
         minus(succ(x), succ(y))  $\stackrel{s}{=} \text{minus}(x, y)$ 
```

The above specification follows the intuition that the new operations are *programs* on a data type built by zero and successor, inductively defined by means of the constructors. Indeed, a term starting with a `prec` or a `minus` symbol can be deduced defined iff it reduces to a term of the form `succk(zero)`, because the axioms are strong equalities. Thus, for instance, `prec(zero)` cannot be deduced to be defined (and indeed in the initial model, it is undefined) and represents an erroneous *call* of `prec`.

It is also worth noting that the given axioms allow the intuitively intended identifications for minimal models, that are those partial first-order structures where each element of the carrier is denoted by a term of the form `succk(zero)`. ■

The existence of initial (free) models for positive conditional axioms is due to the particular structure of the model class, as in the total (both equational and conditional) case.

Roughly speaking the first step to prove the existence of an initial (free) model I is to show that if it exists, then it is term-generated. This property is due to the fact that the model class of a positive conditional specification is closed under closed substructure, so that the term-generated part of I is a model too and hence, as the initial (free) model is in a sense the *smallest*

(w.r.t. the partial order induced by homomorphism existence), I and its term-generated part must coincide.

Thus, I is (isomorphic to) a term-algebra quotient. Moreover, since homomorphisms preserve existential equalities and predicate assertions, the congruence defining I must be *minimal*, i.e. it must be the intersection of all the kernels of term-evaluation in a model of the specification.

The last step is to prove that the quotient of the term algebra w.r.t. the intersection of all the kernels of term-evaluation in a model of the specification is actually a model too, i.e. that it satisfies the axioms. This point too relies on the form of the axioms. Indeed, if the premises of an axiom hold in such a quotient, then they must hold in each model. Hence the consequence too holds in all models, so that it holds in the quotient, that is, therefore, a model.

Theorem 3.73. *Let Sp be a positive conditional specification over a partial first-order signature $\Sigma = \langle S, \Omega, P\Omega, \Pi \rangle$ and X be an S -sorted family of variables. Then there is a free Sp -model for X , that is (isomorphic to) the quotient F of the term algebra $T_\Sigma(X)$ with the following interpretation of predicate symbols:*

$(t_1, \dots, t_n) \in p_{T_\Sigma(X)}$ iff $A \models \forall X.p(t_1, \dots, t_n)$ for all models A of Sp .

by the following congruence \equiv :

$t \equiv t'$ iff $A \models \forall X.t \stackrel{e}{=} t'$ for all models A of Sp .

We also can use the translation from PFOL to FOL to get initial models in the partial conditional case.

Proposition 3.74. *Let $\langle \Sigma, AX \rangle$ be a presentation and I an initial model in $Mod(\langle \Phi(\Sigma), C(\Sigma) \cup \alpha(AX) \rangle)$. Then $\beta(I)$ is an initial model in $Mod(\langle \Sigma, AX \rangle)$.*

A sound and practically complete calculus for the conditional fragment of partial first-order logic for \models^t consists of the rules **Assumption**, **Cut**, **\forall -elimination**, **Reflexivity**, **Congruence**, **Substitution**, **Function Strictness**, **Predicate Strictness** and **Totality**. For \models^p , it consists of the same rules modified for \models^p , except that **Reflexivity** is replaced by **Symmetry**.

But like full partial first-order logic, also positive conditional specifications are reducible, from a deductive point of view, to the usual total first-order specifications which turn out to be conditional again, so that automatic tools and techniques developed for the conditional total case can be *borrowed* for the partial as well.

The key point of such a reduction technique is the translation of a positive conditional specification into a corresponding total conditional specification, whose models satisfy the same atomic formulae (up to translation). This is a particular case of the *borrowing* technique proposed in [CM97] and a sugared version of the borrowing for PFOL, where also definedness predicates are allowed.

Definition 3.75. Let $\Sigma = \langle S, \Omega, P\Omega, \Pi \rangle$ be a partial signature.

- Let Σ^T denote the total first-order signature

$$\langle S, \Omega \cup P\Omega, \Pi \cup \{D_s : s, \overset{e}{=}_s : s \times s\}_{s \in S} \rangle$$

and AX^T denote the following set of total conditional formulae on Σ^T :

$$\begin{aligned} & D_{s_1}(x_1) \wedge \cdots \wedge D_{s_n}(x_n) \Rightarrow D_s(f(x_1, \dots, x_n)) \\ & \quad \text{for all } f : s_1 \times \cdots \times s_n \longrightarrow s \in \Omega \\ & D_s(f(x_1, \dots, x_n)) \Rightarrow D_{s_i}(x_i) \\ & \quad \text{for all } f : s_1 \times \cdots \times s_n \longrightarrow s \in \Omega \cup P\Omega \\ & x \overset{e}{=}_s y \Rightarrow D_s(x) \\ & D_s(x) \Rightarrow x \overset{e}{=}_s x \\ & D_s(x) \wedge x = y \Rightarrow x \overset{e}{=}_s y \\ & x \overset{e}{=}_s y \Rightarrow x = y \\ & x \overset{e}{=}_s y \Rightarrow y \overset{e}{=}_s x \\ & x \overset{e}{=}_s y \wedge y \overset{e}{=}_s z \Rightarrow x \overset{e}{=}_s z \end{aligned}$$

- Let us call *strictly positive conditional* a formula over a finite set of variables X having the form $\forall X. \epsilon_1 \wedge \cdots \wedge \epsilon_n \Rightarrow \epsilon_{n+1}$ with each ϵ_i a predicate application, or an existential equality, or a definedness assertion. For each strictly positive conditional formula $\varphi = \forall X. \epsilon_1 \wedge \cdots \wedge \epsilon_n \Rightarrow \epsilon_{n+1}$, let $\alpha(\varphi)$ denote the following total conditional axiom:

$$\bigwedge_{x \in X_s} D_s(x) \wedge \epsilon_1 \wedge \cdots \wedge \epsilon_n \Rightarrow \epsilon_{n+1}$$

- For each total first-order structure A modeling $\langle \Sigma^T, AX^T \rangle$, let $\beta(A)$ denote the following partial first-order structure B :
 - $|B|_s = D_{sA}$ for all $s \in S$.
 - f_B is the restriction of f_A to the carriers of B for all $f : s_1 \times \cdots \times s_n \longrightarrow s \in \Omega$.
 - pf_B is the restriction of pf_A to the carriers of B for all $pf : s_1 \times \cdots \times s_n \dashrightarrow s \in P\Omega$; in particular if $(a_1, \dots, a_n) \in D_{s_1A} \times \cdots \times D_{s_nA}$ but $pf_A(a_1, \dots, a_n) \notin D_{sA}$, then $pf_B(a_1, \dots, a_n)$ is undefined.
 - p_B is the restriction of p_A to the carriers of B for all $p : s_1 \times \cdots \times s_n \in \Pi$.

◇

Thus, each total first-order structure satisfying AX^T corresponds to the partial first-order structure where the “undefined” elements have been left out and this correspondence reflects on the logic too, in the sense that the reduction of a total to a partial first-order structure satisfies the same strictly positive conditional formulae (up to the α translation).

Lemma 3.76. *Using the notation of Definition 3.75, the following satisfaction condition holds for all total first-order structure A and all strictly positive conditional formulae φ :*

$$\beta(A) \models^t \varphi \Leftrightarrow A \models \alpha(\varphi)$$

Exercise 3.77. Show that each positive conditional specification has the same model class as a specification whose axioms are all strictly positive conditional formulae.

Therefore, for each partial positive conditional specification $Sp = \langle \Sigma, AX \rangle$ the model class of Sp satisfies a strictly positive conditional formula iff the model class of the total conditional specification $Sp^T = (\Sigma^T, \alpha(AX) \cup AX^T)$ satisfies its translation along α . Hence each deductive system (theorem prover) for total conditional specification can be used to verify the validity of strictly positive conditional formulae in the model classes of partial positive conditional specification. Moreover, this result can be extended to any class of formulae that can be effectively translated into strictly positive conditional form without affecting their validity. Indeed, in this case the validity verification splits in

- a preliminary coding of the formula into strictly positive conditional,
- a translation of this form into a total conditional formula via α ,
- an application of any (conditionally complete) deduction system for Sp^T .

Theorem 3.78. *Let $Sp = \langle \Sigma, AX \rangle$ be a partial positive conditional specification (with axioms all in strictly positive conditional form) and φ be a strictly positive conditional formula.*

Using the notation of Definition 3.75, $\text{Mod}_\Sigma(AX) \models^t \varphi$ iff $\alpha(AX) \cup AX^T \vdash \alpha(\varphi)$, where \vdash is given in Definition 3.8.

Example 3.79. Let us consider again the problem of store specification already presented in Example 3.16.

Here, as natural in a context having predicates, we assume that the specification of locations defines also some predicates, instead of their implementations as Boolean functions as in Example 3.16. Using the predicates **AreEqual** and **AreDifferent** to check the equality between locations instead than any of the predefined equalities of our logic, allows a greater freedom. Indeed, the user can axiomatize those predicates in a way that their interpretation in some model is not the identity.

Notice that, since the (assertion of the) negation of the equality between locations is needed in the premises of some axioms, then also its negation has to be axiomatized as a (different) predicate, only because we want to get a *positive conditional* specification. Otherwise, using a more expressive fragment of partial first-order logic, we could have just the predicate **AreEqual**.

Therefore, let us assume that the specification Sp_L of locations includes the following:

```

spec  $Sp_L =$ 
  sorts   loc ...
  preds  AreEqual, AreDifferent: loc  $\times$  loc ...

```

Then we can enrich Sp_L and the specification Sp_V of values, with main sort **value** to get the store specification.

```

spec Stores = enrich  $Sp_L, Sp_V$  by
  sorts   store
  opns    empty:  $\rightarrow$  store
          update: store  $\times$  loc  $\times$  value  $\rightarrow$  store
  popns   retrieve: store  $\times$  loc  $\rightarrow$  value
  vars     $x, y : \text{loc}; v, v_1, v_2 : \text{value}; s : \text{store}$ 
  axioms  AreEqual( $x, y$ )  $\Rightarrow$  update( $s, x, v$ )  $\stackrel{s}{=}$  update( $s, y, v$ )
          AreEqual( $x, y$ )  $\Rightarrow$ 
            update(update( $s, x, v_1$ ),  $y, v_2$ )  $\stackrel{s}{=}$  update( $s, x, v_2$ )
          AreDifferent( $x, y$ )  $\Rightarrow$  update(update( $s, x, v_1$ ),  $y, v_2$ )  $\stackrel{s}{=}$ 
            update(update( $s, y, v_2$ ),  $x, v_1$ )
          AreEqual( $x, y$ )  $\Rightarrow$  retrieve(update( $s, x, v$ ),  $y$ )  $\stackrel{s}{=} v$ 

```

Notice that, thanks to the partial setting, in **Stores** initial model the application $\text{retrieve}(s, x)$ to stores s where x has never been updated, for instance if s is **empty**, does not yield any value, because it cannot be deduced defined, and hence the problem on hierarchical consistency seen in the total case does not apply here.

In [Rei87], a much more sophisticated specification for stores, where for instance it is possible to remove an association from a store, is given in a different setting. The reader is encouraged to rephrase it using positive conditional data types.

The inference system for **Stores** is the total conditional one, for the specification

```

spec StoresT = enrich  $Sp_L^T, Sp_V^T$  by
  sorts   store
  opns    empty:  $\rightarrow$  store
          update: store  $\times$  loc  $\times$  value  $\rightarrow$  store
          retrieve: store  $\times$  loc  $\rightarrow$  value
  vars     $x, y : \text{loc}; v, v_1, v_2 : \text{value}; s : \text{store}$ 
  axioms   $D(\text{empty})$ 
           $D(s) \wedge D(x) \wedge D(v) \Rightarrow D(\text{update}(s, x, v))$ 
           $D(\text{update}(s, x, v)) \Rightarrow D(s)$ 
           $D(\text{update}(s, x, v)) \Rightarrow D(x)$ 
           $D(\text{update}(s, x, v)) \Rightarrow D(v)$ 
           $D(\text{retrieve}(s, x)) \Rightarrow D(s)$ 
           $D(\text{retrieve}(s, x)) \Rightarrow D(x)$ 
           $D(x) \wedge D(y) \wedge D(s) \wedge D(v) \wedge \text{AreEqual}(x, y) \Rightarrow$ 
            update( $s, x, v$ ) = update( $s, y, v$ )
           $D(x) \wedge D(y) \wedge D(s) \wedge D(v_1) \wedge D(v_2) \wedge \text{AreEqual}(x, y) \Rightarrow$ 
            update(update( $s, x, v_1$ ),  $y, v_2$ ) = update( $s, x, v_2$ )

```

$$\begin{aligned}
D(x) \wedge D(y) \wedge D(s) \wedge D(v_1) \wedge D(v_2) \wedge \text{AreDifferent}(x, y) &\Rightarrow \\
&\text{update}(\text{update}(s, x, v_1), y, v_2) = \\
&\text{update}(\text{update}(s, y, v_2), x, v_1) \\
D(x) \wedge D(s) \wedge D(v) \wedge \text{AreEqual}(x, y) &\Rightarrow \\
&\text{retrieve}(\text{update}(s, x, v), y) = v
\end{aligned}$$

where Sp_L^T and Sp_V^T are the corresponding translations where for example axioms like $\text{AreDifferent}(x, y) \Rightarrow D(x)$ have being added. ■

Another case where partial specifications come in hand is the specification of *bounded* data types, where the *constructors* themselves are partial functions. For instance let us consider the specification of bounded stacks, parametric on a positive constant *max* representing the maximum number of element that can be stacked.

Example 3.80.

```

spec BoundedStacks = enrich Nat_{≤}, Σ_{Elem} by
  sorts   bstack
  opns   empty : → bstack
         max : → nat
  popns  Bpush : elem × bstack →→ bstack
         pop : bstack →→ bstack
         top : bstack →→ elem
  vars   x : elem, s : bstack
  axioms depth(empty) = zero
         succ(depth(s)) ≤ max ⇒ depth(Bpush(x, s)) = succ(depth(s))
         D(Bpush(x, s)) ⇔ succ(depth(s)) ≤ max
         D(Bpush(x, s)) ⇒ pop(Bpush(x, s))  $\stackrel{e}{=} s$ 
         D(Bpush(x, s)) ⇒ top(Bpush(x, s))  $\stackrel{e}{=} x$ 

```

Let us finally see a motivating example of a partial recursive function with non-recursive domain (that cannot, hence, be described by a total specification identify all “erroneous applications”). It is (a fragment of) the definition of the semantics for an imperative language based on environments and states. Here we try to capture the key-points of this complex example, leaving the details for the interested reader to fill in. In particular we are not considering the declarative part of the language nor the environment aspects. Thus, commands can be simply represented as functions from states to states (and expressions as functions from states to values).

Example 3.81. Let us assume given the specification of the states; then commands are *partial* functions among them. It is worth noting that we are not interested in deducing the extensional equality between commands, because not only we do not need it in order to describe the language semantics, but it is also too restrictive to capture for instance complexity criteria, that are interesting for imperative language semantics. Therefore, we do not really

need (a representation of) partial higher-order and can, hence, restrict ourselves to positive conditional types, while extensionality implicitly requires a more powerful and technically more complex fragment of the logic (see e.g. [AC92,AC95] for an extended treatment or Section 3.4 of this chapter for an introductory discussion of the subject).

Here we are more interested in *command constructs* than in basic commands like `skip`, `read` and `write`, that are more relevant to the specification of states and the language data types than to proper command part. In particular we are interested in the `while_do` command, as source of possible non-termination and hence as paramount example of partial recursive function with non-recursive domain.

Therefore, we also assume given the specification of a `BExp` data type, where the Boolean expressions of our imperative language should be interpreted, with the obvious constants, axiomatized by the following specification. The actual constructs for the Boolean expressions are omitted, as immaterial. The relevant part of Boolean expressions for the command specification is simply the fact that any such expression can be evaluated onto a state producing, possibly, a truth value.

```
spec BExp = enrich States by
  sorts   bool, BoolExps
  opns    true, false: → bool ...
  popns   BEval: BoolExps × state → bool ...
```

Let us finally see the specification for the kernel of the language semantics concerning the command constructs.

```
spec ImplLang = enrich States, BExp by
  sorts   commands
  opns    skip: → commands ...
          if_then_else: BoolExps × commands × commands → commands
          while_do: BoolExps × commands → commands
          conc: commands × commands → commands ...
  popns   CEval: commands × state → state ...
  vars    c, c': commands; s: state; b: BoolExps
  axioms  CEval(c, s)  $\stackrel{e}{=} s'$  ⇒ CEval(conc(c, c'), s)  $\stackrel{s}{=} CEval(c', s')$ 
          BEval(b, s)  $\stackrel{e}{=} true$  ⇒ CEval(if_then_else(b, c, c'), s)  $\stackrel{s}{=} CEval(c, s)$ 
          BEval(b, s)  $\stackrel{e}{=} false$  ⇒
            CEval(if_then_else(b, c, c'), s)  $\stackrel{s}{=} CEval(c', s)$ 
          BEval(b, s)  $\stackrel{e}{=} false$  ⇒ CEval(while_do(b, c), s)  $\stackrel{e}{=} s$ 
          BEval(b, s)  $\stackrel{e}{=} true$  ∧ CEval(c, s)  $\stackrel{e}{=} s'$  ⇒
            CEval(while_do(b, c), s)  $\stackrel{s}{=} CEval(while_do(b, c), s')$ 
```

Here, as in programming languages, the conditional choice `if_then_else` is *non-strict*, in the sense that `CEval(if_then_else(b, c, c'), s)` can result in a value even if the evaluation of some its subterm in the same state does not. For instance if the Boolean expression yields `true`, then the evaluation of the second branch can be undefined. However the interpretation of all functions

of the signature are *strict*. Indeed, the non-strictness has been achieved by using *functions* instead than *ground elements* as interpretation for commands and expressions.

If *real* non-strictness is needed, as it is sometime the case for instance in the design phase, then partial logic is inadequate and more powerful frameworks should be applied (see e.g. Section 3.4 of this chapter for a discussion of the problem and references). ■

3.3.5 Subsorting in partial first-order logic

Though some applications of subsorting can be better represented using partiality and predicates, there are still important cases where data types are naturally supersets or subsets of other data types. For this reason and to support the many users of order-sorted specification languages, it may be convenient to enrich the language used to describe specifications by subsort declaration. This viewpoint, indeed, has been adopted by the language CASL¹⁹ in course of definition within the CoFI initiative (see e.g. [Mos97]) and the semantics of the subsorting constructs can be very easily described using the theory presented so far.

Indeed, the idea is to encode an order-sorted signature into a partial one, using explicit *embeddings* to represent the subsorting relationship. Then, formulae and models of partial logic are used for the subsorted framework as well. But, since in order-sorted formalisms overloading of function (predicate) symbols usually carries a semantics, in the sense that using the same name for functions on sorts connected by the subsort relation requires that the corresponding interpretations of the function are *compatible*, we have to restrict the models to those satisfying such property by means of a set of formulae. Therefore, we represent an order-sorted signature by a partial positive conditional presentation and borrow from it formulae and models.

The notion of subsorted signatures proposed here extends the notion of order-sorted signatures as given by Goguen and Meseguer [GM92], by allowing not only total function symbols, but also partial function symbols and predicate symbols. The resulting logic is called *subsorted partial first-order logic* (*SubPFOL*).

Definition 3.82. A *subsorted signature* $\Sigma = (S, \Omega, P\Omega, \Pi, \leq_S)$ consists of a partial first-sorted signature $(S, \Omega, P\Omega, \Pi)$ together with a reflexive transitive *subsорт relation* \leq_S on the set S of sorts. The relation \leq_S extends pointwise to sequences of sorts.

For a subsorted signature, $\Sigma = (S, \Omega, P\Omega, \Pi, \leq_S)$, the *overloading relations* \sim_Ω and \sim_P , for function and predicate symbols, are defined by:

¹⁹ See <http://www.brics.dk/Projects/CoFI/DesignProposals/Summary> for a summary of the CASL language.

- $f : w_1 \rightarrow_1 s_1 \sim_\Omega f : w_2 \rightarrow_2 s_2$ iff there exist $w \in S^*$ s.t. $w \leq w_1, w_2$ and a common supersort of s_1 and s_2 ; for each $f : w_1 \rightarrow_1 s_1, f : w_2 \rightarrow_2 s_2 \in \Omega \cup P\Omega$ (where $\rightarrow_1, \rightarrow_2 \in \{\rightarrow, \rightarrow\}$);
- $p : w_1 \sim_P p : w_2$ iff there exists $w \in S^*$ s.t. $w \leq w_1, w_2$; for each $p : w_1, p : w_2 \in \Pi$.

A *signature morphism* $\sigma : \Sigma \rightarrow \Sigma'$ is a many-sorted signature morphism that preserves the relations \leq_S, \sim_Ω and \sim_P . \diamond

Note that two profiles of an overloaded constant declared with two different sorts are in the overloading relation iff the two sorts have a common supersort.

In the following, due to the relevance of name clashes for functions and predicates, we will use $f_{\langle w, s \rangle}$ instead of f for a (total or partial) function symbol in term formation, in order to explicitly disambiguate terms, if needed.

Definition 3.83. With each subsorted signature $\Sigma = (S, \Omega, P\Omega, \Pi, \leq_S)$ we associate a many-sorted signature $\Sigma^\#$, which is the extension of the underlying many-sorted signature $(S, \Omega, P\Omega, \Pi)$ with

1. a total *injection* function symbol $\text{inj} : s \rightarrow s'$, for each $s \leq_S s'$;
2. a partial *projection* function symbol $\text{pr} : s' \rightarrow s$, for each $s \leq_S s'$;
3. and a unary *membership* predicate symbol $\in^s : s'$, for each $s \leq_S s'$.

We assume that the symbols used for injections, projections and membership are not used otherwise in Σ .

Subsorted Σ -sentences are ordinary many-sorted $\Sigma^\#$ -sentences.

Subsorted Σ -models are ordinary many-sorted $\Sigma^\#$ -models satisfying the following set of axioms $J(\Sigma)$:

identity $\forall x : s. \text{inj}_{\langle s, s \rangle}(x) \stackrel{e}{=} x$;

embedding-injectivity $\forall x, y : s. \text{inj}_{\langle s, s' \rangle}(x) \stackrel{e}{=} \text{inj}_{\langle s, s' \rangle}(y) \Rightarrow x \stackrel{e}{=} y$ for $s \leq_S s'$;

transitivity $\forall x : s. \text{inj}_{\langle s', s'' \rangle}(\text{inj}_{\langle s, s' \rangle}(x)) \stackrel{e}{=} \text{inj}_{\langle s, s'' \rangle}(x)$ for $s \leq_S s' \leq_S s''$;

projection $\forall x : s. \text{pr}_{\langle s', s \rangle}(\text{inj}_{\langle s, s' \rangle}(x)) \stackrel{s}{=} x$ for $s \leq_S s'$;

projection-injectivity $\forall x, y : s. \text{pr}_{\langle s', s \rangle}(x) \stackrel{e}{=} \text{pr}_{\langle s', s \rangle}(y) \Rightarrow x \stackrel{e}{=} y$ for $s \leq_S s'$;

membership $\forall x : s'. (\in^s_s x \Leftrightarrow D_s(\text{pr}_{\langle s', s \rangle}(x)))$ for $s \leq_S s'$;

function monotonicity

$$\begin{aligned} & \forall x_1 : s_1 \dots x_n : s_n. \\ & \text{inj}_{\langle s^1, s \rangle}(f_{\langle w_1, s^1 \rangle}(\text{inj}_{\langle s_1, s^1 \rangle}(x_1), \dots, \text{inj}_{\langle s_n, s^1 \rangle}(x_n))) \stackrel{s}{=} \\ & \text{inj}_{\langle s^2, s \rangle}(f_{\langle w_2, s^2 \rangle}(\text{inj}_{\langle s_1, s^2 \rangle}(x_1), \dots, \text{inj}_{\langle s_n, s^2 \rangle}(x_n))) \end{aligned}$$

for $f : w_1 \rightarrow_1 s^1 \sim_\Omega f : w_2 \rightarrow_2 s^1$, where $w_1 = s_1^1 \times \dots \times s_n^1$, $w_2 = s_1^2 \times \dots \times s_n^2$, $w = s_1 \times \dots \times s_n$ with $w \leq w_1, w_2$ and $s^1, s^2 \leq s$;

predicate monotonicity

$$\forall x_1 : s_1 \dots x_n : s_n. p_{w_1}(\text{inj}_{\langle s_1, s_1^1 \rangle}(x_1), \dots, \text{inj}_{\langle s_n, s_n^1 \rangle}(x_n)) \Leftrightarrow p_{w_2}(\text{inj}_{\langle s_1, s_1^2 \rangle}(x_1), \dots, \text{inj}_{\langle s_n, s_n^2 \rangle}(x_n))$$

for $p : w_1 \sim_P p : w_2$, where $w_1 = s_1^1 \times \dots \times s_n^1$, $w_2 = s_1^2 \times \dots \times s_n^2$ and $w = s_1 \times \dots \times s_n$ with $w \leq w_1, w_2$;

Σ -homomorphisms are $\Sigma^\#$ -homomorphisms. \diamond

Let us see a simple application of the subsorting formalism proposed here.

Example 3.84. Let us extend the natural number specification to the (positive) rational numbers.

```

spec  $Sp_{\text{rat}}$  = enrich  $\Sigma_{\text{Nat}}$  by
  sorts    $\text{nat} \leq \text{rat}$ 
  opns    $\text{plus}, \text{times} \dots : \text{rat} \times \text{rat} \rightarrow \text{rat}$ 
  popns   $\_/\_ : \text{nat} \times \text{nat} \rightarrow \text{rat}$ 
  axioms  $\forall x : \text{nat}. \text{inj}_{\langle \text{nat}, \text{rat} \rangle}(x) \stackrel{e}{=} x / \text{succ}(\text{zero})$ 
            $\forall x, y : \text{nat}. D(x/y) \Rightarrow D(x / \text{succ}(y))$ 
            $\forall x_1, y_1, x_2, y_2 : \text{nat}. D(x_1/y_1) \wedge D(x_2/y_2) \wedge$ 
              $\text{times}(x_1, y_2) \stackrel{e}{=} \text{times}(x_2, y_1) \Rightarrow x_1/y_1 \stackrel{e}{=} x_2/y_2$ 
            $\forall x_1, y_1, x_2, y_2 : \text{nat}. \text{plus}(x_1/y_1, x_2/y_2) \stackrel{s}{=}$ 
              $\text{plus}(\text{times}(x_1, y_2), \text{times}(x_2, y_1)) / \text{times}(y_1, y_2) \dots$ 

```

The models of Sp_{rat} are the expected ones, where the carrier of sort nat is isomorphic to a subset of rat , allowing also implementations where for efficiency reasons natural numbers are differently represented w.r.t. rational numbers, and the first and third axioms identify multiple representations of the same number. \blacksquare

In standard order-sorted approaches, signatures are required to satisfy conditions in order to get that the usual functional notation for terms is semantically unambiguous, without requiring disambiguation of multiply defined functions and explicit injections as we require here. Indeed, notice that in the first axiom of the above example, an explicit injection has been introduced to get a well-formed equation. Thus, in our approach we gain in simplicity for the theory, because we do not have to impose regularity, monotonicity and similar conditions on signatures that hinder modular constructions of complex data types. But we pay for this, because the language of sentences is much less user-friendly than in standard order-sorted approaches. However, it is possible to eat the cake and have it too, by introducing a more liberal language in order to describe the axioms and then parse its expressions yielding unambiguous formulae in the standard algebraic style.

This two-level approach has been adopted in CASL, the CoFI language, and is based on the idea that terms and axioms at the user level are liberally built, as in standard order-sorted approaches, by regarding terms of

the subsort as terms of the supersort as well (while projections/retracts have to be explicitly stated) and using function symbols with or without their qualification. Then, such terms are expanded in all possible ways to terms in a standard fully qualified functional style. Finally, a term is successfully parsed if all its expansions are deducible equivalent from the axioms in Definition 3.83, otherwise it is rejected and the user is asked to disambiguate it.

The details may be found in [KKM98,Mos98,CHKM97].

A calculus for *SubPFOL* can be obtained from the calculus for \models^t by adding the axioms $J(\Sigma)$ as axioms to the calculus. If the model theory is restricted to models with non-empty carriers (as it is done in CASL), we also have to drop the variable subscripts from the derivability relation \vdash .

A different version of subsorted partial logic, which has a more restrictive treatment of subsorting, but can also deal with higher-order functions, is developed in [Far93].

Bibliographical notes. Full many-sorted and order-sorted first order logic was introduced by Oberschelp [Obe62]. A survey over the conditional fragment of order-sorted logic can be found in [GM92]. Many-sorted logic is studied extensively in [MT93]. The empty carrier problem, which is ignored by most authors, is treated by Hailperin [Hai53] for the first-order case (the work on so-called free logics is also related to this, see [Lam91] and [KGS88], Chapter 9.2) and by Goguen and Meseguer [GM85] for the many-sorted case.

The two-valued partial first-order logic *of definedness* (using \models^t) was introduced by Burmeister [Bur82,Bur86] and Beeson [Bee85] and generalized to categorical logic by Knijnenburg and Nordemann [KN94]. The partial first-order logic *of existence* (using \models^p) follows ideas of Scott [Sco79], see also [Mog88]. \models^t and \models^p are compared by Beeson in [Bee86], see also [Fef95].

There is a calculus and a Henkin-style completeness theorem for partial higher order logic in [Far91], Burmeister has a calculus for one-sorted partial logic [Bur82], The translation from partial to total first-order logic is described by Scott [Sco79]. This translation generates partial congruence relations, which can be treated in a way similar to equality with the results of Bachmair and Ganzinger [BG94]. There have been two workshops on theorem proving with partial functions.²⁰ The restriction to the positive conditional case is studied by Reichel and others in [Rei87,BR83,AC95,BW82,Cer93,Mos96,Tar85].

3.4 More advanced problems

Although the positive conditional fragment of partial first-order logic is powerful enough for most data type specifications, there are a few cases where it

²⁰ See <http://www.cs.bham.ac.uk/~mmk/partiality/>.

is insufficient to directly represent the intended data type, or where even full partial first-order logic is too poor. In the following paragraphs we will see some of the most relevant and common problems.

Partial higher-order specifications. As we have noticed before, higher-order partial data types are quite common in programming languages, for instance to represent environment and stores in the imperative paradigm, or to describe functional (or procedural) parameters.

Since their use is reasonably restricted, it is not necessary to have *real* higher-order logic, but it is sufficient to consider a particular case of first-order specifications. The main intuition is that the set of sort is not unstructured, but is constructed by a subset B of *basic* sorts and a (polymorphic) operation building the functional type. That is the set of sort is a subset of the set S^\rightarrow inductively defined by the following rules:

$$B \subseteq S^\rightarrow \quad s_1, \dots, s_n, s \in S^\rightarrow \Rightarrow (s_1 \times \dots \times s_n \rightarrow s) \in S^\rightarrow$$

Of course the sort $(s_1 \times \dots \times s_n \rightarrow s)$ represents the type of functions with arguments in the cartesian product $s_1 \times \dots \times s_n$ and result of sort s . Accordingly, in the signature an explicit *apply* operation, taking as input an element of a functional sort and the arguments for it and yielding the result of the application of the function to its input, is provided.

Therefore, the set S of sorts must be downward-closed, that is if $(s_1 \times \dots \times s_n \rightarrow s_{n+1}) \in S$, then all $s_i \in S$.

Since we are interested in the specification of partial functions, the application must be a partial function²¹, even if the other operations of the signature are total (consider, for example, a total function delivering partial functions as result). For instance in the previous example of the specification of a kernel of a programming language semantics, the only partial operation was the application of commands (respectively Boolean expressions) to their input, the current state, denoted by **CEval** (**BEval**), because all the command constructors were total functions.

The intuition that the elements of a functional sort $(s_1 \times \dots \times s_n \rightarrow s_{n+1}) \in S$ actually are (isomorphic to) functions, is expressed not only by the application function, but also by the *extensionality principle*, requiring that two functions yielding the same result on each possible input must be the same.

Although usually the specification of higher-order types for programming languages can be described within the positive conditional fragment, the axiomatization of the extensionality property requires a more powerful logic. Indeed, the natural form of the extensionality axiom (for simplicity in the

²¹ Only if we introduce two different function space constructors, one for total and one for partial functions, we can choose a total application for total function spaces.

case of unary functions) is

$$\star \quad \forall f, f' : (s \rightarrow s). (\forall x : s \text{ apply}(f, x) \stackrel{s}{=} \text{apply}(f', x)) \Rightarrow f \stackrel{e}{=} f'$$

Notice that the equality in the premises is strong, capturing the idea that f and f' should have the same definition domain and yield the same result on applications within their domain.

A particularly interesting case is that of term-generated models. In that case, indeed, the premise of the extensionality axiom is equivalent to the infinitary conjunction of all its possible instantiations on (defined) terms. Therefore, for term-generated models, extensionality can be reduced to an *infinitary* conditional axiom

$$\star^t \quad \bigwedge_{t \in T_{\Sigma_s}} \text{apply}(f, t) \stackrel{s}{=} \text{apply}(f', t) \Rightarrow f \stackrel{e}{=} f'$$

But notice that the equalities in the premises are not, nor can be substituted by, existential. Thus, even in this simplified case, partial higher-order do not reduce to positive conditional specifications. Indeed, in the general case even total equational specifications of partial higher-order algebras do not have an initial model in the class of all extensional models, that are the models satisfying the axiom \star , nor in the class of all term-extensional models, that are the models satisfying the axiom \star^t (see e.g. [AC92]).

It is worth noting that moving from a finitary to an infinitary logic, although obviously affecting computational issues for the involved deductive systems, does not change the nature of the problem. Indeed, the same results as for total conditional types are achieved in [Mei92, Möl87, MTW88] where the same problem is tackled for total types. Moreover, in a partial context, the same problems existing for term-extensional higher-order algebras, are already present for *strongly conditional* partial specifications, that are conditional specifications whose axioms admit (unguarded) strong equalities in their premises (see e.g. [AC95]).

Using the extensionality requirement as unique non-positive conditional axiom of a specification, it is possible to describe awkward data types, for instance with all non-trivial models having finite carriers with bounded cardinality. We demand to [AC92] for an analysis and exposition of the problem of higher-order partial types.

The theory sketched so far can be extended in several ways.

A first possible extension is to add λ -abstraction. Given a term t , the term

$$\lambda x : s. t$$

denotes an anonymous function f defined by

$$\forall x : s. f(x) \stackrel{s}{=} t$$

Thus, typed λ -abstraction works fine when combined with partiality [Far91]. But note that to be able to interpret λ -abstraction consistently, we have to

require that all λ -definable functions exist in our models. Such models are called *generalized Henkin models* [Hen50]. This requirement can be achieved adding a complete set of combinators (of combinatory logic) as operations to the signature and the corresponding axioms to define their semantics as default to any specification. Another possibility is to always require that functional sorts are interpreted with the full function space. This requirement goes beyond the power of first-order logic. Moreover, due to Gödel’s famous Incompleteness Theorem [Göd31], such a logic has no complete, finitely axiomatized proof system.

A second, orthogonal issue is the addition of predicate types. This is useful, for example, for modeling predicate transformers. To achieve this, we have to generalize the sort structure, by introducing a predicate type constructor. Moreover, as we added explicit application functions for the functional sorts, we have to add explicit application predicates for the predicate types.

In this setting, since the application of any predicate to some undefined argument yields false, the application of a term of a predicate type to (appropriately typed) terms yields false if the evaluation of any argument term is undefined as well as the evaluation of the predicate term is undefined. This approach is proposed for higher-order CASL [HKM98].

While for the argument terms, it is natural just to chose the same interpretation as in the first-order case (requiring any undefined argument to lead to false), it may seem strange that undefined predicate terms (which may now be applications of a possibly partial function) to yield false as well. In order to avoid possible conflicts, it is also possible to avoid undefined predicate terms by dividing the set of types into two kinds: kind $*$ contains the types of higher-order functions that may eventually (when applied to enough arguments) deliver a truth value, while kind ι contains all the other types. Now all functions of a $*$ -type are required to be total. This is achieved by introducing, for each type of kind $*$, a canonical value corresponding to “everywhere false”. This canonical value is delivered when a function of a $*$ -type is applied to an undefined argument. This approach is used in LUTINS [Far91, Far93].

Another quite natural generalization of the first-order reduction of functional specifications for predicates, is to introduce a special sort of boolean values and then regard the predicates as functions onto that sort. This is a more powerful approach, as it is now easy to describe logical connectives and use formulae in term formation. To our knowledge, such a logic has not been investigated in the literature. Note that this case, a *non-strict* framework as presented in the next section is needed to do a reduction to first-order logic, because application of boolean-valued functions is not strict: it yields false for undefined arguments.

For the combination of partiality with λ -calculus, combinatory logic and polymorphism, see [Mog88,Fef95].

Non strictness. A completely orthogonal problem is that of non-strictness, because it concerns not the logic used to specify data types, but the semantic side, that is the class of acceptable models. Indeed, in partial logic the interpretation of both total and partial operation symbols in the models are *strict*, that is they can produce a result only if all their inputs are provided correct.

This is not the case, for instance, for the *conditional choice*, like the `if_then_else` in many programming languages, that can results in a correct value even if one of the branches would not, because only one branch is actually evaluated.

The classical *escamotage* for representing such functions is lifting their domain to function spaces.

Consider for example the case of `if true then c_1 else c_2` , where c_1 and c_2 are commands, then its evaluation on a state s is the evaluation of c_1 on s disregarding the value, if any, of the evaluation of c_2 . But, even if the evaluation of c_2 on s is incorrect, the interpretation of c_2 is still a well-defined element of a functional sort and hence, technically, the interpretation of `if_then_else` is strict. The same technique, although less naturally, can be applied for instance to Boolean `and` and `or` with lazy valuation, as follows.

```
spec Bool =
  sorts   bool, BoolExps, dummy
  opns     $\cdot$  :  $\rightarrow$  dummy
          T, F :  $\rightarrow$  bool
          true, false :  $\rightarrow$  BoolExps
          BEval : BoolExps  $\times$  dummy  $\rightarrow$  bool
          and, or : BoolExps  $\times$  BoolExps  $\rightarrow$  BoolExps ...
  vars    x, y : BoolExps
  axioms  BEval(true,  $\cdot$ )  $\stackrel{e}{=} T$ 
          BEval(false,  $\cdot$ )  $\stackrel{e}{=} F$ 
          BEval(x,  $\cdot$ )  $\stackrel{e}{=} F \Rightarrow$  and(x, y)  $\stackrel{e}{=} false$ 
          BEval(x,  $\cdot$ )  $\stackrel{e}{=} T \Rightarrow$  and(x, y)  $\stackrel{e}{=} y \dots$ 
          BEval(x,  $\cdot$ )  $\stackrel{e}{=} T \Rightarrow$  or(x, y)  $\stackrel{e}{=} true$ 
          BEval(x,  $\cdot$ )  $\stackrel{e}{=} F \Rightarrow$  or(x, y)  $\stackrel{e}{=} y \dots$ 
```

Where, of course, the specification becomes interesting only if some *partial* constructors for Boolean expressions are provided.

Instead of trying to implement non-strictness inside a strict framework, it is also possible to weaken the requirements on the semantic models, to get a richer class providing *true* non-strictness, so to speak.

A first possibility is considering the case of *monotonic* non-strictness, that is requiring that if a function can produce a result with some of its arguments undefined, then whatever is substituted for them the result should be the same. This point of view deals perfectly well with the so called *don't care* parameters, like the uninteresting branch in conditional choices or superfluous data for suspended valuations. But it cannot be used for error recovery,

because in that case different “undefined” cases should result in different correct recovered data.

In [AC96], an algebraic paradigm for the non-strict don’t care case is presented, that is based on the idea of *partial product*. The intuition is that, while in the standard strict case the argument of an n -ary function are n -tuples, that are functions from the range $[1 \dots n]$ into the carriers, here *partial* n -tuples, that are partial functions from the range $[1 \dots n]$ into the carriers, are allowed as well.

Starting from this new point of view the standard algebraic theory is developed. But it is important to note that the monotonicity requirement implicitly introduces disjunctive axioms. Indeed if we know that $pf(a)$ is defined, for some constant a and unary function pf , then we have that a is defined or $pf(x)$ is defined for whatever value of x (including the undefined). Thus, $D(pf(a))$ is equivalent to $D(a) \vee D(pf(x))$.

Therefore, the theory of equational non-strict data types is more or less equivalent to the theory of disjunctive non-strict data types, that are studied in [AC96], giving necessary and sufficient conditions for the existence of initial models.

When moving from algebras to first-order structures, non-strictness extends to predicates in the following way: a predicate is strict if it yields false whenever some of its arguments is undefined, while non-strict predicates can be true even for undefined arguments. A non-strict partial first-order logic along these lines is developed in [GL97]. It uses total variable valuations (corresponding to our \models^t), and has a special constant denoting undefined. A monotonicity requirement is not imposed.

A completely different point of view on non-strictness is presented in [Cer95], where the intuition is that non-strictness comes from evaluational issues and is not inherent to the underlying data-type. Thus, the idea is to keep as models standard partial algebras, but each one is equipped with a total congruence on terms, representing the simplifications that are done on terms before the actual evaluation.

In this approach error recovery follows the intuition that errors never take place, because terms can be simplified before their evaluation. Thus, a term can be simplified into a perfectly correct term, even if it contains some subterm that is incorrect. For instance a term denoting an integer value, of the form $\mathbf{zero} * t$ can be simplified to \mathbf{zero} , and then evaluated onto the 0 value, even if t is incorrect, allowing in this way a strategy of error recovery.

The form of the axioms allow the definition of subtle error recovery strategy, or lazy/suspended evaluation.

The description operator. A definite description operator allows a term to be constructed from a formula describing the properties of a unique value.

The presence of partial functions makes it easy to introduce a definite description operator [LvF67, Far91]. Assume, indeed, that we have an arbitrary

first order-formula φ ; then $\iota x : s.\varphi$ (read “the x for which φ ”) is a new term of sort s . φ should be the implicit description of some value, such that φ is true if and only if this value is substituted for x . The intended meaning is that $\iota x : s.\varphi$ denotes this unique value, if existing, while it is undefined, if no such value exists or there is more than one. Thus, we have to add the following semantical rule:²²

$$v^\#(\iota x : s.\varphi) = \begin{cases} \xi(x), & \text{if there is a unique } \xi: X \cup \{x : s\} \longrightarrow A \\ & \text{extending } v \text{ on } X \setminus \{x : s\} \\ & \text{for which } \xi \Vdash \varphi \\ \text{undefined,} & \text{otherwise} \end{cases}$$

The description operator is characterized by the axiom

$$\forall y : s.(y \stackrel{e}{=} \iota x : s.\varphi \Leftrightarrow \forall x : s.(\varphi \Leftrightarrow x \stackrel{e}{=} y))$$

which has to be added to the calculus.

For example, the division function can now be defined easily in terms of multiplication:

$$\forall x, y : \text{real}.(x/y \stackrel{s}{=} \iota z : \text{real}.x \stackrel{e}{=} y * z)$$

where $x/0$ is undefined, as expected.

In absence of a division operation, the term $\iota z : \text{real}.x \stackrel{e}{=} y * z$ is not equivalent to a term without ι . Therefore, Corollary 3.44 does no longer hold: there may be term-generated structures which are not reachable.

Three-valued logic. When reasoning about programs, we cannot avoid the use of some three-valued logic. For example, the condition in a while-loop is of type three-valued Boolean, because it may either be true, or false, or undefined due to some infinite computation or some exception.

On the other hand, when writing specifications, we want to specify *properties* of data types and programs, which may hold or not hold, but without a third possibility. For example, a definedness predicate delivering undefined when the argument is undefined does not make much sense. That is, the outer level of specifications is two-valued, but somewhere we have to have the possibility to talk about three-valued programs.

Basically, there are two points where three-valuedness can be introduced.

1. Incorporate three-valuedness into the logic itself. Thus, there is a distinction between the false and the undefined. The fact that some term is non-denoting is propagated to the formulae containing the term. Thus, valuations of terms *and* of formulae have to be partial. The latter causes the need for a third truth-value, say $-$, which is assigned to formulae containing a non-denoting term. Then, non-strict predicates also may yield

²² Syntax of terms and formulae as well as their semantics (i.e. $v^\#$ and $v \Vdash _$) have to be defined in parallel now.

– when applied to denoting terms. The connectives can be extended to deal with – in several ways. A natural choice is Kleene’s three-valued logic [Kle52], which is guided by a strictness (resp. continuity, w.r.t. to a natural associated topology) principle (cf. [Cle91]).

$$\begin{array}{c|c} \wedge & t \quad - \quad f \\ \hline t & t \quad - \quad f \\ - & - \quad - \quad f \\ f & f \quad f \quad f \end{array} \quad \begin{array}{c|c} \vee & t \quad - \quad f \\ \hline t & t \quad t \quad t \\ - & - \quad t \quad - \\ f & f \quad t \quad - \quad f \end{array} \quad \begin{array}{c|c} \neg & \\ \hline t & f \\ - & - \\ f & t \end{array} \quad \begin{array}{c|c} \Rightarrow & t \quad - \quad f \\ \hline t & t \quad - \quad f \\ - & - \quad t \quad - \\ f & t \quad t \quad t \end{array}$$

It is used e.g. in the specification languages SPECTRUM [BFG⁺93] and VDM [Jon90]. The latter reference also describes an easy calculus for the Kleene logic.

But as said above, at the level of specifications there is the need for other, two-valued connectives which are non-strict (resp. non-continuous) when incorporated in the three-valued world. For example, the definedness predicate

$$\begin{array}{c|c} D & \\ \hline t & t \\ - & f \\ f & t \end{array}$$

is non-strict but nevertheless needed for specifications. Another example is an implication like

$$\forall x, y, z : \text{nat}. x = z/y \Rightarrow x * y = z$$

which is valid without a restriction on y in two-valued logic, but not in three-valued logic. To make it valid in three-valued logic, we have to use a non-strict implication which identifies false and –.

But non-strict connectives complicate the calculus by introducing the need for a more complex case analysis.

See [Cle91] for an overview over multi-valued logics.

2. Use a two-valued logic as introduced above and shift the issue of three-valuedness to the object level. This can be done with a specification like

```
spec ThreeValued =
  sorts   Bool3
  preds    $\stackrel{3}{=} : \text{Bool3} \times \text{Bool3}$ 
  opns    t, f, - :  $\rightarrow \text{Bool3}$ 
            $\neg^3 : \text{Bool3} \rightarrow \text{Bool3}$ 
            $\wedge^3, \text{logor}^3, \Rightarrow^3 : \text{Bool3} \times \text{Bool3} \rightarrow \text{Bool3}$ 
  vars    x, y : Bool3
  axioms   $\forall x : \text{Bool3}. x \stackrel{e}{=} t \vee x \stackrel{e}{=} f \vee x \stackrel{e}{=} -$ 
            $t \vee^3 - \stackrel{e}{=} t$ 
           ...
```

$$\begin{aligned}
(x \stackrel{3}{=} y) \stackrel{e}{=} t &\Leftrightarrow x \stackrel{e}{=} y \\
(x \stackrel{3}{=} y) \stackrel{e}{=} f &\Leftrightarrow (D(x) \wedge D(y) \wedge \neg x \stackrel{e}{=} y) \\
(x \stackrel{3}{=} y) \stackrel{e}{=} - &\Leftrightarrow (\neg D(x) \vee \neg D(y)) \\
\dots
\end{aligned}$$

A quantified formula $\forall X.\varphi$ is expressed by first translating φ inductively to a term $\overset{3}{\varphi} \stackrel{e}{=} \text{Bool13}$ and then taking $\forall X.\overset{3}{\varphi}$ to be

$$\begin{aligned}
ib : \text{Bool13. } ((\forall X. \overset{3}{\varphi} \stackrel{e}{=} t) \Rightarrow b \stackrel{e}{=} t) \\
\wedge ((\forall X. \neg \overset{3}{\varphi} \stackrel{e}{=} - \wedge \exists X. \overset{3}{\varphi} \stackrel{e}{=} f) \Rightarrow b \stackrel{e}{=} f) \\
\wedge ((\exists X. \overset{3}{\varphi} \stackrel{e}{=} -) \Rightarrow b \stackrel{e}{=} -)
\end{aligned}$$

The need for a case analysis when doing theorem proving is made explicit in the definitions here.

References

- [AC92] E. Astesiano and M. Cerioli. Partial higher-order specifications. *Fundamenta Informaticae*, 16(2):101–126, 1992.
- [AC95] E. Astesiano and M. Cerioli. Free objects and equational deduction for partial conditional specifications. *Theoretical Computer Science*, 152(1):91–138, 1995.
- [AC96] E. Astesiano and M. Cerioli. Non-strict don't care algebras and specifications. *Mathematical Structures in Computer Science*, 6(1):85–125, 1996.
- [AHS90] J. Adámek, H. Herrlich, and G. Strecker. *Abstract and Concrete Categories*. Wiley, New York, 1990.
- [AR94] J. Adámek and J. Rosický. *Locally Presentable and Accessible Categories*. Cambridge University Press, 1994.
- [BBC86] G. Bernot, M. Bidoit, and C. Choppy. Abstract data types with exception handling: an initial approach based on a distinction between exceptions and errors. *Theoretical Computer Science*, 46(1):13–45, 1986.
- [Bee85] Michael J. Beeson. *Foundations of Constructive Mathematics*, volume 6 of *Ergebnisse der Mathematik und ihrer Grenzgebiete (3. Folge)*. Springer, revised edition, 1985.
- [Bee86] M.J. Beeson. Proving programs and programming proofs. In R.B. Marcus et al., editors, *Logic, Methodology and Philosophy of Science VII*, pages 51–82. North Holland, 1986.
- [BFG⁺93] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hußmann, D. Nazareth, F. Regensburger, and K. Stølen (The Munich SPECTRUM Group). The requirement and design specification language SPECTRUM: An informal introduction, version 1.0, part I. Technical Report TUM–19311, TUM–19312, Institut für Informatik, Technische Universität München, 1993.
- [BG94] L. Bachmair and H. Ganzinger. Rewrite techniques for transitive relations. In *Proc. 9th IEEE Symposium on Logic in Computer Science*, pages 384–393. IEEE Computer Society Press, 1994. Short version of TR MPI-I-93-249.
- [Bor94] F. Borceux. *Handbook of Categorical Algebra I – III*. Cambridge University Press, 1994.
- [BR83] K. Benecke and H. Reichel. Equational partiality. *Algebra Universalis*, 16:219–232, 1983.
- [BT87] J.A. Bergstra and J.V. Tucker. Algebraic specifications of computable and semicomputable data types. *Theoretical Computer Science*, 50:137–181, 1987.
- [Bur82] P. Burmeister. Partial algebras – survey of a unifying approach towards a two-valued model theory for partial algebras. *Algebra Universalis*, 15:306–358, 1982.
- [Bur86] P. Burmeister. *A Model Theoretic Oriented Approach to Partial Algebras*. Akademie-Verlag, Berlin, 1986.
- [BW82] M. Broy and M. Wirsing. Partial abstract data types. *Acta Informatica*, 18(1):47–64, 1982.
- [Cer93] M. Cerioli. *Relationships between Logical Formalisms*. PhD thesis, Universities of Genova, Pisa and Udine, 1993. Available as internal report of Pisa University, TD-4/93 or by anonymous ftp at `ftp.disi.unige.it` in `/pub/cerioli/thesis92.ps.z`.

- [Cer95] M. Cerioli. A lazy approach to partial algebras. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specification: 10th Workshop on Specification of Abstract Data Types – Selected Papers*, volume 906 of *Lecture Notes in Computer Science*, pages 188–202. Springer, 1995.
- [CGW95] I. Claßen, M. Große-Rhode, and U. Wolter. Categorical concepts for parameterized partial specifications. *Mathematical Structures in Computer Science*, 5(2):153–188, 1995.
- [CHKM97] M. Cerioli, A. Hauxthausen, B. Krieg-Brückner, and T. Mossakowski. Permissive subsorted partial logic in CASL. In Michael Johnson, editor, *Algebraic Methodology and Software Technology (AMAST'97)*, volume 1349 of *Lecture Notes in Computer Science*, pages 91–107. Springer, 1997.
- [CJ91] J.H. Cheng and C.B. Jones. On the usability of logics which handle partial functions. In C. Morgan and J.C.P. Woodcock, editors, *Proc. of the Third Refinement Workshop*, Workshops in Computing series, pages 51–69. Springer, 1991.
- [Cle91] J.P. Cleave. *A Study of Logics*. Oxford University Press, 1991.
- [CM97] M. Cerioli and J. Meseguer. May I borrow your logic? (transporting logical structures along maps). *Theoretical Computer Science*, 173(2):311–347, 1997.
- [Dav65] M. Davis, editor. *The Undecidable*. Raven Press, New York, 1965.
- [DJ90] Nachem Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Chapter 6*, pages 244–320. Elsevier Science Publishers B.V. (North-Holland), 1990.
- [EM85] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.
- [Far91] W.M. Farmer. A partial functions version of Church's simple type theory. *Journal of Symbolic Logic*, 55:1269–1291, 1991.
- [Far93] W.M. Farmer. A simple type theory with partial functions and subtypes. *Annals of Pure and Applied Logic*, 64:211–240, 1993.
- [Far95] W.M. Farmer. Reasoning about partial functions with the aid of a computer. *Erkenntnis*, 43:279–294, 1995.
- [Fef92] S. Feferman. A new approach to abstract data types, i informal development. *Mathematical Structures in Computer Science*, 2:193–229, 1992.
- [Fef95] S. Feferman. Definedness. *Erkenntnis*, 43:295–320, 1995.
- [G⁺93] D.M. Gabbay et al. *Handbook of Logic in Artificial Intelligence and Logic Programming, Vol. 1: Logical Foundations*. Oxford Science Publications, 1993.
- [GL97] R.D. Gumb and K. Lambert. Definitions in nonstrict positive free logic. *Modern Logic*, 7:25–55, 1997.
- [GM85] J. Goguen and J. Meseguer. Completeness of many-sorted equational logic. *Houston Journal of Mathematics*, 11(3):307–334, 1985.
- [GM86a] J.A. Goguen and J. Meseguer. Remarks on remarks on many-sorted equational logic. *EATCS Bulletin*, 30:66–73, 1986.

- [GM86b] Joseph A. Goguen and José Meseguer. EQLOG: Equality, types, and generic modules for logic programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations and Equations*, pages 259–363. Prentice Hall, 1986. An earlier version appeared in *Journal of Logic Programming*, 1(2):179–210, 1984.
- [GM87] J.A. Goguen and J. Meseguer. Order-sorted algebra solves the constructor selector, multiple representation and coercion problems. In *Proc. Second Symposium on Logic in Computer Science*, pages 18–29. IEEE Computer Society, 1987. Also Report CSLI-87-92, Center for the Study of Language and Information, Stanford University, March 1987; revised version in *Information and Computation*, 103, 1993.
- [GM92] J.A. Goguen and J. Meseguer. Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273, 1992.
- [GMJ85] J.A. Goguen, J. Meseguer, and J.-P. Jouannaud. Operational semantics of order-sorted algebra. In Wilfried Brauer, editor, *Proc. 1985 Intl. Conference on Automata, Languages and Programming*, volume 194 of *Lecture Notes in Computer Science*, pages 221–231. Springer, 1985.
- [GMW⁺93] J. Goguen, J. Meseguer, T. Winkler, K. Futatsugi, P. Lincoln, and J.-P. Jouannaud. Introducing OBJ3. Technical Report SRI-CSL-88-8, Computer Science Lab, SRI International, August 1988; to appear in *Applications of Algebraic Specification using OBJ*, 1993.
- [Göd31] Kurt Gödel. Über formal unentscheidbare Sätze der *Principia Mathematica* und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–98, 1931. Translation by Elliot Mendelsohn printed in [Dav65].
- [Gog87] M. Gogolla. On parametric algebraic specifications with clean error handling. In H. Ehrig, R. Kowalski, G. Levi, and U. Montanari, editors, *Proc. TAPSOFT'87*, volume 249 of *Lecture Notes in Computer Science*, pages 81–95. Springer, 1987.
- [Grä79] G. Grätzer. *Universal Algebra*. Springer, 2nd edition, 1979.
- [GTW78] Joseph A. Goguen, James W. Thatcher, and Eric G. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In Raymond Yeh, editor, *Current Trends in Programming Methodology, IV*, pages 80–149. Prentice-Hall, 1978. Also as Report RC 6487, IBM T.J. Watson Research Center, 1976.
- [Hai53] T. Hailperin. Quantification theory and empty individual-domains. *Journal of Symbolic Logic*, 18:197–200, 1953.
- [Hen50] L. Henkin. Completeness in the theory of types. *Journal of Symbolic Logic*, 15:81–91, 1950.
- [Her73] H. Hermes. *Introduction to mathematical logic*. Springer, 1973.
- [HKM98] A.E. Haxthausen, B. Krieg-Brückner, and T. Mossakowski. Extending CASL with higher-order functions – design proposal. CoFI note: L-8²³, 1998.
- [HO80] G. Huet and D. Oppen. Equations and rewrite rules: A survey. In R.V. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 349–405. Academic Press, New York, 1980.

²³ <http://www.brics.dk/Projects/CoFI/Notes/L-8/index.html>

- [Jon90] C.B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 1990.
- [KGS88] L. Kreiser, S. Gottwald, and W. Stelzner. *Nichtklassische Logik – Eine Einführung*. Akademie Verlag, Berlin, 1988.
- [KKM98] Kolyang, B. Krieg-Brückner, and T. Mossakowski. Static semantic analysis of CASL. In F. Parisi Presicce, editor, *Proc. 12th Intl. Workshop on Recent Trends in Algebraic Development Techniques*, volume 1376 of *Lecture Notes in Computer Science*, pages 333–348. Springer, 1998.
- [Kle52] S.C. Kleene. *Introduction to Metamathematics*. North Holland, 1952.
- [Klo92] Jan Willem Klop. Term rewriting systems. In S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science, Vol. 2, Chapter 1*, pages 1–116. Oxford University Press, 1992.
- [KM95] H.J. Kreowski and T. Mossakowski. Equivalence and difference of institutions: Simulating horn clause logic with based algebras. *Mathematical Structures in Computer Science*, 5:189–215, 1995.
- [KN94] P. Knijnenburg and F. Nordemann. Partial hyperdoctrines: categorical models for partial function logic and Hoare logic. *Mathematical Structures in Computer Science*, 4:117–146, 1994.
- [Kre87] H.J. Kreowski. Partial algebras flow from algebraic specifications. In T. Ottmann, editor, *Proc. ICALP'87*, volume 267 of *Lecture Notes in Computer Science*, pages 521–530. Springer, 1987.
- [Lam91] K. Lampert, editor. *Philosophical Applications of Free Logic*. Oxford University Press, New York, 1991.
- [Llo87] J.W. Lloyd. *Foundations of Logic Programming*. Springer, 1987.
- [LvF67] K. Lampert and B. v. Fraassen. On free description theory. *Zeitschrift für Logik und Grundlagen der Mathematik*, 13:225–240, 1967.
- [Mei92] K. Meinke. Universal algebra in higher types. *Theoretical Computer Science*, 100(2):385–417, 1992.
- [MG85] J. Meseguer and J.A. Goguen. Initiality, induction and computability. In M. Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–541. Cambridge University Press, 1985.
- [MG93] J. Meseguer and J.A. Goguen. Order-sorted algebra solves the constructor selector, multiple representation and coercion problems. *Information and Computation*, 103(1):114–158, 1993.
- [Mog88] E. Moggi. *The Partial Lambda-Calculus*. PhD thesis, University of Edinburgh, 1988. Report CST-53-88.
- [Möl87] B. Möller. Algebraic specification with higher-order operations. In *Proc. IFIP TC 2 Working Conference on Program Specification and Transformation*. North Holland, 1987.
- [Mos89] P. Mosses. Unified algebras and institutions. In *Proc. 4th Annual IEEE Symposium on Logic in Computer Science*, pages 304–312, 1989.
- [Mos93] Peter D. Mosses. The use of sorts in algebraic specifications. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification, Selected Papers from the 8th Workshop on Specification of Abstract Data Types*, volume 655 of *Lecture Notes in Computer Science*, pages 66–91. Springer, 1993.
- [Mos96] T. Mossakowski. Equivalences among various logical frameworks of partial algebras. In H. Kleine Büning, editor, *Proc. 9th Workshop on Computer Science Logic (CSL'95). Selected Papers*, volume 1092 of *Lecture Notes in Computer Science*, pages 403–433. Springer, 1996.

- [Mos97] Peter D. Mosses. CoFI: The common framework initiative for algebraic specification and development. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97*, volume 1214 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 1997.
- [Mos98] T. Mossakowski. Colimits of order-sorted specifications revisited. In F. Parisi Presicce, editor, *Proc. 12th Intl. Workshop on Recent Trends in Algebraic Development Techniques*, volume 1376 of *Lecture Notes in Computer Science*, pages 316–332. Springer, 1998.
- [MSS90] V. Manca, A. Salibra, and G. Scollo. Equational type logic. *Theoretical Computer Science*, 77:131–159, 1990. Special Issue dedicated to AMAST'89.
- [MSS92] V. Manca, A. Salibra, and G. Scollo. On the expressiveness of equational type logic. In C.M.I. Rattray and R.G. Clark, editors, *The Unified Computation Laboratory*. Oxford University Press, 1992.
- [MT93] J. Meinke and J.V. Tucker, editors. *Many-sorted Logic and its Applications*. Wiley, 1993.
- [MTW88] B. Möller, A. Tarlecki, and M. Wirsing. Algebraic specification with built-in domain constructions. In M. Dauchet and M. Nivat, editors, *Proc. CAAP'88*, volume 299 of *Lecture Notes in Computer Science*, pages 132–148. Springer, 1988.
- [Obe62] A. Oberschelp. Untersuchungen zur mehrsortigen Quantorenlogik. *Mathematische Annalen*, 145(1):297–333, 1962.
- [Pad88] P. Padawitz. *Computing in Horn Clause Theories*. Springer, 1988.
- [Par95] D.L. Parnas. A logic for describing, not verifying, software. *Erkenntnis*, 43:321–338, 1995.
- [Poi87] A. Poigné. Partial algebras, subsorting, and dependent types: Prerequisites of error handling in algebraic specifications. In *Recent Trends in Data Type Specification: 5th Workshop on Specification of Abstract Data Types – Selected Papers*, volume 332 of *Lecture Notes in Computer Science*, pages 208–234. Springer, 1987.
- [Poi90] A. Poigné. Parameterization for order-sorted algebraic specification. *Journal of Computer and System Sciences*, 40:229–268, 1990.
- [Rei87] H. Reichel. *Initial Computability, Algebraic Specifications, and Partial Algebras*. Oxford University Press, 1987.
- [Sch70] J. Schmidt. A homomorphism theorem for partial algebras. *Coll. Math.*, 21:5–21, 1970.
- [Sco79] D.S. Scott. Identity and existence in intuitionistic logic. In M.P. Fourman, C.J. Mulvey, and D.S. Scott, editors, *Application of Sheaves*, volume 753 of *Lecture Notes in Mathematics*, pages 660–696. Springer Verlag, 1979.
- [Tar85] A. Tarlecki. On the existence of free models in abstract algebraic institutions. *Theoretical Computer Science*, 37(3):269–304, 1985.
- [TWW81] J. Thatcher, E.G. Wagner, and J.B. Wright. Specification of abstract data types using conditional axioms. Technical Report RC 6214, IBM Yorktown Heights, 1981.
- [Wol90] U. Wolter. An algebraic approach to deduction in equational partial horn theories. *J. Inf. Process. Cybern. EIK*, 27(2):85–128, 1990.
- [Yan93] Han Yan. *Theory and Implementation of Sort Constraints for Order Sorted Algebra*. PhD thesis, Oxford University, 1993.