Chapter 3

From total equational to partial first order

(Preliminary version of the chapter for the forthcoming IFIP state of the art book)

Maura Cerioli, Till Mossakowski and Horst Reichel

The focus of this Chapter is the incremental presentation of partial first-order logic, seen as a powerfull 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 is investigated in detail, and in particular the existence of initial (free) objects 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 type used in programming languages and their semantics. Indeed, Bergstra and Tucker have shown in a series of papers (see [16] 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.

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 reacher structure..

Example 3.0.1 *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.*

$$\begin{array}{l} \textbf{sig } \Sigma_{\texttt{Nat}} = \\ \quad \textbf{sorts } \texttt{nat}, \texttt{nat}/\equiv \\ \quad \textbf{opns } \texttt{zero}: \to \texttt{nat} \\ \quad\quad\quad\; \texttt{succ}: \texttt{nat} \to \texttt{nat} \\ \quad\quad\quad\; nat_{\equiv}: \texttt{nat} \to \texttt{nat}/\equiv \end{array}$$

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 $f^k(\ldots)$ 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.*

$\texttt{succ}^k(\texttt{zero}) = \texttt{succ}^n(\texttt{zero})$ *that is either trivial or not valid in our intended model, as the term $\texttt{succ}^k(\texttt{zero})$ is interpreted by the number k.*

$\texttt{succ}^k(\texttt{zero}) = \texttt{succ}^n(x)$ *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.*

$\mathtt{succ}^k(x) = \mathtt{succ}^n(x)$ *that is either trivial or does not hold in our intended model if* 0 *is substituted for* x.

$\mathtt{succ}^k(x) = \mathtt{succ}^n(y)$ *with* x *and* y *distinct variables, that does not hold if* 0 *is substituted for* x *and* $k+1$ *for* y

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

$nat_\equiv(\mathtt{succ}^k(\mathtt{zero})) = nat_\equiv(\mathtt{succ}^n(\mathtt{zero}))$ *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.*

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

$nat_\equiv(\mathtt{succ}^k(x)) = nat_\equiv(\mathtt{succ}^n(x))$ *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.

$nat_\equiv(\mathtt{succ}^k(x)) = nat_\equiv(\mathtt{succ}^n(y))$ *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*

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

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* $nat_\equiv(\mathtt{succ}^{2k+1}(\mathtt{zero}))$ *can be inferred.*

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

$$
\begin{array}{ll}
\textbf{spec } \mathtt{Odd} = & \textbf{enrich } \Sigma_{\mathtt{Nat}} \textbf{ by} \\
\quad \textbf{sorts} & \mathtt{bool} \\
\quad \textbf{opns} & \mathtt{true}, \mathtt{false} \colon \to \mathtt{bool} \\
 & \mathtt{odd} \colon \mathtt{nat} \to \mathtt{bool} \\
 & \mathtt{cond} \colon \mathtt{bool} \times \mathtt{nat} \times \mathtt{nat} \to \mathtt{nat} \\
\textbf{axioms} & \mathtt{cond}(\mathtt{true}, x, y) = x \\
 & \mathtt{cond}(\mathtt{false}, x, y) = y \\
 & \mathtt{odd}(\mathtt{zero}) = \mathtt{false} \\
 & \mathtt{odd}(\mathtt{succ}(\mathtt{zero})) = \mathtt{true} \\
 & \mathtt{odd}(\mathtt{succ}(\mathtt{succ}(x))) = \mathtt{odd}(x) \\
 & nat_\equiv(x) = \mathtt{cond}(\mathtt{odd}(x), nat_\equiv(\mathtt{succ}(\mathtt{zero})), nat_\equiv(x))
\end{array}
$$

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. On the other hand, positive conditional logics have a simpler 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 formulas 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 loose the initial semantic approach. Thus, our formulas 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, or 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 compromise 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 3.0.1, we need a way to impose equations only to those values that satisfy a condition, implemented in that example by introducing the operation `cond`, corresponding to a conditional choice, and then imposing the axiom

$$nat_{\equiv}(x) = \mathtt{cond}(\mathtt{odd}(x), nat_{\equiv}(\mathtt{succ}(\mathtt{zero})), nat_{\equiv}(x))$$

that corresponds, logically, to requiring

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

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

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

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 [16], 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 3.0.1, that cannot be axiomatized be a finite set of equations on its signature, can be easily defined by the following conditional axiom

$$\phi_{\mathtt{odd}} \quad nat_{\equiv}(x) = nat_{\equiv}(\mathtt{succ}(\mathtt{zero})) \Rightarrow nat_{\equiv}(\mathtt{succ}(\mathtt{succ}(x))) = nat_{\equiv}(x)$$

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

The point is that the informal specification of $nat_{\equiv}$ is based on the distinction between odd and even numbers, but our signature does not have syntactical meaning 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 a `odd` Boolean function, but it is much more clear to add a facility to our specification framework, providing symbols for *predicates*.

Definition 3.1.1 *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 (*predicate symbols*)*

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

Example 3.1.2 *A reasonable signature for the Example 3.0.1, then is the following.*

sig $\Sigma_{\mathtt{Nat}_P}$ = **enrich** $\Sigma_{\mathtt{Nat}}$ **by**
sorts $\mathtt{nat}/\equiv$
opns $nat_{\equiv}: \mathtt{nat} \to \mathtt{nat}/\equiv$
preds $\mathtt{is\_odd}: \mathtt{nat}$

The signature $\Sigma_{\mathtt{Nat}_P}$ *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.*

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 3.1.3 *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* $h: A \longrightarrow B$ *s.t. if* $(a_1, \ldots, a_n) \in P_A$*, then* $(h_{s_1}(a_1), \ldots, h_{s_n}(a_n)) \in P_B$ *for all* $P: s_1 \times \cdots \times s_n \in \Pi$ *and all* $a_i \in |A|_{s_i}$ *for* $i = 1, \ldots, n$*.*

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* $!_A: I \longrightarrow A$ *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*.*

Notice the difference between enriching a signature by a Boolean sort and some operations, as in the equational presentation of Example 3.0.1, 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.

This distinction has an important implication for proof theory. Indeed, with positive conditional axioms and predicates, we cannot talk about falsehood of predicates. The approach of enriching a signature by a Boolean sort and treating predicates as operations to this sort does not have this restriction. But to ensure that the Boolean sort indeed has just two elements, a more complex (first-order) axiom is needed. Thus the simpler proof theory of positive conditional axioms cannot be used here.

Another aspect is the easy specification of inductively defined relations using initial semantics (or initial constraints). This is possible only with the predicate approach. Note that initial constraints are a second-order principle, so proof theory here gets even more complex, since an induction principle is needed.

Consider for instance once again the Example 3.0.1. Then a model of the specification `Odd` is the algebra

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

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

spec $\mathbf{N}_p$ =
Carriers
$|\mathbf{N}_p|_{\mathtt{nat}} = \mathbf{N}$
$|\mathbf{N}_p|_{\mathtt{nat}/\equiv} = 2\mathbf{N} \cup \{\bar{1}\}$
Functions
$\mathtt{zero}_{\mathbf{N}_p} = 0$
$\mathtt{succ}_{\mathbf{N}_p}(n) = n + 1$

$$\mathtt{plus}_{\mathbf{N}_p}(n,m) = n+m$$
$$nat_{\equiv\mathbf{N}_p}(n) = \begin{cases} n, & \text{if there is } k \text{ s.t. } n = 2k \\ \bar{1}, & \text{otherwise} \end{cases}$$

Predicates

$$\mathtt{is\_odd}_{\mathbf{N}_p} = \{n \mid \exists k \in \mathbf{N} \text{ s.t. } n = 2k+1\}$$

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

Let us now formally define conditional axioms and their validity.

Definition 3.1.4 *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*

$$\begin{array}{rcl} At(\Sigma, X) & = & \{t = t' \mid t, t' \in |T_\Sigma(X)|_s\} \cup \{P(t_1, \ldots, t_n) \mid \\ & & P\colon s_1 \times \cdots \times s_n \in \Pi \text{ and } t_i \in |T_\Sigma(X)|_{s_i}\ i = 1, \ldots, n\} \end{array}$$

The set of Σ-conditional axioms *on X is*

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

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 3.1.5 *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 ν for X in A* satisfy *φ (denoted by $\nu \Vdash \varphi$), where satisfaction of a conditional axiom by a valuation is defined by the following rules:*

- $\nu \Vdash t = t'$ *iff* $\nu^\#(t) = \nu^\#(t')$
- $\nu \Vdash P(t_1, \ldots, t_n)$ *iff* $(\nu^\#(t_1), \ldots, \nu^\#(t_n)) \in P_A$
- $\nu \Vdash \epsilon_1 \wedge \cdots \wedge \epsilon_n \Rightarrow \epsilon_{n+1}$ *iff* $\nu \Vdash \epsilon_{n+1}$ *or there is an ϵ_i s.t.* $\nu \not\Vdash \epsilon_i$

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

Exercise 3.1.6 *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. In particular the definition of sub-structure, product, and reachable structure are inherited from the many-sorted case, with the predicates defined in the one reasonable way (the interested reader is encouraged to generalize the theory of the previous chapter to the case of predicates). Moreover conditional axioms, as in the case without predicates, define *quasi varieties* and hence their model classes always admit initial models.

Since Σ-structures are a particular case of *partial first-order structures*, studied afterward in the present chapter, and their theory does not introduce innovative aspects, here we do not investigate the matter in details. But 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.

Definition 3.1.7 *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

$$\begin{array}{l} \Phi \vdash \forall X.t = t \\ \Phi \vdash \forall X.t = t' \Rightarrow t' = t \\ \Phi \vdash \forall X.t = t' \wedge t' = t'' \Rightarrow t = t'' \\ \Phi \vdash \forall X.t_1 = t'_1 \wedge \ldots \wedge t_n = t'_n \Rightarrow f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n) \\ \Phi \vdash \forall X.t_1 = t'_1 \wedge \ldots \wedge t_n = t'_n \wedge P(t_1, \ldots, t_n) \Rightarrow P(t'_1, \ldots, t'_n) \end{array}$$

Proper Axioms

$$\Phi \vdash \forall X.\varphi \qquad \forall X.\varphi \in \Phi$$

Substitution

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

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

Cut Rule

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

Proposition 3.1.8 *The calculus introduced in Definition 3.1.7 is sound, that is if $\Phi \vdash \forall X.\varphi$ and $M \models \Phi$, then $M \models \forall X.\varphi$.*

Proof. *By induction on the definition of $\vdash$.* □

Exercise 3.1.9 *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 & no-confusion* conditions.

Definition 3.1.10 *Let $\langle \Sigma, \Phi \rangle$ be a presentation and $\Theta \subseteq At(\Sigma, X)$ be a finite set of atoms. Then the structure $F_{\langle \Sigma, \Phi \rangle}(X.\Theta)$ has as underlying algebra $T_\Sigma(X)/\equiv_\Theta$, where $t \equiv_\Theta t'$ if and only if $\Phi \vdash \forall X. \bigwedge \Theta' \Rightarrow t = t'$ for some $\Theta' \subseteq \Theta$, and a predicate $P: s_1 \times \cdots \times s_n \in \Pi$ is interpreted as*

$$\{ ([t_1], \ldots, [t_n]) \mid \Phi \vdash \forall X. \bigwedge \Theta' \Rightarrow P(t_1, \ldots, t_n) \text{ for some } \Theta' \subseteq \Theta \}$$

where we denote by $[t]$ the equivalence class in $\equiv_\Theta$ of any term t. □

It is immediate to verify that $\equiv_\Theta$ 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, $P_{F_{\langle \Sigma, \Phi \rangle}(X.\Theta)}$ is well defined.

Lemma 3.1.11 *Each valuation $\nu: Y \longrightarrow F_{\langle \Sigma, \Phi \rangle}(X.\Theta)$ can be factorized (in general not uniquely) through $\_/\equiv_\Theta: T_\Sigma(X) \longrightarrow F_{\langle \Sigma, \Phi \rangle}(X.\Theta)$ as follows*

$$\begin{tikzcd} Y \arrow[rr, "\nu"] \arrow[dr, "\overline{\nu}"'] & & F_{\langle \Sigma, \Phi \rangle}(X.\Theta) \\ & T_\Sigma(X) \arrow[ur, "\_/\equiv_\Theta"'] & \end{tikzcd}$$

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

Proof. *Let us define $\overline{\nu}(y) := t$ for some $t \in \nu(y)$. Then it is easy to show by induction on term structure that $\nu^\#(u) = [\overline{\nu}^\#(u)]$ for each $u \in T_\Sigma(Y)$.*

Now if ϵ is of form $u_1 = u_2$, then $\nu \Vdash \epsilon$ iff $\nu^\#(u_1) = \nu^\#(u_2)$ iff $[\overline{\nu}^\#(u_1)] = [\overline{\nu}^\#(u_2)]$ iff $\overline{\nu}^\#(u_1) \equiv \overline{\nu}^\#(u_2)$ iff $\Phi \vdash \forall X. \bigwedge \Theta' \Rightarrow \overline{\nu}^\#(u_1) = \overline{\nu}^\#(u_2)$, for some $\Theta' \subseteq \Theta$.

If ϵ is of form $P(u_1, \ldots, u_n)$, the proof is similar. □

Theorem 3.1.12 *Using the notation of Definition 3.1.10.*

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

$$\begin{tikzcd} X \arrow[rr, "\iota"] \arrow[dr, "\nu"'] & & F_{\langle \Sigma, \Phi \rangle}(X.\Theta) \arrow[dl, "\tilde{\nu}"] \\ & A & \end{tikzcd}$$

3. *If $\Theta = \emptyset$ then $F_{\langle \Sigma, \Phi \rangle}(X.\Theta)$ and ι are the free object in the model class of Φ.*

Proof.

1. Let $\forall Y.\epsilon_1 \wedge \cdots \wedge \epsilon_n \Rightarrow \epsilon \in \Phi$, so that $\Phi \vdash \forall Y.\epsilon_1 \wedge \cdots \wedge \epsilon_n \Rightarrow \epsilon$, and $\nu: Y \longrightarrow F_{\langle \Sigma, \Phi \rangle}(X.\Theta)$ be a valuation satisfying $\epsilon_1, \ldots, \epsilon_n$. By the Substitution Rule applied with any $\overline{\nu}$ s.t. $\_/\equiv_\Theta \circ \overline{\nu} = \nu$, $\Phi \vdash \forall X.\overline{\nu}^\#(\epsilon_1) \wedge \cdots \wedge \overline{\nu}^\#(\epsilon_n) \Rightarrow \overline{\nu}^\#(\epsilon)$. By Lemma 3.1.11, for all $i = 1, \ldots, n$, $\Phi \vdash \forall X. \bigwedge \Theta_i \Rightarrow \overline{\nu}^\#(\epsilon_i)$ for some $\Theta_i \subseteq \Theta$. By the Cut Rule, $\Phi \vdash \forall X. \bigwedge \cup_{i=1}^n \Theta_i \Rightarrow \overline{\nu}^\#(\epsilon)$. Again by Lemma 3.1.11, $\nu \Vdash \epsilon$. Thus $F_{\langle \Sigma, \Phi \rangle}(X.\Theta)$ is a $\langle \Sigma, \Phi \rangle$-algebra.

2. Using the notation of Lemma 3.1.11, let $\overline{\iota}$ be defined by $\overline{\iota}(x) := x$. Then, $\overline{\iota}^\# = id_{T_\Sigma(X)}$; hence Lemma 3.1.11 immediately gives us $\iota \Vdash \Theta$, as $\Phi \vdash \forall X.\theta \Rightarrow \theta$ for all atoms $\theta \in \Theta$ (see Exercise 3.1.9).

   Now let $\nu: X \longrightarrow A$ be a valuation into a $\langle \Sigma, \Phi \rangle$-algebra A such that $\nu \Vdash \Theta$.

   Just put $\tilde{\nu}([t]) := \nu^\#(t)$, which is the only possible choice. To prove well-definedness, let $t \equiv t'$. Then $\Phi \vdash \forall X. \bigwedge \Theta \Rightarrow t = t'$. By soundness of the calculus and $\nu \Vdash \Theta$, we get $\nu \Vdash t = t'$, that is, $\nu^\#(t) = \nu^\#(t')$.

   The homomorphic property is showed similarly.

3. If Θ is empty, then all algebras satisfy it. Thus, by the previous point 2, for any valuation $\nu: X \longrightarrow A$ into a $\langle \Sigma, \Phi \rangle$-algebra A, there exists exactly one homomorphism $\tilde{\nu}: F_{\langle \Sigma, \Phi \rangle}(X.\Theta) \longrightarrow A$ with $\tilde{\nu} \circ \iota = \nu$. That is $F_{\langle \Sigma, \Phi \rangle}(X.\Theta)$ and ι are the free object in the model class of Φ.

□

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 *strong completeness*. Obviously any strongly complete system can be made complete in the usual sense by adding a weakening rule.

Theorem 3.1.13 *The calculus is strongly complete, that is if* $\Phi \models \forall X. \bigwedge \Theta \Rightarrow \epsilon$*, then* $\Theta' \subseteq \Theta$ *exists s.t.* $\Phi \vdash \forall X. \bigwedge \Theta' \Rightarrow \epsilon$*.*

Proof. Assume $\Phi \models \forall X. \bigwedge \Theta \Rightarrow \epsilon$. By Theorem 3.1.12, $\iota \Vdash \Theta$. By this and the assumption, $\iota \Vdash \epsilon$. By Lemma 3.1.11, $\Phi \vdash \forall X. \bigwedge \Theta' \Rightarrow \bar{\iota}^{\#}(\epsilon)$ for some $\Theta' \subseteq \Theta$. But since $\bar{\iota}^{\#} = id_{T_\Sigma(X)}$, $\Phi \vdash \forall X. \bigwedge \Theta' \Rightarrow \epsilon$. □

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

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

Example 3.1.14 *Let us consider the problem of the definition of a very primitive dynamic data type that is a subset of the CCS language. 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 possible activities of an agent at a fixed instant cannot be given by a function, because our agents are able to perform non-deterministic choices, and 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`*. 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. Moreover there is an* internal *action 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 changement 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.*

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{spec}\ \mathtt{Action}_{-} =} \\
\quad\mathbf{sorts} & \mathtt{Act} \\
\quad\mathbf{opns} & \tau : \to \mathtt{Act} \\
 & \bar{\_} : \mathtt{Act} \to \mathtt{Act} \\
\quad\mathbf{axioms} & \bar{\tau} = \tau \\
 & \bar{\bar{a}} = a
\end{array}
$$

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.*

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{spec}\ \mathtt{CCS}_{-} = \ \mathbf{enrich}\ \mathtt{Action}\ \mathbf{by}} \\
\quad\mathbf{sorts} & \mathtt{Agents} \\
\quad\mathbf{opns} & \lambda : \to \mathtt{Agents} \\
 & \_.\_ : \mathtt{Act} \times \mathtt{Agents} \to \mathtt{Agents} \\
 & \_||\_,\ \_+\_ : \mathtt{Agents} \times \mathtt{Agents} \to \mathtt{Agents} \\
\quad\mathbf{preds} & \_ \stackrel{-}{\Rightarrow} \_ : \mathtt{Agents} \times \mathtt{Act} \times \mathtt{Agents} \\
\quad\mathbf{axioms} & p + \lambda = p \\
 & p||\lambda = p \\
 & p + q = q + p \\
 & p||q = q||p \\
 & a.p \stackrel{a}{\Rightarrow} p \\
 & p \stackrel{a}{\Rightarrow} p' \Rightarrow p + q \stackrel{a}{\Rightarrow} p' \\
 & p \stackrel{a}{\Rightarrow} p' \Rightarrow p||q \stackrel{a}{\Rightarrow} p'||q \\
 & p \stackrel{a}{\Rightarrow} p' \wedge q \stackrel{\bar{a}}{\Rightarrow} q' \Rightarrow p||q \stackrel{\tau}{\Rightarrow} p'||q'
\end{array}
$$

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 impose that an action take place instead of another when both choices are available. This 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* $a \neq b \wedge p \stackrel{a}{\Rightarrow} p' \Rightarrow p_{|b} \stackrel{a}{\Rightarrow} p'_{|b}$*, 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 limit 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 must be expressed as a Boolean function, introducing Boolean sort and operations as well, instead than as a predicate.*

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

spec $\mathtt{CCS} =$ **enrich** $\mathtt{CCS\_}$ **by**
opns $\_|\_ : \mathtt{Agents} \times \mathtt{Act} \to \mathtt{Agents}$
axioms $p \overset{a}{\Rightarrow} p' \wedge \mathtt{different}(a,b) \Rightarrow p_{|b} \overset{a}{\Rightarrow} p'_{|b}$

Negation of equality (and more in general of atomic sentences) is not the unique kind of logical expression that we may want to express but are 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 $\mathtt{CCS}_1 =$ **enrich** $\mathtt{CCS}$ **by**
preds $\_\mathtt{must\ do}\_ : \mathtt{Agents} \times \mathtt{Act}$
axioms $p \ \mathtt{must\ do}\ a \Leftrightarrow (\forall p' : \mathtt{Agents}.\forall b : \mathtt{Act}.p \overset{b}{\Rightarrow} p' \Rightarrow a = b)$

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 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 and this situation is unavoidable in the phase of the requirement *specification of a data type, when the implementative details are still to be fixed, and are left underspecified for the* design *phase to complete. 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 we can as well specify as follows the same predicate, exploiting the information we have on the definition of the transition predicate.*

spec $\mathtt{CCS}_2 =$ **enrich** $\mathtt{CCS}$ **by**
preds $\_\mathtt{must\ do}\_ : \mathtt{Agents} \times \mathtt{Act}$
axioms $a.p \ \mathtt{must\ do}\ a$
axioms $p \ \mathtt{must\ do}\ a \wedge q \ \mathtt{must\ do}\ a \Rightarrow p||q \ \mathtt{must\ do}\ a$
axioms p must do $a \wedge q \ \mathtt{must\ do}\ a \Rightarrow p + q \ \mathtt{must\ do}\ a$

Notice, however, that the latter specification has the expected initial model, but has models that do not satisfy the specification $\mathtt{CCS}_1$ *(as some agent in them has more transitions than those strictly required by the specification* `CCS`*) and moreover the definition of the predicate* `must do` *is correct only for this description of the agents, but should be updated if, for instance, a new combinator would be added for agents. Therefore,* $\mathtt{CCS}_1$ *is much more flexible and can be used during the requirement phase, while* $\mathtt{CCS}_2$ *can be adopted as a solution only for the design phase.*

Let us see one more example of (initial) specification of data types, that is 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 3.1.15 *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 and 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 3.1.14. Therefore we assume that a Boolean function representing equality is implemented in our specification of locations.*

spec $Sp_L =$
sorts $\mathtt{loc}, \mathtt{bool} \ldots$
opns $T, F : \to \mathtt{bool}$
$eq : \mathtt{loc} \times \mathtt{loc} \to \mathtt{bool} \ldots$

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.

spec $\mathtt{Stores}_1 =$ **enrich** Sp_L, Sp_V **by**
sorts `store`
opns $\mathtt{empty} : \to \mathtt{store}$
$\mathtt{update} : \mathtt{store} \times \mathtt{loc} \times \mathtt{value} \to \mathtt{store}$
axioms $\mathtt{update}(\mathtt{update}(s,x,v_1),x,v_2) = \mathtt{update}(s,x,v_2)$
$eq(x,y) = F \Rightarrow \mathtt{update}(\mathtt{update}(s,x,v_1),y,v_2) =$
$\mathtt{update}(\mathtt{update}(s,y,v_2),x,v_1)$

The initial model of $\mathtt{Stores}_1$ *has the intended stores as elements of sort* `store`*, but no tools to retrieve the stored values. In order to introduce an operation* $\mathtt{retrieve} : \mathtt{store} \times \mathtt{loc} \longrightarrow \mathtt{value}$*, 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 $\mathtt{Stores}_2 =$ **enrich** $\mathtt{Stores}_1$ **by**
opns $\mathtt{retrieve} : \mathtt{store} \times \mathtt{loc} \to \mathtt{value}$
axioms $\mathtt{retrieve}(\mathtt{update}(s,x,v),x) = v$

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 $\texttt{Stores}_2$*, 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 uncorrect 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, 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, the need for conditional axioms was soonly recognized [91]. Conditional axioms with predicates (but without full equality) are also used in logic programming and Prolog [62]. A combination of both points of view, that is conditional axioms with equations *and* predicates, is done in the Eqlog language [42], see also [82].

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* [34]

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 parted in three categories:

- 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);
- a partial recursive function with non-recursive domain, for example an interpreter of a programming language, has to be specified;
- a usual total abstract data type, like the positive natural numbers, is enriched by a partial function, like the subtraction; in particular the inverse of some constructor is axiomatically introduced. 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* and 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 on 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 constructed 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 constructor and result in an error when applied to the bounds of the subrange), or bounded *stacks* (where *push*ing an element on a full *stack* yield error).

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

3.2.1 Programming on Data Types

Consider the following situation: We have defined a data type by a minimal set of (total) functions providing a denotation for each element of our data type, and 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*.

$$\begin{array}{ll} \textbf{spec}\ \mathtt{Nat} = & \\ \quad \textbf{sorts} & \mathtt{nat} \\ \quad \textbf{opns} & \mathtt{zero}: \to \mathtt{nat} \\ & \mathtt{succ}: \mathtt{nat} \to \mathtt{nat} \end{array}$$

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`.

Since we are within a total approach, 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 `prec(zero)` denotes an *erroneous* element `err`, and hence all other errors, for instance those due to the application of the operations of the data type to the error, should be identified with `err`.

It is worth to stress that both possibilities, that we will see more in details in the next paragraphs, correspond to an entirely static type checking, where errors are completely predictable and can be detected at the signature level. Thus partial recursive functions with non-recursive domains are not supported.

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

$$\begin{array}{ll} \textbf{spec}\ \mathtt{Nat}^{\mathtt{pos}} = & \\ \quad \textbf{sorts} & \mathtt{nat}, \mathtt{pos} \\ \quad \textbf{opns} & \mathtt{zero}: \to \mathtt{nat} \\ & \mathtt{succ}: \mathtt{nat} \to \mathtt{pos} \\ & \mathtt{prec}: \mathtt{pos} \to \mathtt{nat} \\ & e: \mathtt{pos} \to \mathtt{nat} \\ \quad \textbf{axioms} & \forall x: \mathtt{nat}.\mathtt{prec}(\mathtt{succ}(x)) = x \end{array}$$

Thus, for instance, the expression $0 + 1 + 1 - 1$ is represented by the term `prec(succ(`e`(succ(zero))))`, while we would have expected `prec(succ(succ(zero)))`. Therefore, it is much preferable to enrich the theory by explicitly allowing the *sub-sort* case, 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 sub-sort declaration and 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 will be more extensively presented in the next subsection and we refer to it for a formal presentation of the order-sorted approach. Here we informally use the following specification

$$\begin{array}{ll} \textbf{spec}\ \mathtt{Nat}^{osa} = & \\ \quad \textbf{sorts} & \mathtt{pos} \leq \mathtt{nat} \\ \quad \textbf{opns} & \mathtt{zero}: \to \mathtt{nat} \\ & \mathtt{succ}: \mathtt{nat} \to \mathtt{pos} \\ & \mathtt{prec}: \mathtt{pos} \to \mathtt{nat} \\ \quad \textbf{axioms} & \forall x: \mathtt{nat}.\mathtt{prec}(\mathtt{succ}(x)) = x \end{array}$$

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 $\mathtt{Nat}^{\mathtt{pos}}$ 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 `prec(prec(succ(succ(zero))))` is incorrect, because the result type of `prec` is `nat`, while `prec` expects an argument of sort `pos`, even if `prec(succ(succ(zero)))` can be deduced equal to `succ(zero)` and intuitively should, hence, 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.

Dynamic treatment of errors: retracts and sort-constraints In OBJ3, this problem is solved by automatically adding retracts $r : \mathtt{nat} > \mathtt{pos}$ which can be removed using retract equations

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

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

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

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

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

which can be seen as an error message. The problem is, however, that this term introduces a new error element, thus changing the semantics of the specification. In [45] 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 retracts. 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 in course of definition within the CoFI initiative. An alternative approach uses (conditional) sort constraints [72, 93]. 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 $s' \leq s$[1].

Now let us add a sort constraint

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

to the above specification.

If now well-typedness of terms is defined by also taking sort constraints into account [40], the term

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

is well-typed because prec(succ(succ(n))) is of sort pos by the sort constraint. This means that type checking now can generate proof obligations which in general can be resolved only dynamically by doing some theorem proving. 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 there definedness can be checked only dynamically with theorem proving.

[1] Actually, s and s' only need to belong to the same connected component

So parsing of terms becomes quite complex. Some work in this direction has be done in [93].

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 may be worthwhile to combine the order-sorted and the partial approach[2], see [79].

The second point is that $\mathtt{Nat}^{osa}$ ($\mathtt{Nat}^{\mathtt{pos}}$) can be hardly said *modular* w.r.t. Nat, though our aim was just to add a derived function to the already defined data type Nat. Indeed, the functionality of succ has been changed to use that function in order to build the subsort pos. Of course, instead of changing the functionality of succ, we could add a new symbol $f\colon \mathtt{nat} \longrightarrow \mathtt{pos}$[3] with the axiom $\forall x:\mathtt{nat}.f(x) = \mathtt{succ}(x)$. But in this way, even if formally the original specification is preserved, the constructor succ has become redundant. [4]

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. This can be done with sort constraints, which allow to specify subsorts to consists of all those values which 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 $\mathtt{Nat}^2$ =
sorts $\mathtt{even} \leq \mathtt{nat}$
$\_\ \mathtt{mod}\ 2 : \mathtt{nat} \rightarrow \mathtt{nat}$
$\_\ \mathtt{div}\ 2 : \mathtt{even} \rightarrow \mathtt{nat}$
$\mathtt{zero\ mod\ 2 = zero}$
$\mathtt{succ(zero)\ mod\ 2 = succ(zero)}$
$\forall x:\mathtt{nat}.\mathtt{succ}(\mathtt{succ}(x))\ \mathtt{mod}\ 2 = x\ \mathtt{mod}\ 2$
$\forall x:\mathtt{nat}.x:\mathtt{even} \Leftrightarrow x\ \mathtt{mod}\ 2 = \mathtt{zero}$
$\mathtt{zero\ div\ 2 = zero}$
$\forall x:\mathtt{even}.\mathtt{succ}(\mathtt{succ}(x))\ \mathtt{div}\ 2 = \mathtt{succ}(x\ \mathtt{div}\ 2)$

[2] In the algebraic language that is currently being defined within the CoFIforum, indeed, subsorting is combined with and given semantics through partialfirst-order structures. We refer to http://www.brics.dk/Projects/CoFI/DesignProposals/Summary for further details on such setting.

[3] In an order-sorted approach overloading of symbols is allowed and even encouraged; in particular functions with the same name must coincide on the "common part" of their domain, that is if $g\colon s \longrightarrow s$ and $g\colon s' \longrightarrow s'$ are both declared with $s \leq s'$, then the interpretation of the first g is the restriction of the interpretation of the second g. Thus in this case we should use again the name succ instead of f and automatically get the axiom to hold.

[4] In most order-sorted approaches, data types which are defined by constructors always are equipped with one subsort for each constructor. Thus, the problems presented here do not arise in that case. But they persist for functions that are not constructors

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

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

together with an ordinary axiom

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

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 the 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.

Error elements The second 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 (s) interpreting the error(s) have to be added to the models of the original specification (see [83] for an argumentation against the introduction of error elements in basic types by the hierarchic building of 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 [39], 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, $x * \mathtt{err} = \mathtt{err}$, on zero and the standard basis of the inductive product definition, $x * \mathtt{zero} = \mathtt{zero}$, on err, we deduce $\mathtt{zero} = \mathtt{err}$.

Thus in [39] 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, but using predicates and conditional formulae, their usage is improved, because the implementation of the Boolean type, with its connectives, is not needed anymore. However, 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.

For instance one of the more classical example, 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.

$$
\begin{array}{lll}
\textbf{spec}\ \mathtt{Nat}^{\star} = & & \\
 & \textbf{sorts} & \mathtt{nat} \\
 & \textbf{opns} & \mathtt{zero}, \mathtt{err} : \to \mathtt{nat} \\
 & & \mathtt{succ} : \mathtt{nat} \to \mathtt{nat} \\
 & & \mathtt{plus}, \mathtt{minus}, \mathtt{times} : \mathtt{nat} \times \mathtt{nat} \to \mathtt{nat} \\
 & \textbf{preds} & \mathtt{OK}, \mathtt{IsErr} : \mathtt{nat} \\
 & \textbf{axioms} & \mathtt{OK}(\mathtt{zero}) \\
 & & \mathtt{OK}(x) \Rightarrow \mathtt{OK}(\mathtt{succ}(x)) \\
 & & \mathtt{IsErr}(x) \Rightarrow \mathtt{IsErr}(\mathtt{succ}(x)) \\
 & & \mathtt{IsErr}(\mathtt{err}) \\
 & & \mathtt{OK}(x) \Rightarrow \mathtt{plus}(x, \mathtt{zero}) = x \\
 & & \mathtt{OK}(x) \wedge \mathtt{OK}(y) \Rightarrow \mathtt{plus}(x, \mathtt{succ}(y)) = \mathtt{succ}(\mathtt{plus}(x, y)) \\
 & & \mathtt{IsErr}(x) \Rightarrow \mathtt{IsErr}(\mathtt{plus}(x, y)) \\
 & & \mathtt{IsErr}(x) \Rightarrow \mathtt{IsErr}(\mathtt{plus}(y, x)) \\
 & & \mathtt{OK}(x) \Rightarrow \mathtt{times}(x, \mathtt{zero}) = \mathtt{zero} \\
 & & \mathtt{OK}(x) \wedge \mathtt{OK}(y) \Rightarrow \mathtt{times}(x, \mathtt{succ}(y)) = \mathtt{plus}(x, \mathtt{times}(x, y)) \\
 & & \mathtt{IsErr}(x) \Rightarrow \mathtt{IsErr}(\mathtt{times}(x, y)) \\
 & & \mathtt{IsErr}(x) \Rightarrow \mathtt{IsErr}(\mathtt{times}(y, x)) \\
 & & \mathtt{OK}(x) \Rightarrow \mathtt{minus}(x, \mathtt{zero}) = x \\
 & & \mathtt{OK}(x) \wedge \mathtt{OK}(y) \Rightarrow \mathtt{minus}(\mathtt{succ}(x), \mathtt{succ}(y)) = \mathtt{minus}(x, y) \\
 & & \mathtt{IsErr}(\mathtt{minus}(\mathtt{zero}, \mathtt{succ}(x))) \\
 & & \mathtt{IsErr}(x) \Rightarrow \mathtt{IsErr}(\mathtt{minus}(x, y)) \\
 & & \mathtt{IsErr}(x) \Rightarrow \mathtt{IsErr}(\mathtt{minus}(y, x)) \\
 & & \mathtt{IsErr}(x) \Rightarrow x = \mathtt{err}
\end{array}
$$

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

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 [18], where both the elements of algebras and the terms are labeled to capture the difference between errors and exceptions, or the *clean algebras* in [38], where an order-sorted approach is adopted to catalogue the elements of algebras). 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*.

3.2.2 Partial constructors

A quite different problem from that introduced by the last section is the definition of data types whose constructor themselves are partial functions.

A paramount example of this case is in the formal languages field. Indeed, each production of a contex-free grammar, with the form $s ::= w_1 s_1 \ldots s_n w_{n+1}$, 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 $s_1 \times \ldots \times s_n$ into s. Thus, contex-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. For instance, let us consider the specification of bounded stacks, where pushing an element on a stack is correct only if the stack in not "too large".

Example 3.2.1 *Let us define the data type of stacks of elements with no more than a prefixed number of items. This specification is, of course, parametric on the definition of the element data type, that we assume given by the specification* `Elem`*, with principal sort* `elem`*, and is based also on a specification of natural number with an order relation* $\leq$ *on them, in order to define the depth of a stack.*

spec $\mathtt{Nat}_{\leq}$ =
  sorts $\mathtt{nat}$
  opns $\mathtt{zero}\colon \to \mathtt{nat}$
    $\mathtt{succ}\colon \mathtt{nat} \to \mathtt{nat}$
  preds $\_ \leq \_\colon \mathtt{nat} \times \mathtt{nat} \ldots$
  axioms $\leq (\mathtt{zero}, x)$
    $x \leq y \Rightarrow \mathtt{succ}(x) \leq \mathtt{succ}(y)$

Now the point to fix is the value of an application of `push` *on a stack already full. Indeed, being in a total approach, it should yields a value.*

Notice that the order-sorted style is not convenient in this case, as the domain of the `push` *function is not intuitively built by a total constructor.*

Indeed, the natural constructor for `push` *is* `push` *itself, but only a fixed number 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 $i = 1 \ldots n-1$, 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.*

Then we can consider a classical total approach where pushing too many elements on the same stack results in one error *element.*

spec $\mathtt{BoundedStacks}$ = **enrich** $\mathtt{Nat}_{\leq}, \mathtt{Elem}$ **by**
  sorts $\mathtt{bstack}$
  opns $\mathtt{max}\colon \to \mathtt{nat}$
  opns $\mathtt{empty}, \mathtt{err}\colon \to \mathtt{bstack}$
    $\mathtt{Bpush}\colon \mathtt{elem} \times \mathtt{bstack} \to \mathtt{bstack}$
    $\mathtt{depth}\colon \mathtt{bstack} \to \mathtt{nat}$
  preds $\mathtt{OK}, \mathtt{Is\_Err}\colon \mathtt{bstack}$
  axioms $\mathtt{max} = \ldots$
    $\mathtt{OK}(\mathtt{empty})$
    $\mathtt{depth}(\mathtt{empty}) = \mathtt{zero}$
    $\mathtt{depth}(\mathtt{err}) = \mathtt{succ}(\mathtt{max})$
    $\mathtt{depth}(s) \leq \mathtt{max} \Rightarrow \mathtt{OK}(\mathtt{Bpush}(x, s))$
    $\mathtt{OK}(\mathtt{Bpush}(x, s)) \Rightarrow \mathtt{depth}(\mathtt{Bpush}(x, s)) = \mathtt{succ}(\mathtt{depth}(s))$
    $\mathtt{succ}(\mathtt{max}) \leq \mathtt{depth}(s) \Rightarrow \mathtt{Is\_Err}(\mathtt{Bpush}(x, s))$
    $\mathtt{Is\_Err}(s) \Rightarrow s = \mathtt{err}$

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

Even if obvious operations should be added to this type, the standard error propagation problem would arise. In particular, for operations with result type `nat` *arbitrary correct values should picked up as results on* `err`*, as in the forth axiom in the case of* `depth(err)`*, or an "error" should be added to the* `nat`*, violating the modularity principle.*

With sort constraints, one has to specify bounded stacks as a subsort of ordinary stack. Note that the sort of bounded stacks here contains those stack on which a push can safely be performed staying within the bound. Thus the largest bounded stack is *not* in the sort `bstack`.

spec $\mathtt{BoundedStacks}$ = **enrich** $\mathtt{Nat}_{\leq}, \mathtt{Elem}$ **by**
  sorts $\mathtt{bstack} \leq \mathtt{stack}$
  opns $\mathtt{max}\colon \to \mathtt{nat}$
  opns $\mathtt{empty}\colon \to \mathtt{bstack}$
    $\mathtt{push}\colon \mathtt{elem} \times \mathtt{stack} \to \mathtt{stack}$

$\mathtt{push}: \mathtt{elem} \times \mathtt{bstack} \to \mathtt{stack}$
$\mathtt{depth}: \mathtt{stack} \to \mathtt{nat}$
axioms $\mathtt{max} = \ldots$
$\mathtt{depth}(\mathtt{empty}) = \mathtt{zero}$
$\forall x : \mathtt{elem}, s : \mathtt{stack}.\mathtt{depth}(\mathtt{push}(x, s)) = \mathtt{succ}(\mathtt{depth}(s))$
$\forall s : \mathtt{stack}.s : \mathtt{bstack} \Leftrightarrow \mathtt{depth}(s) \leq \mathtt{max}$

Although 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, we cannot delay the semantics of the data until every detail has been decided, because of methodological reasons.

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 stack, that cannot be furtherly refined. For instance all errors have been identified and 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 stack. 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 Structures

When considering 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 [33] 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 by Burmeister [22] and others. In a two-valued logic of partial functions, formulae which contain some nondenoting term are interpreted as false. Later on, we briefly

sketch a three-valued logic where the valuations for formulae may be partial as well, that is, a formula contain some nondenoting 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 the (a) category 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 framework, we will try to clarify the basic nature of our category, discussing 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 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 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 3.3.1 *A* partial signature $\Sigma = \langle S, \Omega, \Psi, \Pi \rangle$ *consists of a set* S*, denoted by* $S(\Sigma)$*, of* sorts*, two componentwise disjoint* $S^* \times S$*-sorted families* Ω *and* Ψ*, respectively denoted by* $\Omega(\Sigma)$ *and* $\Psi(\Sigma)$*, of* total *and* partial function symbols *and an* S^+*-sorted family* Π*, denoted by* $\Pi(\Sigma)$*, of* predicate symbols.

Given a partial signature $\Sigma = \langle S, \Omega, \Psi, \Pi \rangle$ *and an* S*-sorted family* X *of variables, the* terms *of sort* s *are the carrier of sort* s *of the term algebra over the signature* Σ' *and* X*, for* $\Sigma' = (S, \Omega \cup \Psi)$*, as introduced in Definition 1.4.1.* □

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 3.3.2 *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 $\Sigma_{\mathtt{Nat}}$ =
sorts `nat`
opns `zero`: $\to$ `nat`
`succ`: `nat` $\to$ `nat`
`plus`, `times`: `nat` $\times$ `nat` $\to$ `nat`
popns `prec`: `nat` $\mathrel{-\!\!\circ\!\!\!\to}$ `nat`
`minus`, `div`, `mod`: `nat` $\times$ `nat` $\mathrel{-\!\!\circ\!\!\!\to}$ `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 $\Sigma_{\mathtt{Elem}}$ describing the type `elem` *of the elements for the stack, is the following.*

sig $\Sigma_{\mathtt{Stack}}$ = **enrich** $\Sigma_{\mathtt{Elem}}$ **by**
sorts `stack`
opns `empty`: $\to$ `stack`
`push`: `elem` $\times$ `stack` $\to$ `stack`
popns `pop`: `stack` $\mathrel{-\!\!\circ\!\!\!\to}$ `stack`
`top`: `stack` $\mathrel{-\!\!\circ\!\!\!\to}$ `elem`
preds `is_in`: `elem` $\times$ `stack`
`is_empty`: `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 3.3.3 *For each partial function $g\colon X \mathrel{-\!\!\circ\!\!\!\to} Y$, we will denote by $dom\ \ g$ its* domain, *that is the subset of X defined by $dom\ \ g = \{x \mid x \in X \text{ and } g(x) \in Y\}$.*

Exercise 3.3.4 *Generalize the notion of signature morphism for partial signature.*

As the interpretations of function symbols in Ψ 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 undistinguishable), 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 $e \stackrel{\mathrm{E}}{=} e'$ (*existential equality*) to state that both sides denote the same value, $e \stackrel{\mathrm{W}}{=} e'$ (*weak equality*) to state that if both sides denote a value, then the two values coincide, and $e \stackrel{\mathrm{S}}{=} e'$ (*strong equality*) to state that either both sides denote the same value or both do not denote any value. Thus, in particular $e \stackrel{\mathrm{E}}{=} e$, 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 $e \stackrel{\mathrm{E}}{=} e$ or as $(\exists x. e \stackrel{\mathrm{S}}{=} x)$ with x of sort s non free in e and then using $e\downarrow$ as syntactic sugar for the above mentioned logical formulas we have the following table.

using:	$\stackrel{\mathrm{E}}{=}$ becomes	$\stackrel{\mathrm{W}}{=}$ becomes	$\stackrel{\mathrm{S}}{=}$ becomes
$e \stackrel{\mathrm{E}}{=} e'$	$e \stackrel{\mathrm{E}}{=} e'$	$(e\downarrow \wedge e'\downarrow) \supset e \stackrel{\mathrm{E}}{=} e'$	$(e\downarrow \vee e'\downarrow) \supset e \stackrel{\mathrm{E}}{=} e'$
$e \stackrel{\mathrm{S}}{=} e'$	$e\downarrow \wedge e \stackrel{\mathrm{S}}{=} e'$	$(e\downarrow \wedge e'\downarrow) \supset e \stackrel{\mathrm{S}}{=} e'$	$e \stackrel{\mathrm{S}}{=} e'$

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. Indeed in any trivial first-order structure with singleton carriers, all weak equalities are true, disregarding the definedness of the involved expressions. But using weak equality and definedness assertions it is possible to represent both existential and strong equalities, as follows.

$\stackrel{\mathrm{E}}{=}$ becomes	$\stackrel{\mathrm{W}}{=}$ becomes	$\stackrel{\mathrm{S}}{=}$ becomes
$e\downarrow \wedge e'\downarrow \wedge e \stackrel{\mathrm{W}}{=} e'$	$e \stackrel{\mathrm{W}}{=} e'$	$(e\downarrow \vee e'\downarrow) \supset (e\downarrow \wedge e'\downarrow \wedge e \stackrel{\mathrm{W}}{=} e')$

Definition 3.3.5 *Given a partial signature $\Sigma = \langle S, \Omega, \Psi, \Pi \rangle$, a* partial Σ-structure *A is a triple consisting of*

- *an S-sorted family $|A|$ of* carriers;

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

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* $h_s\colon |A|_s \longrightarrow |B|_s$ *s.t.*

- $h_s(f_A(a_1,\dots,a_n)) \stackrel{\mathrm{E}}{=} f_B(h_{s_1}(a_1),\dots,h_{s_n}(a_n))$ *for all* $f\colon s_1 \times \cdots \times s_n \longrightarrow s \in \Omega$ *and all* $a_i \in |A|_{s_i}$ *for* $i = 1,\dots,n$;
- *if* $g_A(a_1,\dots,a_n)\downarrow$, *then* $h_s(g_A(a_1,\dots,a_n)) \stackrel{\mathrm{E}}{=} g_B(h_{s_1}(a_1),\dots,h_{s_n}(a_n))$ *for all* $g\colon s_1 \times \cdots \times s_n \longrightarrow\!\!\!\!\!\circ\;\; s \in \Psi$ *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* $P\colon s_1 \times \cdots \times s_n \in \Pi$ *and all* $a_i \in |A|_{s_i}$ *for* $i = 1,\dots,n$.

The category $\mathrm{Mod}(\Sigma)$ *has partial* Σ*-structures as objects and homomorphism of partial* Σ*-structures as arrows, with the obvious composition and identities.* $\square$

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

- the S-sorted family $|A|$ of carriers;
- for each $f\colon s_1 \times \cdots \times s_n \longrightarrow s \in \Omega$ a total function $f_A\colon |A|_{s_1} \times \cdots \times |A|_{s_n} \longrightarrow |A|_s$, the *interpretation* of f in A;
- for each $g\colon s_1 \times \cdots \times s_n \longrightarrow\!\!\!\!\!\circ\;\; s \in \Psi$ a partial function $g_A\colon |A|_{s_1} \times \cdots \times |A|_{s_n} \longrightarrow\!\!\!\!\!\circ\;\; |A|_s$, the *interpretation* of g in A;
- for each $P\colon s_1 \times \cdots \times s_n \in \Pi$ a subset P_A of $|A|_{s_1} \times \cdots \times |A|_{s_n}$, representing the *truth values* of P in A.

Notice that, although the interpretation function for partial and total function symbol 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 3.3.2.

Example 3.3.6 *The natural numbers with the "obvious" interpretation of function symbols is a partial* $\Sigma_{\mathtt{Nat}}$*-structure.*

spec $\mathbf{N}$ =
Carriers
$|\mathbf{N}|_{\mathtt{nat}} = \mathbf{N}$
Functions

$$\mathtt{zero}_{\mathbf{N}} = 0$$
$$\mathtt{succ}_{\mathbf{N}}(n) = n+1$$
$$\mathtt{plus}_{\mathbf{N}}(n,m) = n+m$$
$$\mathtt{times}_{\mathbf{N}}(n,m) = n*m$$
$$\mathtt{prec}_{\mathbf{N}}(n) = \begin{cases} n-1, & \text{if } n>0 \\ \text{undefined}, & \text{otherwise} \end{cases}$$
$$\mathtt{minus}_{\mathbf{N}}(n,m) = \begin{cases} n-m, & \text{if } n \geq m \\ \text{undefined}, & \text{otherwise} \end{cases}$$
$$\mathtt{div}_{\mathbf{N}}(n,m) = \begin{cases} n \,\mathbf{div}\, m, & \text{if } m \neq 0 \\ \text{undefined}, & \text{otherwise} \end{cases}$$
$$\mathtt{mod}_{\mathbf{N}}(n,m) = \begin{cases} n \,\mathbf{mod}\, m, & \text{if } m \neq 0 \\ \text{undefined}, & \text{otherwise} \end{cases}$$

Predicates

$$\mathtt{multiple}_{\mathbf{N}} = \{(n,m) \mid n \,\mathbf{mod}\, m \stackrel{\mathrm{E}}{=} 0\}$$

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

spec $\mathbf{N}_E$ =
Carriers
$|\mathbf{N}_E|_{\mathtt{nat}} = \mathbf{N} \cup E$ for $E = \{\mathtt{underflow}, \mathtt{err\_minus}, \mathtt{division\_by\_0}\}$
Functions

$$\mathtt{zero}_{\mathbf{N}_E} = 0$$
$$\mathtt{succ}_{\mathbf{N}_E}(n) = \begin{cases} n+1, & \text{if } n \in \mathbf{N} \\ n, & \text{otherwise} \end{cases}$$
$$\mathtt{plus}_{\mathbf{N}_E}(n,m) = \begin{cases} n+m, & \text{if } n,m \in \mathbf{N} \\ n, & \text{if } n \in E \\ m, & \text{otherwise} \end{cases}$$
$$\mathtt{times}_{\mathbf{N}_E}(n,m) = \begin{cases} n*m, & \text{if } n,m \in \mathbf{N} \\ n, & \text{if } n \in E \\ m, & \text{otherwise} \end{cases}$$
$$\mathtt{prec}_{\mathbf{N}_E}(n) = \begin{cases} n-1, & \text{if } n \in \mathbf{N}, n>0 \\ \mathtt{underflow}, & \text{if } n=0 \\ n, & \text{otherwise} \end{cases}$$
$$\mathtt{minus}_{\mathbf{N}_E}(n,m) = \begin{cases} n-m, & \text{if } n,m \in \mathbf{N}, n \geq m \\ n, & \text{if } n \in E \\ m, & \text{if } n \in \mathbf{N}, m \in E \\ \mathtt{err\_minus}, & \text{otherwise} \end{cases}$$

$$\mathtt{div}_{\mathbf{N}_E}(n,m) = \begin{cases} n \ \mathbf{div}\ m, & \text{if } n,m \in \mathbf{N}, m \neq 0 \\ n, & \text{if } n \in E \\ m, & \text{if } n \in \mathbf{N}, m \in E \\ \mathtt{division\_by\_0}, & \text{otherwise} \end{cases}$$

$$\mathtt{mod}_{\mathbf{N}_E}(n,m) = \begin{cases} n \ \mathbf{mod}\ m, & \text{if } n,m \in \mathbf{N}, m \neq 0 \\ n, & \text{if } n \in E \\ m, & \text{if } n \in \mathbf{N}, m \in E \\ \mathtt{division\_by\_0}, & \text{otherwise} \end{cases}$$

Predicates

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

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

Exercise 3.3.7 *Following the guideline of the previous example, define several $\Sigma_{\mathtt{Stack}}$-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 $g_A(a_1, \ldots, a_n) \downarrow$ can be dropped or substituted by $g_B(h_{s_1}(a_1), \ldots, h_{s_n}(a_n)) \downarrow$), that are used in literature (see e.g. [22, 84]). The definition adopted here guarantees that initial (free) models (if any) in a class are *minimal*, following the *no-junk & no-confusion* principle from [71].

Exercise 3.3.8 *Generalize the notion of reduct for partial first-order structures.*

It is worth noting that 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); however the usual inductive definition of term evaluation is not a homomorphism, disregarding the interpretation of predicates, because it is, in general, a *partial* function. Thus, a different notion of homomorphism has to be introduced to capture term evaluations.

Definition 3.3.9 *Let $\Sigma = \langle S, \Omega, \Psi, \Pi\rangle$ be a partial signature, A and B be partial Σ-structures, and X be an S-sorted family of variables.*

A strict partial *homomorphism from A into B is an S- sorted family h of partial functions $h_s: |A|_s \multimap |B|_s$ s.t.*

- $h_s(f_A(a_1, \ldots, a_n)) \stackrel{\mathrm{S}}{=} f_B(h_{s_1}(a_1), \ldots, h_{s_n}(a_n))$ *for all $f \in \Omega_{w,s} \cup \Psi_{w,s}$, where $w = s_1 \times \cdots \times s_n$, and all $a_i \in |A|_{s_i}$ for $i = 1, \ldots, n$;*

- *if $h_{s_i}(a_i) \downarrow$ for $i = 1, \ldots, n$ and $(a_1, \ldots, a_n) \in P_A$, then $(h_{s_1}(a_1), \ldots, h_{s_n}(a_n)) \in P_B$ for all $P: s_1 \times \cdots \times s_n \in \Pi$ and all $a_i \in |A|_{s_i}$ for $i = 1, \ldots, n$.*

A strict homomorphism *(called* closed *in [22]) is a strict partial homomorphism which happens to be total (i.e., h_s is a total function for all $s \in S$) and for which $(h_{s_1}(a_1), \ldots, h_{s_n}(a_n)) \in P_B$ implies $(a_1, \ldots, a_n) \in P_A$ for all $P: s_1 \times \cdots \times s_n \in \Pi$ and all $a_i \in |A|_{s_i}$ for $i = 1, \ldots, n$.*

A term Σ-structure *over X, consists of the (total) term algebra $T_\Sigma(X)$ over $(S, \Omega \cup \Psi)$ and an interpretation $P_{T_\Sigma(X)} \subseteq |T_\Sigma(X)|_{s_1} \times \cdots \times |T_\Sigma(X)|_{s_n}$ of each $P: s_1 \times \cdots \times s_n \in \Pi$.*

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

A variable valuation *ν for X in A is any S-sorted family of partial functions $\nu_s: X_s \multimap |A|_s$; given a variable valuation ν for X in A, the* term evaluation *$\nu^\#: T_\Sigma(X) \multimap A$ is the strict partial homomorphism inductively defined by:*

- $\nu^\#_s(x) \stackrel{\mathrm{S}}{=} \nu(x)$ *for all $x \in X_s$ and all $s \in S$;*

- $\nu^\#_s(f_A(t_1, \ldots, t_n)) \stackrel{\mathrm{S}}{=} f_B(\nu^\#_{s_1}(t_1), \ldots, \nu^\#_{s_n}(t_n))$ *for all $f \in \Omega_{w,s} \cup \Psi_{w,s}$, where $w = s_1 \times \cdots \times s_n$, and all $t_i \in |T_\Sigma(X)|_{s_i}$ for $i = 1, \ldots, n$;*

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

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

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

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

Lemma 3.3.10 *Let $\Sigma = \langle S, \Omega, \Psi, \Pi\rangle$ be a partial signature, A be a partial Σ-structures, X be an S-sorted family of variables, and ν be a valuation for X in A.*

- *For each term t, if $\nu^\#(t) \downarrow$, then $h(\nu^\#(t)) \stackrel{\mathrm{E}}{=} (h \cdot \nu)^\#(t)$ for all homomorphisms $h: A \longrightarrow B$.*

- *For each valuation ν' for X in A, if $\nu(x) \stackrel{\mathrm{W}}{=} \nu'(x)$ for all variables $x \in X$, then $\nu^\#(t) \stackrel{\mathrm{W}}{=} \nu'^\#(t)$ for all terms t.* □

Notation 3.3.11 *In the sequel let us fix a partial signature* $\Sigma = \langle S, \Omega, \Psi, \Pi \rangle$.
□

From the definition of homomorphism, a notion of sub-object immediately follows, as domain of a monomorphism.

Proposition 3.3.12 *Let* A *and* B *be partial* Σ*-structures. A homomorphism of partial* Σ*-structures* $h: A \longrightarrow B$ *is a monomorphism, that is* $h \cdot h_1 = h \cdot h_2$ *implies* $h_1 = h_2$ *for all* $h_1, h_2: A_0 \longrightarrow A$, *if and only if* h_s *is injective for all* $s \in S$.

Proof. *It is immediate to see that if* h_s *is injective for all* $s \in S$ *then* $h \cdot h_1 = h \cdot h_2$ *for some homomorphisms* $h_1, h_2: A_0 \longrightarrow A$ *implies* $h_1 = h_2$.

Vice versa let us assume that $h \cdot h_1 = h \cdot h_2$ *implies* $h_1 = h_2$ *for all homomorphisms* $h_1, h_2: A_0 \longrightarrow A$ *and show that* h_s *is injective for all* $s \in S$. *Let us assume by contradiction that an* $\overline{s} \in S$ *exists s.t.* $h_{\overline{s}}$ *is not injective, i.e. s.t. there are two different elements, say* a_1 *and* a_2, *in* $|A|_{\overline{s}}$ *s.t.* $h_{\overline{s}}(a_1) = h_{\overline{s}}(a_2)$. *Let us denote by* A_0 *the partial* Σ*-structure defined by:*

- $|A_0|$ *are the carriers of the total term algebra* $T_{\Sigma'}(\{x\})$ *over* $\Sigma' = \langle S, \Omega \rangle$ *and one variable* x *of sort* $\overline{s}$*;*
- *for each* $f: s_1 \times \cdots \times s_n \longrightarrow s \in \Omega$ *the total function* f_{A_0} *is the interpretation of* f *in* $T_\Sigma(\{x\})$*;*
- *for each* $g: s_1 \times \cdots \times s_n \not\longrightarrow s \in \Psi$ *the partial function* g_{A_0} *is totally undefined;*
- *for each* $P: s_1 \times \cdots \times s_n \in \Pi$ *the subset* P_A *is empty.*

Then the total Σ'*-term evaluations* $\nu_1^\#$ *and* $\nu_2^\#$ *induced respectively by* $\nu_1(x) = a_1$ *and* $\nu_2(x) = a_2$, *are different homomorphisms of partial* Σ*-structures, by definition, but* $h \cdot \nu_1^\# = h \cdot \nu_2^\#$. □

Thus a sub-object is, up to isomorphism, a subset of the carriers, inheriting the interpretation of total functions from the original Σ-structure and with partial functions and predicates possibly weakened; this justifies the following definition.

Definition 3.3.13 *Let* A *be a partial* Σ*-structure; then a partial* Σ*-structure* A_0 *is a* weak substructure *(a* subobject*) of* A *iff*

- $|A_0| \subseteq |A|$*;*

- $f_A(a_1, \ldots, a_n) \overset{\mathrm{E}}{=} f_{A_0}(a_1, \ldots, a_n)$ *for all* $f: s_1 \times \cdots \times s_n \longrightarrow s \in \Omega$ *and all* $a_i \in |s_i|_{A_0}$ *for* $i = 1, \ldots, n$*;*
- *if* $g_{A_0}(a_1, \ldots, a_n) \downarrow$, *then* $g_{A_0}(a_1, \ldots, a_n) \overset{\mathrm{E}}{=} g_A(a_1, \ldots, a_n)$ *for all* $g: s_1 \times \cdots \times s_n \not\longrightarrow s \in \Psi$ *and all* $a_i \in |s_i|_{A_0}$ *for* $i = 1, \ldots, n$*;*
- $P_{A_0} \subseteq P_A$ *for all* $P: s_1 \times \cdots \times s_n \in \Pi$.

A weak substructure Σ*-structure* A_0 *of* A *is a* substructure *(a* regular subobject*) of* A *iff*

- $g_{A_0}(a_1, \ldots, a_n) \overset{\mathrm{S}}{=} g_A(a_1, \ldots, a_n)$ *for all* $g: s_1 \times \cdots \times s_n \not\longrightarrow s \in \Psi$ *and all* $a_i \in |s_i|_{A_0}$ *for* $i = 1, \ldots, n$*;*
- $(a_1, \ldots, a_n) \in P_{A_0}$ *iff* $(a_1, \ldots, a_n) \in P_A$ *for all* $P: s_1 \times \cdots \times s_n \in \Pi$ *and all* $a_i \in |s_i|_{A_0}$ *for* $i = 1, \ldots, n$.

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 substructures is said to be* reachable. □

Notice that substructures as given by the above definition correspond to *closed* substructures in [22].

Proposition 3.3.14 *A weak substructure is a substructure iff the embedding is a strict homomorphism.*

Proof. *It is straightforward from the definition of substructure and strict homomorphism.* □

Thus, while many different weak substructures of one Σ-structure exist sharing the same carriers, as partial function and predicate interpretations can be weakened, the carriers completely determine the substructure having them. Moreover, each subset of the carriers closed under functional application obviously defines a substructure.

Example 3.3.15 *Let us consider the following homogeneous signature*

sig $\Sigma =$
  sorts s
  opns $c: \to s$
    $f: s \to s$
  popns $pc: \not\to s$
    $g: s \times s \not\to s$
  preds $P: s \times s \times s$

and the following Σ*-structure* A

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

Then any subset of the range $\{0, \dots, \mathtt{max}\}$ 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:

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

spec $A_2 =$
Carriers
$|A_2|_s = \{0\}$
Functions
$c_{A_2} = 0$
$f_{A_2}(x) = x$
$pc_{A_2} = \text{undefined}$
$g_{A_2}(x,y) = \text{undefined}$
Predicates
$P_{A_2} = \{(0,0,0)\}$

and moreover the two substructures are not related by homomorphisms in either way. There does not exist a 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 $\{0, \dots, \mathtt{max}\}$ including 0 and max *defines a* unique *substructure A_X of A, consisting of:*

spec $A =$
Carriers
$|A|_s = X$
Functions
$c_A = 0$
$f_A(x) = x$
$pc_A = \mathtt{max}$
$g_A(x,y) = \begin{cases} x, & \text{if } x = y \\ \text{undefined}, & \text{otherwise} \end{cases}$
Predicates
$P_A = \{(x,y,z) \mid x = y \text{ or } x = z \text{ or } y = z, x, y, z \in X\}$

Notice that the image of a homomorphism is not, in general, a substructure, because the interpretation of a partial function in the target can be more defined than in the source.

Exercise 3.3.16 *Prove that the image of a homomorphism is a weak substructure of the homomorphism target and show an example of homomorphism whose image is not a substructure of the target. Moreover show that the image of a partial strict homomorphism is a 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 substructure including its image.

Lemma 3.3.17 *Let A_0 be a substructure of a partial Σ-structure A and $h\colon B \longrightarrow A$ be a homomorphism s.t. the image of B is an S-indexed subset of A_0. Then $h\colon B \longrightarrow A_0$ is a homomorphism.*

Proof. *For all $f\colon s_1 \times \cdots \times s_n \longrightarrow s \in \Omega$ and all $b_i \in |B|_{s_i}$ for $i = 1, \dots, n$, as h is a homomorphism from B into A, we have $h_s(f_B(b_1, \dots, b_n)) \stackrel{\mathrm{E}}{=} f_A(h_{s_1}(b_1), \dots, h_{s_n}(b_n))$ and, as A_0 is a substructure of A, $f_{A_0}(h_{s_1}(b_1), \dots, h_{s_n}(b_n)) \stackrel{\mathrm{E}}{=} f_A(h_{s_1}(b_1), \dots, h_{s_n}(b_n))$.*

Analogously for all $g\colon s_1 \times \cdots \times s_n \longrightarrow\!\!\!\circ\, s \in \Psi$ and all $b_i \in |B|_{s_i}$ for $i = 1, \dots, n$, as h is a homomorphism from B into A, if $g_B(b_1, \dots, b_n) \downarrow$, then $h_s(g_B(b_1, \dots, b_n)) \stackrel{\mathrm{E}}{=} g_A(h_{s_1}(b_1), \dots, h_{s_n}(b_n))$ and, as A_0 is a substructure of A, $g_{A_0}(h_{s_1}(b_1), \dots, h_{s_n}(b_n)) \stackrel{\mathrm{S}}{=} g_A(h_{s_1}(b_1), \dots, h_{s_n}(b_n))$. Thus $h_s(g_B(b_1, \dots, b_n)) \stackrel{\mathrm{E}}{=} g_{A_0}(h_{s_1}(b_1), \dots, h_{s_n}(b_n))$.

Finally for all $P\colon s_1 \times \cdots \times s_n \in \Pi$ and all $b_i \in |B|_{s_i}$ for $i = 1, \dots, n$, as h is a homomorphism from B into A, if $(b_1, \dots, b_n) \in P_B$, then $(h_{s_1}(b_1), \dots, h_{s_n}(b_n)) \in P_A$; moreover $h_{s_i}(b_i) \in |A_0|_{s_i}$ and hence, as A_0 is a substructure of A, $(h_{s_1}(b_1), \dots, h_{s_n}(b_n)) \in P_{A_0}$.

Therefore h is a homomorphism of partial Σ-structures from B into A_0.
$\square$

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

Exercise 3.3.18 *Prove that if the valuation of a family of variables is contained in the carriers of a substructure, the evaluation of any term w.r.t. such a valuation in the Σ-structure and in the substructure are equal.*

Corollary 3.3.19 *A Σ-structure is term-generated if and only if it is reachable (i.e. it does not have proper substructures).*

Proof. *Since substructures are closed w.r.t. term evaluation, the image of $eval_A$ is contained in all substructures of A. Thus if $eval_A$ is surjective A is contained in all its substructures, that is, it does not have proper substructures.*

Moreover, it is immediate to verify that the image of a strict homomorphism is a substructure. Therefore the image of $eval_A$ is a substructure of A; hence if A does not have proper substructures, then $eval_A$ is surjective.
$\square$

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

Proposition 3.3.20 *Let $h, h': A \longrightarrow B$ be parallel homomorphisms of partial Σ-structures; then the equalizer of h and h' is the (embedding of the) substructure E of A whose carriers are defined by $| E |_s = \{a \mid a \in | A |_s$ and $h_s(a) \stackrel{\mathrm{E}}{=} h'_s(a)\}$ for all $s \in S$ (into A).*

Proof. *Let us first show that the carriers of E actually describe a substructure of A, i.e. that they are closed under function application.*

- *Let us consider $a_i \in | E |_{s_i}$ for $i = 1, \dots, n$; then $h_{s_i}(a_i)$ and $h'_{s_i}(a_i)$ denote the same value b_i and hence, by definition of homomorphism, $h_s(f_A(a_1, \dots, a_n)) \stackrel{\mathrm{E}}{=} f_B(b_1, \dots, b_n) \stackrel{\mathrm{E}}{=} h'_s(f_A(a_1, \dots, a_n))$. Therefore $f_A(a_1, \dots, a_n) \in | E |_s$.*

- *Analogously, let us consider $a_i \in | E |_{s_i}$ for $i = 1, \dots, n$ s.t. $g_A(a_1, \dots, a_n) \downarrow$. Then $h_{s_i}(a_i)$ and $h'_{s_i}(a_i)$ denote the same value b_i and hence $h_s(g_A(a_1, \dots, a_n)) \stackrel{\mathrm{E}}{=} g_B(b_1, \dots, b_n) \stackrel{\mathrm{E}}{=} h'_s(g_A(a_1, \dots, a_n))$, as $g_A(a_1, \dots, a_n) \downarrow$. Therefore if $g_A(a_1, \dots, a_n) \downarrow$, then $g_A(a_1, \dots, a_n) \in | E |_s$.*

Thus E is a partial Σ-structure and by definition the embedding $e : E \hookrightarrow A$ is an equalizing homomorphism; hence we only have to show that the universal property holds for e.

Let us assume that $h \cdot k = h' \cdot k$ for some homomorphism $k: K \longrightarrow A$; then $h \cdot k(a) = h' \cdot k(a)$ for all $a \in | K |_s$ and hence $k_s(a) \in | E |_s$, i.e. k_s is a total function from $| K |_s$ into $| E |_s$. Thus, by Lemma 3.3.17, k is a homomorphism from K into E and hence it is the required unique factorization of itself through e. $\square$

Thus the domains of equalizers are substructures[5], that justifies the equivalent notation "regular subobject" for substructures.

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

Proposition 3.3.21 *Let $h: A \longrightarrow B$ be a homomorphism of partial Σ-structures. If B is generated by h (regarded as a valuation of the family $|A|$ of variables), then h is an epimorphism[6], that is $k \cdot h = k' \cdot h$ implies $k = k'$ for all $k, k': B \longrightarrow C$.*

Proof. *Let us assume that $k \cdot h = k' \cdot h$ for some homomorphisms $k, k': B \longrightarrow C$. Then h equalizes k and k' and hence, by Proposition 3.3.20, its image is a subfamily of the equalizer E of k and k'. But, since B is generated by h, the functional closure of the image of h in B, that is the smallest substructure including the image itself, is B itself. Therefore the equalizer of k and k' is B, that is k and k' coincide.* $\square$

[5] 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.

[6] Assuming that substructures coincide with regular subobjects, it is easy to show that this condition is necessary, too. Indeed, calling B_0 the smallest substructure including the image of h, two parallel homomorphisms $k, k': B \longrightarrow C$ exist s.t. the embedding of B_0 into B is their equalizer, because substructures are regular subobjects, and obviously $k \cdot h = k' \cdot h$, as h factorizes through the embedding. Thus $B_0 \neq B$ implies $k \neq k'$, that is h is not epi.

It is worth noting that bijective homomorphisms are not required to be isomorphisms, that is their inverse may do not exist. Consider, indeed, the signature Σ with one sort, no total function, one partial constant and no predicates 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 monomorphisms are not required to be sections (that is their left inverse may do not exist) and epimorphisms are not required to be retractions (that is their right inverse may do not exist).

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

Definition 3.3.22 *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 $i = 1, \ldots, n$, then $f_A(a_1, \ldots, a_n) \equiv_s f_A(a'_1, \ldots, a'_n)$ for all $f: s_1 \times \cdots \times s_n \longrightarrow s \in \Omega$* (total function closure)*;*
- *if $a_i \equiv_{s_i} a'_i$ for $i = 1, \ldots, n$ and both $g_A(a_1, \ldots, a_n) \equiv_s g_A(a_1, \ldots, a_n)$ and $g_A(a'_1, \ldots, a'_n) \equiv_s g_A(a'_1, \ldots, a'_n)$, then $g_A(a_1, \ldots, a_n) \equiv_s g_A(a'_1, \ldots, a'_n)$ for all $g: s_1 \times \cdots \times s_n \not\longrightarrow s \in \Psi$* (partial function weak closure)*.*

Given a congruence $\equiv$ on a partial Σ-structure A the domain *of $\equiv$ is the S-family $\{a \mid a \in |A|_s$ 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:*

- *$|A/\equiv|_s = \{[a] \mid a \equiv_s a\}$ for all $s \in S$;*
- *$f_{A/\equiv}([a_1], \ldots, [a_n]) = [f_A(a_1, \ldots, a_n)]$, for all $f: s_1 \times \cdots \times s_n \longrightarrow s \in \Omega$ and all $[a_i] \in |s_i|_{A/\equiv}$ for $i = 1, \ldots, n$;*
- *$g_{A/\equiv}(x_1, \ldots, x_n) \stackrel{\mathrm{E}}{=} [g_A(a_1, \ldots, a_n)]$ if $a_i \in x_i$ exist for $i = 1, \ldots, n$ s.t. $g_A(a_1, \ldots, a_n) \equiv_s g_A(a_1, \ldots, a_n)$; otherwise it is undefined, for all $g: s_1 \times \cdots \times s_n \not\longrightarrow s \in \Psi$ and all $x_i \in |s_i|_{A/\equiv}$ for $i = 1, \ldots, n$;*

- *$(x_1, \ldots, x_n) \in P_{A/\equiv}$ iff $(a_1, \ldots, a_n) \in P_A$ for some $a_i \in x_i$ for $i = 1, \ldots, n$, for all $P: s_1 \times \cdots \times s_n \in \Pi$ and all $x_i \in |s_i|_{A/\equiv}$ for $i = 1, \ldots, n$.*

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 *Ker(h) of h is the total congruence on A defined by a Ker(h) a' iff $h(a) = h(a')$ for all $a, a' \in |A|_s$ and all $s \in S$.* □

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

Proposition 3.3.23 *Given a homomorphism of partial Σ-structures h from A into B, there exists a unique $h_{Ker(h)}: A/Ker(h) \longrightarrow B$ s.t. $h = h_{Ker(h)} \cdot nat_{Ker(h)}$.*

Proof. *By definition of $A/Ker(h)$ such a $h_{Ker(h)}$ has to be given by $h_{Ker(h)}([a]) = h(a)$. It is a well defined total function, by definition of kernel, and satisfies the conditions of homomorphism, by definition of the function interpretation in $A/Ker(h)$. Thus, it is a homomorphism and hence it is the unique factorization of h through $nat_{Ker(h)}$.* □

Definition 3.3.24 *Let $\mathcal{A}$ be a class of partial first-order structures over a partial first-order signature $\Sigma = \langle S, \Omega, \Psi, \Pi \rangle$ 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 ν for X in A a unique homomorphism $h_\nu: F \longrightarrow A$ exists s.t. $\nu = h_\nu \cdot e$.*

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

Lemma 3.3.25 *Let $\mathcal{A}$ be a class of partial first-order structures over a partial first-order signature $\Sigma = \langle S, \Omega, \Psi, \Pi \rangle$ closed under substructures. Then a free first-order structure for an S-sorted family X of variables in $\mathcal{A}$, if any, is generated by X.*

Proof. *Let us assume that $\mathcal{A}$ has a free first-order structure for X, i.e. that there exist $F \in \mathcal{A}$ and a total valuation e for X in F s.t. for all $A \in \mathcal{A}$ and all total valuations ν for X in A a unique homomorphism $h_\nu \colon F \longrightarrow A$ exists s.t. $\nu = h_\nu \cdot e$. Let us denote by F_X the image of e, that is a substructure of F because of Exercise 3.3.16 and hence belongs to $\mathcal{A}$, and by i_X its embedding into F.*

Since F is free, there exists a unique $h_e \colon F \longrightarrow F_X$ s.t. $e = h_e \cdot e$. Therefore $e = h_e \cdot i_X \cdot e$ and, since the identity of F is the unique map s.t. $e = id_F \cdot e$, $h_e \cdot i_X = id_F$.

Let us consider a generic element a of F_X. By definition of F_X some term $t \in T_\Sigma(X)$ exists s.t. $e^{\#}(t) \stackrel{\mathrm{E}}{=} a$; therefore, by Lemma 3.3.10, $h_e(a) \stackrel{\mathrm{E}}{=} h_e(e^{\#}(t)) \stackrel{\mathrm{E}}{=} (h_e \cdot e)^{\#}(t)$ and, by definition of h_e, $(h_e \cdot e)^{\#}(t) \stackrel{\mathrm{E}}{=} e^{\#}(t) \stackrel{\mathrm{E}}{=} a$. Thus, $h_e(a) \stackrel{\mathrm{E}}{=} a$ for all a and hence $i_X \cdot h_e = id_{F_X}$.

Therefore h_e is the inverse of i_X; hence $F = F_X$, that is F is term-generated by X. $\square$

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 formulas, having three different kinds of atomic formulas representing respectively existential, weak and strong equalities between terms, besides atomic predicate formulas.

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 3.1.14. 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 formulas, which eventually serve as axioms in specifications.

Definition 3.3.26 First-order formula
Let $\Sigma = \langle S, \Omega, \Psi, \Pi \rangle$ be a signature. We inductively define for all S-sorted sets X in parallel the set $Form(\Sigma, X)$ of Σ-formulas in variables X. $Form(\Sigma, X)$ is the least S-sorted set containing

- $t_1 \stackrel{e}{=} t_2$ *for* $t_1, t_2 \in |T_\Sigma(X)|_s$
- $P(t_1, \ldots, t_n)$ *for* $P \colon s_1 \times \cdots \times s_n \in \Pi$ *and* $t_i \in |T_\Sigma(X)|_{s_i}$, $i = 1, \ldots, n$
- 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 3.3.27 First-order axiom
A first-order axiom *over a signature Σ is a pair (X, φ), written $X.\varphi$, where $\varphi \in Form(\Sigma, X)$.*

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

Definition 3.3.28 *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 $\varphi \in Form(\Sigma, X)$, the set $FV(\varphi)$ of free variables of φ *is the least set $X' \subseteq X$ such that already $\varphi \in Form(\Sigma, X')$.*

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. We base the definition of satisfaction of a first-order axiom in a Σ-structure on partial valuations, but we want to quantify over the defined only, so quantification is treated by extension of valuations which have to be defined for the quantified variables.

Definition 3.3.29 Satisfaction
Satisfaction of a formula $\varphi \in Form(\Sigma, X)$ by a (possibly partial) valuation $\nu\colon X \longrightarrow |A|$ is defined inductively over the structure of φ:

- $\nu \Vdash_\Sigma t_1 \overset{e}{=} t_2$ *iff* $\nu^\#(t_1) \overset{\mathrm{E}}{=} \nu^\#(t_2)$
- $\nu \Vdash_\Sigma P(t_1, \ldots, t_n)$ *iff* $\nu^\#(t_1)\downarrow$ *and* $\cdots$ *and* $\nu^\#(t_n)\downarrow$ *and* $(\nu^\#(t_1), \ldots, \nu^\#(t_n)) \in P_A$
- *not* $\nu \Vdash_\Sigma F$
- $\nu \Vdash_\Sigma (\varphi \wedge \psi)$ *iff* $\nu \Vdash_\Sigma \varphi$ *and* $\nu \Vdash_\Sigma \psi$
- $\nu \Vdash_\Sigma (\varphi \Rightarrow \psi)$ *iff* $\nu \Vdash_\Sigma \varphi$ *implies* $\nu \Vdash_\Sigma \psi$
- $\nu \Vdash_\Sigma (\forall Y.\varphi)$ *iff for all valuations* $\xi\colon X \cup Y \longrightarrow |A|$ *which*
  - *extend ν on $X\backslash Y$ (i.e. $\xi(x) \overset{\mathrm{S}}{=} \nu(x)$ for all $x \in X_s \backslash Y_s, s \in S$) 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 ν on $X\backslash Y$ only, variables in Y are treated as fresh variables: their value under ν is disregarded within φ.

A partial Σ-structure A satisfies *a first-order axiom $X.\varphi$* w.r.t. total valuations *(written $A \models^t_\Sigma X.\varphi$), if all total valuations $\nu\colon X \longrightarrow |A|$ satisfy φ.*

A partial Σ-structure A satisfies *a first-order axiom $X.\varphi$* w.r.t. partial valuations *(written $A \models^p_\Sigma X.\varphi$), if all (partial of total) valuations $\nu\colon X \longrightarrow |A|$ satisfy φ.*

A first-order axiom $X.\varphi$ is called *closed*, if $X = \emptyset$. 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 3.3.30 *Prove that for a partial Σ-structure A and $\varphi \in Form(\Sigma, X)$*

$$A \models^t_\Sigma X.\varphi \text{ if and only if } A \models^t_\Sigma \emptyset.\forall X.\varphi \text{ if and only if } A \models^p_\Sigma \emptyset.\forall X.\varphi$$

The counterexample $A \models^p_\Sigma \emptyset.\forall x : s.x \overset{e}{=} x$ but $A \not\models^p_\Sigma \{x : s\}.x \overset{e}{=} x$ shows that open formulas are interpreted by $\models^p$ quite differently.

Proposition 3.3.31 *Let $X \subseteq Y$ be two S-sorted variable systems and $\varphi \in Form(\Sigma, X) \subseteq Form(\Sigma, Y)$ be a first-order formula. Then for any partial Σ-structure A,*

$$A \models^p_\Sigma Y.\varphi \text{ if and only if } A \models^p_\Sigma X.\varphi$$

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

Proof. Since the free variables of φ are already contained in X, an easy induction over the structure of φ shows that for a valuation $\xi\colon Y \longrightarrow A$, $\xi \Vdash_\Sigma \varphi$ iff $\xi|_X \Vdash_\Sigma \varphi$, where $\xi|_X\colon X \longrightarrow A$ is the restriction of ξ to X. Now any valuation $\nu\colon X \longrightarrow A$ can be extended to a valuation $\xi\colon Y \longrightarrow A$ with $\xi|_X = \nu$ by just taking ξ to be undefined on $Y \backslash X$. Thus $A \models^p_\Sigma Y.\varphi \Leftrightarrow A \models^p_\Sigma X.\varphi$ follows.

Now the counterexample for $\models^t$: Take the signature

sig $\Sigma =$
  sorts $s\ s'$
  opns $a, b\colon \to s$

and the partial Σ-structure

spec $A =$
  Carriers
    $|A|_s = \{0, 1\}$
    $|A|_{s'} = \emptyset$
  Functions
    $a_A = 0$
    $b_A = 1$

Then $A \models^t_\Sigma \{x : s'\}.a = b$, since there is no total valuation $\nu\colon \{x : s'\} \longrightarrow A$, but $A \not\models^t_\Sigma \emptyset.a = b$ □

Definition 3.3.32 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 $M \models^t_\Sigma \varphi$ (resp. $M \models^p_\Sigma \varphi$), if for all total (resp. total or partial) valuations $\nu\colon X \cup \bigcup_{Y.\psi \in M} Y \longrightarrow |A|$ into partial Σ-structures A we have:*

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

Proposition 3.3.31 can be easily extended to semantical consequence:

Proposition 3.3.33 *Let $X \subseteq Y$ be to S-sorted variable systems and $M \cup \{\varphi\} \subseteq Form(\Sigma, X) \subseteq Form(\Sigma, Y)$ be first-order formulas. Then*

$$M \models^p_\Sigma Y.\varphi \text{ if and only if } M \models^p_\Sigma X.\varphi$$

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

The peculiarity of $\models^t$ shown in Propositions 3.3.31 and 3.3.33 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 literature[7] 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 [52] 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 design 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 3.3.30 and Proposition 3.3.31 (the latter together with its companion 3.3.33) 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 formulas to their universal closure (which holds for $\models^t$) and invariance under changes of the variable system (which holds for $\models^p$).[8] While many treatments of partial logics [22, 84] are guided by the former, we prefer the latter, also because of the easier Substitution Lemma for $\models^p$ (see Lemmas 3.3.40 and 3.3.41 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 3.3.34 *Concerning $\models^p_\Sigma$, we may drop the variable system from formulas and understand φ as an abbreviation of $FV(\varphi).\varphi$.*

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

[7] Indeed, the only references developing many-sorted first-order logic with possibly empty carriers we found are [2] and [59].

[8] 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.

Σ-structure A such that $A \models^p_\Sigma \Phi$. $Mod(\langle \Sigma, \Phi \rangle)$ is the class of all models of $\langle \Sigma, \Phi \rangle$.

Definition 3.3.35 *A presentation $\langle \Sigma, \Phi \rangle$ is called* semantically inconsistent, *if* $\mathrm{Mod}(\langle \Sigma, \Phi \rangle)$ *is empty. Otherwise, it is called* semantically consistent.

Proposition 3.3.36 *For Σ-first-order formulas $\varphi_1, \ldots, \varphi_n$, ψ, the following are equivalent:*

1. $M \cup \{\varphi_1, \ldots, \varphi_n\} \models^p_\Sigma \psi$;
2. $M \models^p_\Sigma \varphi_1 \wedge \cdots \wedge \varphi_n \Rightarrow \psi$;
3. $\langle \Sigma, M \cup \{\varphi_1, \ldots, \varphi_n, \neg\psi\}\rangle$ *is inconsistent.* □

Example 3.3.37 *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 behaviour 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 [22] 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. [43]). The second calculus captures $\models^p$ and follows the ideas of Scott [86]. In this calculus, because of Proposition 3.3.33, 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 3.3.38 *A function $\theta: X \longrightarrow \mid T_\Sigma(Y) \mid$ is called a* substitution. *Given a substitution $\theta: X \longrightarrow \mid T_\Sigma(Y) \mid$ and an S-sorted variable system Z, we denote by $\theta \backslash Z: X \cup Z \longrightarrow T_\Sigma(Y \cup Z)$ the substitution being the identity on Z and being θ on $X \backslash Z$.*

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

Definition 3.3.39 *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:

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

The last case, causing $(\forall Z.\varphi)[\theta]$ to be undefined in the case of name clashes, prevents a free variable in $\theta(x)$ 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 [22]:

Lemma 3.3.40 Substitution Lemma for $\models^t$
Let A be a partial Σ-structure, $\nu: Y \longrightarrow A$ be a total valuation, $\theta: X \longrightarrow \mid T_\Sigma(Y) \mid$ be a substitution and $\varphi \in Form(\Sigma, X)$ a formula. Under the conditions that

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

we have

$$\nu^\# \circ \theta \Vdash_\Sigma \varphi \textit{ if and only if } \nu \Vdash_\Sigma \varphi[\theta] \Box$$

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 3.3.41 Substitution Lemma for $\models^p$
Let A be a partial Σ-structure, $\nu: Y \longrightarrow A$ be a (total or partial) valuation, $\theta: X \longrightarrow \mid T_\Sigma(Y) \mid$ be a substitution and $\varphi \in Form(\Sigma, X)$ a formula. Under the condition that $\varphi[\theta]$ is defined, we have

$$\nu^\# \circ \theta \Vdash_\Sigma \varphi \textit{ if and only if } \nu \Vdash_\Sigma \varphi[\theta] \Box$$

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 $\Phi, \Phi_1, \Phi_2, \Phi_3$ be finite sets of formulas in $Form(\Sigma, X)$. Application of substitution to such sets is understood elementwise. We introduce the following rules of derivation:

Assumption

$$\frac{}{\Phi \ \vdash^t_{\Sigma,X} \ \varphi} \quad \varphi \in \Phi$$

∧-introduction

$$\begin{array}{lll} \Phi_1 & \vdash^t_{\Sigma,X} & \varphi \\ \Phi_2 & \vdash^t_{\Sigma,X} & \psi \\ \hline \Phi_1 \cup \Phi_2 & \vdash^t_{\Sigma,X} & (\varphi \wedge \psi) \end{array}$$

∧-left elimination

$$\begin{array}{lll} \Phi & \vdash^t_{\Sigma,X} & (\varphi \wedge \psi) \\ \hline \Phi & \vdash^t_{\Sigma,X} & \varphi \end{array}$$

∧-right elimination

$$\begin{array}{lll} \Phi & \vdash^t_{\Sigma,X} & (\varphi \wedge \psi) \\ \hline \Phi & \vdash^t_{\Sigma,X} & \psi \end{array}$$

Tertium non datur

$$\begin{array}{lll} \Phi_1 \cup \{ \varphi \} & \vdash^t_{\Sigma,X} & \psi \\ \Phi_2 \cup \{ (\varphi \Rightarrow F) \} & \vdash^t_{\Sigma,X} & \psi \\ \hline \Phi_1 \cup \Phi_2 & \vdash^t_{\Sigma,X} & \psi \end{array}$$

Absurdity

$$\begin{array}{lll} \Phi & \vdash^t_{\Sigma,X} & F \\ \hline \Phi & \vdash^t_{\Sigma,X} & \psi \end{array}$$

Cut

$$\begin{array}{lll} \Phi_1 & \vdash^t_{\Sigma,X} & \varphi_1 \wedge \cdots \wedge \varphi_n \Rightarrow \psi_i \\ \Phi_2 & \vdash^t_{\Sigma,Y} & \psi_1 \wedge \cdots \wedge \psi_k \Rightarrow \epsilon \\ \hline \Phi_1 \cup \Phi_2 & \vdash^t_{\Sigma,X\cup Y} & \psi_1 \wedge \cdots \wedge \psi_{i-1} \wedge \varphi_1 \wedge \cdots \wedge \varphi_n \wedge \psi_{i+1} \wedge \ldots \wedge \eta_k \Rightarrow \epsilon \end{array}$$

⇒-introduction

$$\begin{array}{lll} \Phi \cup \{ \varphi_1, \ldots, \varphi_n \} & \vdash^t_{\Sigma,X} & \psi \\ \hline \Phi & \vdash^t_{\Sigma,X} & \varphi_1 \wedge \cdots \wedge \varphi_n \Rightarrow \psi \end{array}$$

∀-elimination

$$\begin{array}{lll} \Phi & \vdash^t_{\Sigma,X} & (\forall Y.\varphi) \\ \hline \Phi & \vdash^t_{\Sigma,X\cup Y} & \varphi \end{array}$$

∀-introduction

$$\begin{array}{lll} \Phi & \vdash^t_{\Sigma,X\cup Y} & \varphi \\ \hline \Phi & \vdash^t_{\Sigma,X} & (\forall Y.\varphi) \end{array} \quad \text{if } Y \cap FV(\Phi) = \emptyset$$

Reflexivity

$$\begin{array}{l} \\ \hline \Phi \vdash^t_{\Sigma,X} \quad x \overset{\mathrm{e}}{=} x \end{array} \quad \text{for } x \in X_s$$

Congruence

$$\begin{array}{lll} \Phi & \vdash^t_{\Sigma,X} & \varphi \\ \hline \Phi & \vdash^t_{\Sigma,X\cup Y} & (\bigwedge_{x \in X_s} x \overset{\mathrm{e}}{=} \theta(x)) \Rightarrow \varphi[\theta] \end{array} \quad \text{for } \theta\colon X \longrightarrow| T_\Sigma(Y) \,| \text{ with } \varphi[\theta]\downarrow$$

Substitution

$$\begin{array}{lll} \Phi & \vdash^t_{\Sigma,X} & \varphi \\ \hline \Phi[\theta] & \vdash^t_{\Sigma,Y} & (\bigwedge_{x \in X_s} D(\theta(x))) \Rightarrow \varphi[\theta] \end{array}$$

for $\theta\colon X \longrightarrow| T_\Sigma(Y) \,|$ with $\varphi[\theta]\downarrow$ and $\Phi[\theta]\downarrow$

Function Strictness

$$\begin{array}{lll} \Phi & \vdash^t_{\Sigma,X} & t_1 \overset{\mathrm{e}}{=} t_2 \\ \hline \Phi & \vdash^t_{\Sigma,X} & D(t) \end{array} \quad t \text{ some subterm of } t_1 \text{ or } t_2$$

Predicate Strictness

$$\begin{array}{lll} \Phi & \vdash^t_{\Sigma,X} & P(t_1, \ldots, t_n) \\ \hline \Phi & \vdash^t_{\Sigma,X} & D(t_i) \end{array} \quad \text{for } P\colon s_1 \times \cdots \times s_n \in \Pi$$

Totality

$$\begin{array}{lll} \Phi & \vdash^t_{\Sigma,X} & \bigwedge_{i=1,\ldots,n} D(t_i) \\ \hline \Phi & \vdash^t_{\Sigma,X} & D(f(t_1, \ldots, t_n)) \end{array} \quad \text{for } f\colon s_1 \times \cdots \times s_n \longrightarrow s \in \Omega$$

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

A *derivation* is a finite sequence of judgements of form $\Phi \vdash^t_{\Sigma,X} \varphi$ 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 judgement $\emptyset \vdash^t_{\Sigma,X} \varphi$.

Burmeister states the following theorem in [22]:

Theorem 3.3.42 *The calculus is sound and complete, i.e.*

$$\Phi \models^t_\Sigma X.\varphi \textit{ if and only if } \Phi \vdash^t_{\Sigma, X \cup \bigcup_{Y.\psi \in \Phi} Y} \varphi$$

Proof. Soundness is shown by an induction on the rules.

The proof of completeness follows the same lines as the usual Henkin style proof for total first-order logic. See [33] for a completeness proof for partial higher-order logic. □

All rules of the calculus up to **Congruence** are taken from a calculus for total first-order logic [49] (except that we index judgements 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 formulas 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 judgement, 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 rules for the defined connectives, quantifiers and equalities, some of which can be found in [49]) 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 [36]. One crucial source of efficiency is the use of unification (i.e. finding a substitution under which two given formulas 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 favour 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

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

$\forall$-introduction

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

Symmetry

$$\frac{}{\Phi \quad \vdash^p_\Sigma \quad x \overset{e}{=} y \Rightarrow y \overset{e}{=} x}$$

Substitution

$$\frac{\Phi \quad \vdash^p_\Sigma \quad \varphi}{\Phi[\theta] \quad \vdash^p_\Sigma \quad \varphi[\theta]}$$

for $\theta\colon X \longrightarrow |T_\Sigma(Y)|$ with $\varphi[\theta]\downarrow$ and $\Phi[\theta]\downarrow$

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

Theorem 3.3.43 *The calculus is sound and complete, i.e.*

$$\Phi \models^p_\Sigma \varphi \text{ if and only if } \Phi \vdash^p_\Sigma \varphi \Box$$

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 $\overset{e}{=}$ 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 [68] 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 [26].

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 $\vdash^{FOL}_{\Sigma}$.

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 [22]. 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 $g(t_1, \dots, t_n)$ is translated to the formula $\alpha(g(t_1, \dots, t_n)) =$

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

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 [86]. Though model categories are not represented faithfully, proof theory is, so it fits for our purposes here.

A PFOL-signature $\Sigma = \langle S, \Omega, \Psi, \Pi \rangle$ is translated to a FOL-signature $\Phi(\Sigma) = (\{ * \}, \Omega' \uplus \Psi', \Pi')$, where Ω' and Ψ' result from Ω and Ψ 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 $s \in S$
- $x_1 \equiv_{s_1} y_1 \wedge \cdots \wedge x_n \equiv_{s_n} y_n \Rightarrow f(x_1, \dots, x_n) \equiv_s f(y_1, \dots, y_n)$ for $f : s_1 \times \cdots \times s_n \longrightarrow s \in \Omega$
- $x_1 \equiv_{s_1} y_1 \wedge \cdots \wedge x_n \equiv_{s_n} y_n \wedge g(x_1, \dots, x_n) \equiv_s g(x_1, \dots, x_n) \Rightarrow g(x_1, \dots, x_n) \equiv_s g(y_1, \dots, y_n)$ for $g : s_1 \times \cdots \times s_n \longrightarrow\!\!\!\!\circ\; s \in \Psi$

- $x_1 \equiv_{s_1} y_1 \wedge \cdots \wedge x_n \equiv_{s_n} y_n \wedge P(x_1, \dots, x_n) \Rightarrow P(y_1, \dots, y_n)$ for $P : s_1 \times \cdots \times s_n \in \Pi$
- $f(x_1, \dots, x_n) \equiv_s f(x_1, \dots, x_n) \Rightarrow x_i \equiv_{s_i} x_i$ for for $f : s_1 \times \cdots \times s_n \longrightarrow s \in \Omega$
- $g(x_1, \dots, x_n) \equiv_s g(x_1, \dots, x_n) \Rightarrow x_i \equiv_{s_i} x_i$ for for $g : s_1 \times \cdots \times s_n \longrightarrow\!\!\!\!\circ\; s \in \Psi$
- $P(x_1, \dots, x_n) \Rightarrow x_i \equiv_{s_i} x_i$ for for $P : s_1 \times \cdots \times s_n \in \Pi$

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

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

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

where $B|_{\Sigma \longrightarrow \Phi(\Sigma)}$ is the reduct of B along the obvious map $\Sigma \longrightarrow \Phi(\Sigma)$, i.e. B interpreted as partial Σ-structure, and the quotient is taken as in Definition 3.3.22.

This translation now has the following crucial properties:

Proposition 3.3.44 *For each Σ-axiom φ and each total valuation $\nu : X \longrightarrow B$ into a $\langle \Phi(\Sigma), C(\Sigma) \rangle$-structure B we have*

$$nat_{\equiv_B} \circ \nu \Vdash^{PFOL}_{\Sigma} \varphi \text{ if and only if } \nu \Vdash^{FOL}_{\Phi(\Sigma)} \alpha_\Sigma(\varphi)$$

Proof. By induction over the structure of φ.

Considering existence equations, we have

$$\begin{array}{ll}
 & \mathrm{nat}_{\equiv_B} \circ \nu \Vdash^{PFOL}_{\Sigma} t_1 \stackrel{\mathrm{e}}{=} t_2 \\
\text{iff} & (\mathrm{nat}_{\equiv_B} \circ \nu)^{\#}(t_1) \stackrel{\mathrm{E}}{=} (\mathrm{nat}_{\equiv_B} \circ \nu)^{\#}(t_2) \\
\text{iff} & \nu^{\#}(t_1) \equiv_{s,B} \nu^{\#}(t_2) \\
\text{iff} & \nu \Vdash^{FOL}_{\Phi(\Sigma)} t_1 \equiv_{s,B} t_2 \\
\text{iff} & \nu \Vdash^{FOL}_{\Phi(\Sigma)} \alpha_\Sigma(t_1 \stackrel{\mathrm{e}}{=} t_2)
\end{array}$$

Here, strictness of $\equiv_B$ is used to show the that definedness condition is equivalent in each line.

Considering universally quantified formulas, we have

$$\begin{array}{ll}
 & \mathrm{nat}_{\equiv_B} \circ \nu \Vdash^{PFOL}_{\Sigma} \forall Y.\varphi \\
\text{iff} & \text{for all } \xi : X \cup Y \longrightarrow \beta_\Sigma(B) \text{ extending } \mathrm{nat}_{\equiv_B} \circ \nu \text{ on } X \setminus Y \\
 & \text{and being defined on } Y,\ \xi \Vdash^{PFOL}_{\Sigma} \varphi \\
\text{iff} & \text{for all } \rho : X \cup Y \longrightarrow B \text{ extending } \nu \text{ on } X \setminus Y \text{ for which} \\
 & \rho(y) \equiv_{s,B} \rho(y) \text{ for all } y \in Y_s, s \in S, \\
 & \mathrm{nat}_{\equiv_B} \circ \rho \Vdash^{PFOL}_{\Phi(\Sigma)} \alpha_\Sigma(\varphi) \\
\text{iff} & \text{for all } \rho : X \cup Y \longrightarrow B \text{ extending } \nu \text{ on } X \setminus Y \text{ for which} \\
 & \rho(y) \equiv_{s,B} \rho(y) \text{ for all } y \in Y_s, s \in S,\ \rho \Vdash^{FOL}_{\Phi(\Sigma)} \alpha_\Sigma(\varphi) \\
\text{iff} & \nu \Vdash^{FOL}_{\Phi(\Sigma)} (\forall Y. \bigwedge_{\{y \in Y_s, s \in S\}} y \equiv_{s,B} y) \Rightarrow \alpha_\Sigma(\varphi) \\
\text{iff} & \nu \Vdash^{FOL}_{\Phi(\Sigma)} \alpha_\Sigma(\forall Y.\varphi)
\end{array}$$

The other cases are treated similarly. □

Proposition 3.3.45 *For each partial Σ-structure A there exists a $\langle \Phi(\Sigma), C(\Sigma)\rangle$-structure B with $\beta_\Sigma(B) = A$, such that for each valuation $\rho : X \longrightarrow A$ there exists a total valuation $\nu : X \longrightarrow B$ with $\rho = nat_{\equiv_B} \circ \nu$. If ρ moreover is total as well, then $\nu(x) \equiv_{s,B} \nu(x)$ for all $x \in X_s, s \in S$.*

Proof. Take

- $|B|_s = |A|_s \uplus \{\perp\}$ $(s \in S)$
- $f_B(a_1, \dots, a_n) = \begin{cases} f_A(a_1, \dots, a_n), & \text{if } a_i \in |A|_{s_i} \\ \perp, & \text{otherwise} \end{cases}$
- $g_B(a_1, \dots, a_n) = \begin{cases} g_A(a_1, \dots, a_n), & \text{if } a_i \in |A|_{s_i} \text{ and } g_A(a_1, \dots, a_n)\downarrow \\ \perp, & \text{otherwise} \end{cases}$
- $P_B(a_1, \dots, a_n)$ iff $a_i \in |A|_{s_i}$ and $P_A(a_1, \dots, a_n)$
- $a_1 \equiv_{s,B} a_2$ if $a_1 = a_2 \in |A|_s$.
- $\nu(x) = \begin{cases} \rho(x), & \text{if } \rho(x)\downarrow \\ \perp, & \text{otherwise} \end{cases}$

□

Theorem 3.3.46 Borrowing of proof calculus from FOL for PFOL

$$\Phi \models^t_\Sigma X.\varphi \textit{ iff } C(\Sigma) \cup \alpha_\Sigma(\Phi) \cup \{ x \equiv_s x \mid x : s \in X \cup \bigcup_{Y.\psi \in \Phi} Y \} \vdash^{FOL}_{\Phi(\Sigma)} \alpha_\Sigma(\varphi)$$

$$\Phi \models^p_\Sigma \varphi \textit{ iff } C(\Sigma) \cup \alpha_\Sigma(\Phi) \vdash^{FOL}_{\Phi(\Sigma)} \alpha_\Sigma(\varphi)$$

Proof. We here only prove the second statement.

	$C(\Sigma) \cup \alpha_\Sigma(\Phi) \vdash^{FOL}_{\Phi(\Sigma)} \alpha_\Sigma(\varphi)$	
iff	$C(\Sigma) \cup \alpha_\Sigma(\Phi) \models^{FOL}_{\Phi(\Sigma)} \alpha_\Sigma(\varphi)$	by soundness and completeness of $\vdash^{FOL}$
iff	for all total valuations $\nu : X \longrightarrow B$ into a $\langle \Phi(\Sigma), C(\Sigma)\rangle$-structure B, $\nu \Vdash^{FOL}_{\Phi(\Sigma)} \alpha_\Sigma(\Phi) \Rightarrow \nu \Vdash^{FOL}_{\Phi(\Sigma)} \alpha_\Sigma(\varphi)$	
iff	for total all valuations $\nu : X \longrightarrow B$ into a $\langle \Phi(\Sigma), C(\Sigma)\rangle$-structure B, $nat_{\equiv_B} \circ \nu \vdash \vdash^{PFOL}_{\Sigma} \Phi \Rightarrow nat_{\equiv_B} \circ \nu \Vdash^{PFOL}_{\Sigma} \varphi$	by Proposition 3.3.44
iff	for all valuations $\rho : X \longrightarrow A$ into a Σ-model A, $\rho \Vdash^{PFOL}_{\Sigma} \Phi \Rightarrow \rho \Vdash^{PFOL}_{\Sigma} \varphi$	by Proposition 3.3.45
iff	$\Phi \models^p_\Sigma \varphi$	□

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 [11].

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 [82]) and conditional term-rewriting techniques (see Chapter 1 by H. Kirchner and [30, 58]).

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 3.3.47 *Let us see as, using the partial framework, the specification of* stacks *with their constructors and selectors becomes easy and elegant, indeed. Let us recall the signature* $\Sigma_{\texttt{Stack}}$ *from Example 3.3.2*

$$
\begin{array}{lll}
\textbf{sig } \Sigma_{\texttt{Stack}} = & \textbf{enrich } \Sigma_{\texttt{Elem}} \textbf{ by} & \\
& \textbf{sorts} & \texttt{stack} \\
& \textbf{opns} & \texttt{empty}\colon \to \texttt{stack} \\
& & \texttt{push}\colon \texttt{elem} \times \texttt{stack} \to \texttt{stack} \\
& \textbf{popns} & \texttt{pop}\colon \texttt{stack} \overset{\circ}{\to} \texttt{stack} \\
& & \texttt{top}\colon \texttt{stack} \overset{\circ}{\to} \texttt{elem} \\
& \textbf{preds} & \texttt{is\_in}\colon \texttt{elem} \times \texttt{stack} \\
& & \texttt{is\_empty}\colon \texttt{stack}
\end{array}
$$

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`

$$
\begin{array}{ll}
\textbf{spec } \texttt{Stack} = & \textbf{enrich } \Sigma_{\texttt{Stack}} \textbf{ by} \\
& \textbf{axioms } \mathrm{pop}(\mathrm{push}(e,s)) \overset{\mathrm{s}}{=} s \\
& \qquad\quad\ \mathrm{top}(\mathrm{push}(e,s)) \overset{\mathrm{s}}{=} e
\end{array}
$$

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*.*

spec $I =$

Carriers

$|I|_{\texttt{elem}} = X$

$|I|_{\texttt{stack}} = X^*$

Functions

$\texttt{empty}_I = \lambda$

$\texttt{push}_I(x,s) = x \cdot s$

$$\texttt{pop}_I(s) = \begin{cases} s', & \text{if } s = x \cdot s' \\ \text{undefined}\,, & \text{otherwise} \end{cases}$$

$$\texttt{top}_I(s) = \begin{cases} x, & \text{if } s = x \cdot s' \\ \text{undefined}\,, & \text{otherwise} \end{cases}$$

It is interesting to note that the specification `Stack` *is the most abstract interpretation of stacks and can be furtherly 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*

$$\texttt{pop}(\texttt{empty}) \overset{s}{=} \texttt{empty}$$

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

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

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

Definition 3.3.48 *A* positive conditional *formula is a well-formed first-order formula* $\varphi \in Form(\Sigma, X)$ *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* $Sp' = (\Sigma, Ax)$ *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 $\Psi(\Sigma) = \emptyset$ 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 $\overset{\mathrm{e}}{=}$.

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 3.3.49 *Show that a specification $Sp = (\Sigma, Ax)$ 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 $j \in 1, \ldots, n$ s.t. ϵ_j is $D(t)$ or $D(t')$;*
- *if ϵ_i is the weak equality $t \stackrel{w}{=} t'$, then there exist $j, k \in 1, \ldots, n$ 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 3.3.50 *The basic specification of non-negative integers is the usual (total) one, given by the absolutely free constructors "zero" and "successor".*

sig $Sp_{\mathtt{Nat}}{}^T =$
  sorts nat
  opns zero: $\to$ nat
    succ: nat $\to$ nat

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

spec $Sp_{\mathtt{Nat}} =$ **enrich** $Sp_{\mathtt{Nat}}{}^T$ **by**
  popns prec: nat $\rightharpoonup$ nat
    minus: nat $\times$ nat $\rightharpoonup$ nat
  axioms $\mathtt{prec}(\mathtt{succ}(x)) \stackrel{s}{=} x$
    $\mathtt{minus}(x, \mathtt{zero}) \stackrel{s}{=} x$
    $\mathtt{minus}(\mathtt{succ}(x), \mathtt{succ}(y)) \stackrel{s}{=} \mathtt{minus}(x, y)$

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* $\mathtt{succ}^k(\mathtt{zero})$*, 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 $\mathtt{succ}^k(\mathtt{zero})$.

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 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 3.3.51 *Let Sp be a positive conditional conditional specification over a partial first-order signature $\Sigma = \langle S, \Omega, \Psi, \Pi \rangle$ 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:*

$(t_1, \ldots, t_n) \in P_{T_\Sigma(X)}$ iff $A \models \forall X. P(t_1, \ldots, t_n)$ 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 3.3.52 *Let $\langle \Sigma, \Phi \rangle$ be a presentation and I an initial model in $Mod(\langle \Phi(\Sigma), C(\Sigma) \cup \alpha(\Phi) \rangle$ Then $\beta(I)$ is an initial model in $Mod(\langle \Sigma, \Phi \rangle)$.*

A sound and 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 formulas (up to translation). This is a particular case of the *borrowing* technique proposed in [26] and a sugared version of the borrowing for PFOL, where also definedness predicates are allowed.

Definition 3.3.53 *Let $\Sigma = \langle S, \Omega, \Psi, \Pi \rangle$ be a partial signature.*

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

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

and Ax^T denote the following set of total conditional formulas on Σ^T:

$$\begin{array}{l}
D_{s_1}(x_1) \wedge \cdots \wedge D_{s_n}(x_n) \Rightarrow D_s(f(x_1, \ldots, x_n)) \\
\qquad \text{for all } f : s_1 \times \cdots \times s_n \longrightarrow s \in \Omega \\
D_s(f(x_1, \ldots, x_n)) \Rightarrow D_{s_i}(x_i) \\
\qquad \text{for all } f : s_1 \times \cdots \times s_n \longrightarrow s \in \Omega \cup \Psi \\
x \stackrel{e}{=}_s y \Rightarrow D_s(x) \\
D_s(x) \Rightarrow x \stackrel{e}{=}_s x \\
D_s(x) \wedge x = y \Rightarrow x \stackrel{e}{=}_s y \\
x \stackrel{e}{=}_s y \Rightarrow x = y \\
x \stackrel{e}{=}_s y \Rightarrow y \stackrel{e}{=}_s x \\
x \stackrel{e}{=}_s y \wedge y \stackrel{e}{=}_s z \Rightarrow x \stackrel{e}{=}_s z
\end{array}$$

- *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 all strictly positive conditional formulas $\varphi = \forall X. \epsilon_1 \wedge \cdots \wedge \epsilon_n \Rightarrow \epsilon_{n+1}$, let $\alpha(\varphi)$ denote the following total conditional axiom:

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

- *For all total first-order structure A modelling (Σ^T, Ax^T), 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 $f : s_1 \times \cdots \times s_n \longrightarrow s \in \Omega$.*
  - *g_B is the restriction of g_A to the carriers of B for all $g : s_1 \times \cdots \times s_n \longrightarrow\!\!\circ\, s \in \Psi$; in particular if $(a_1, \ldots, a_n) \in D_{s_1 A} \times \ldots \times D_{s_n A}$ but $g_A(a_1, \ldots, a_n) \notin D_{sA}$, then $g_B(a_1, \ldots, a_n)$ 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 dropped 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 3.3.54 *Using the notation of Definition 3.3.53, the following* satisfaction condition *holds for all total first-order structure A and all strictly positive conditional formulas φ:*

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

Exercise 3.3.55 *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 $Sp = (\Sigma, A)$ the model class of Sp satisfies a strictly positive conditional formula iff the model class of the total conditional specification $Sp^T = (\Sigma^T, \alpha(A) \cup Ax^T)$ 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 3.3.56 *Let $Sp = (\Sigma, A)$ 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 3.3.53, $\mathrm{Mod}(Sp) \models^t \varphi$ iff $\alpha(A) \cup Ax^T \vdash \alpha(\varphi)$, where $\vdash$ is given in Definition 3.1.7.

Example 3.3.57 *Let us consider again the problem of store specification already presented in Example 3.1.15.*

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 3.1.15. 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:

$$
\begin{array}{l}
\textbf{spec } Sp_L = \\
\quad \textbf{sorts} \quad \texttt{loc}\ldots \\
\quad \textbf{preds} \quad \texttt{AreEqual}, \texttt{AreDifferent}\colon \texttt{loc} \times \texttt{loc}\ldots
\end{array}
$$

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

$$
\begin{array}{ll}
\textbf{spec } \texttt{Stores} = & \textbf{enrich } Sp_L, Sp_V \textbf{ by} \\
\quad \textbf{sorts} & \texttt{store} \\
\quad \textbf{opns} & \texttt{empty}\colon \to \texttt{store} \\
 & \texttt{update}\colon \texttt{store} \times \texttt{loc} \times \texttt{value} \to \texttt{store} \\
\quad \textbf{popns} & \texttt{retrieve}\colon \texttt{store} \times \texttt{loc} \not\to \texttt{value} \\
\quad \textbf{axioms} & \texttt{AreEqual}(x,y) \Rightarrow \texttt{update}(s,x,v) \overset{s}{=} \texttt{update}(s,y,v) \\
 & \texttt{AreEqual}(x,y) \Rightarrow \texttt{update}(\texttt{update}(s,x,v_1),y,v_2) \overset{s}{=} \texttt{update}(s,x,v_2) \\
 & \texttt{AreDifferent}(x,y) \Rightarrow \texttt{update}(\texttt{update}(s,x,v_1),y,v_2) \overset{s}{=} \\
 & \quad \texttt{update}(\texttt{update}(s,y,v_2),x,v_1) \\
 & \texttt{retrieve}(\texttt{update}(s,x,v),x) \overset{s}{=} v
\end{array}
$$

Notice that, thanks to the partial setting, in `Stores` *initial model the application* `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 [84], 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*

$$
\begin{array}{ll}
\textbf{spec } \texttt{Stores}^T = & \textbf{enrich } Sp_L^T, Sp_V^T \textbf{ by} \\
\quad \textbf{sorts} & \texttt{store} \\
\quad \textbf{opns} & \texttt{empty}\colon \to \texttt{store} \\
 & \texttt{update}\colon \texttt{store} \times \texttt{loc} \times \texttt{value} \to \texttt{store} \\
 & \texttt{retrieve}\colon \texttt{store} \times \texttt{loc} \to \texttt{value} \\
\quad \textbf{axioms} & D(\texttt{empty}) \\
 & D(s) \wedge D(x) \wedge D(v) \Rightarrow D(\texttt{update}(s,x,v)) \\
 & D(\texttt{update}(s,x,v)) \Rightarrow D(s) \\
 & D(\texttt{update}(s,x,v)) \Rightarrow D(x) \\
 & D(\texttt{update}(s,x,v)) \Rightarrow D(v) \\
 & D(\texttt{retrieve}(s,x)) \Rightarrow D(s) \\
 & D(\texttt{retrieve}(s,x)) \Rightarrow D(x) \\
 & D(x) \wedge D(y) \wedge D(s) \wedge D(v) \wedge \texttt{AreEqual}(x,y) \Rightarrow \\
 & \quad \texttt{update}(s,x,v) = \texttt{update}(s,y,v) \\
 & D(x) \wedge D(y) \wedge D(s) \wedge D(v_1) \wedge D(v_2) \wedge \texttt{AreEqual}(x,y) \Rightarrow \\
 & \quad \texttt{update}(\texttt{update}(s,x,v_1),y,v_2) = \texttt{update}(s,x,v_2) \\
 & D(x) \wedge D(y) \wedge D(s) \wedge D(v_1) \wedge D(v_2) \wedge \texttt{AreDifferent}(x,y) \Rightarrow \\
 & \quad \texttt{update}(\texttt{update}(s,x,v_1),y,v_2) = \texttt{update}(\texttt{update}(s,y,v_2),x,v_1) \\
 & D(x) \wedge D(s) \wedge D(v) \Rightarrow \texttt{retrieve}(\texttt{update}(s,x,v),x) = v
\end{array}
$$

where Sp_L^T and Sp_V^T are the corresponding translations where for example axioms like `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 3.3.58

$$
\begin{array}{lll}
\textbf{spec}\ \mathtt{BoundedStacks} = & \multicolumn{2}{l}{\textbf{enrich}\ \mathtt{Nat}_{\leq}, \Sigma_{\mathtt{Elem}}\ \textbf{by}} \\
 & \textbf{sorts} & \mathtt{bstack} \\
 & \textbf{opns} & \mathtt{empty}\colon \to \mathtt{stack} \\
 & & \mathtt{max}\colon \to \mathtt{nat} \\
 & \textbf{opns} & \mathtt{empty}\colon \to \mathtt{bstack} \\
 & \textbf{popns} & \mathtt{Bpush}\colon \mathtt{elem} \times \mathtt{stack} \overset{\circ}{\longrightarrow} \mathtt{stack} \\
 & & \mathtt{pop}\colon \mathtt{stack} \overset{\circ}{\longrightarrow} \mathtt{stack} \\
 & & \mathtt{top}\colon \mathtt{stack} \overset{\circ}{\longrightarrow} \mathtt{elem} \\
 & \textbf{axioms} & \mathtt{depth}(\mathtt{empty}) = \mathtt{zero} \\
 & & \mathtt{depth}(s) \leq \mathtt{succ}(\mathtt{max}) \Rightarrow \mathtt{depth}(\mathtt{Bpush}(x,s)) = \mathtt{succ}(\mathtt{depth}(s)) \\
 & & D(\mathtt{Bpush}(x,s)) \Leftrightarrow \mathtt{depth}(s) \leq \mathtt{max} \\
 & & D(\mathtt{Bpush}(x,s)) \Rightarrow \mathtt{pop}(\mathtt{Bpush}(x,s)) \stackrel{\mathrm{e}}{=} s \\
 & & D(\mathtt{Bpush}(x,s)) \Rightarrow \mathtt{top}(\mathtt{Bpush}(x,s)) \stackrel{\mathrm{e}}{=} x
\end{array}
$$

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 state to values).

Example 3.3.59 *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. [7, 9] for an extended treatment 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.*

$$
\begin{array}{lll}
\textbf{spec}\ \mathtt{BExp} = & \multicolumn{2}{l}{\textbf{enrich}\ \mathtt{States}\ \textbf{by}} \\
 & \textbf{sorts} & \mathtt{bool}, \mathtt{BoolExps} \\
 & \textbf{opns} & \mathtt{true}, \mathtt{false}\colon \to \mathtt{bool}\ldots \\
 & \textbf{popns} & \mathtt{BEval}\colon \mathtt{BoolExps} \times \mathtt{state} \overset{\circ}{\longrightarrow} \mathtt{bool}\ldots
\end{array}
$$

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

$$
\begin{array}{lll}
\textbf{spec}\ \mathtt{ImpLang} = & \multicolumn{2}{l}{\textbf{enrich}\ \mathtt{States}, \mathtt{BExp}\ \textbf{by}} \\
 & \textbf{sorts} & \mathtt{commands} \\
 & \textbf{opns} & \mathtt{skip}\colon \to \mathtt{commands}\ldots \\
 & & \mathtt{if\_then\_else}\colon \mathtt{BoolExps} \times \mathtt{commands} \times \mathtt{commands} \to \mathtt{commands} \\
 & & \mathtt{while\_do}\colon \mathtt{BoolExps} \times \mathtt{commands} \to \mathtt{commands} \\
 & & \mathtt{conc}\colon \mathtt{commands} \times \mathtt{commands} \to \mathtt{commands}\ldots \\
 & \textbf{popns} & \mathtt{CEval}\colon \mathtt{commands} \times \mathtt{state} \overset{\circ}{\longrightarrow} \mathtt{state}\ldots \\
 & \textbf{axioms} & \mathtt{CEval}(c,s) \stackrel{\mathrm{e}}{=} s' \Rightarrow \mathtt{CEval}(\mathtt{conc}(c,c'),s) \stackrel{\mathrm{s}}{=} \mathtt{CEval}(c',s') \\
 & & \mathtt{BEval}(b,s) \stackrel{\mathrm{e}}{=} \mathtt{true} \Rightarrow \mathtt{CEval}(\mathtt{if\_then\_else}(b,c,c'),s) \stackrel{\mathrm{s}}{=} \mathtt{CEval}(c,s) \\
 & & \mathtt{BEval}(b,s) \stackrel{\mathrm{e}}{=} \mathtt{false} \Rightarrow \mathtt{CEval}(\mathtt{if\_then\_else}(b,c,c'),s) \stackrel{\mathrm{s}}{=} \mathtt{CEval}(c',s) \\
 & & \mathtt{BEval}(b,s) \stackrel{\mathrm{e}}{=} \mathtt{false} \Rightarrow \mathtt{CEval}(\mathtt{while\_do}(b,c),s) \stackrel{\mathrm{e}}{=} s \\
 & & \mathtt{BEval}(b,s) \stackrel{\mathrm{e}}{=} \mathtt{true} \wedge \mathtt{CEval}(c,s) \stackrel{\mathrm{e}}{=} s' \Rightarrow \\
 & & \quad \mathtt{CEval}(\mathtt{while\_do}(b,c),s) \stackrel{\mathrm{s}}{=} \mathtt{CEval}(\mathtt{while\_do}(b,c),s')
\end{array}
$$

It is worth noting that, as usual in programming languages, the conditional choice `if_then_else` *is* non-strict*, in the sense that* $\mathtt{CEval}(\mathtt{if\_then\_else}(b,c,c'),s)$ *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. [10, 25]).* □

Bibliographical notes

Full many-sorted and order-sorted first order logic was introduced by Oberschelp [81]. Many-sorted logic is studied extensively in [68]. The empty carrier problem, which is ignored by most authors, is treated by Goguen and Meseguer in [43].

Two-valued first-order logic with partial functions and $\models^t$ was introduced by Burmeister [22, 23] and generalized to categorical logic by Knijnenburg and Nordemann [59]. The $\models^p$-logic follows ideas of Scott [86], which are worked out for the classical first-order logic by Moggi [73]. $\models^t$ and $\models^p$ are compared by Beeson [13].

There is a calculus and a Henkin-style completeness theorem for partial higher order logic in [33], Burmeister has a calculus for one-sorted partial logic [22], The translation from partial to total first-order logic is described by Scott [86]. This translation generates partial congruence relations, which can be treated in a way similar to equality with the results of Bachmair and Ganzinger [11].

The restriction to the positive conditional case is studied by Reichel and others in [84, 14, 9].

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 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:

$$B \subseteq S^{\rightarrow} \qquad\qquad s_1, \ldots, s_n\,,\, s \in S^{\rightarrow} \Rightarrow (s_1 \times \cdots \times s_n \rightarrow s) \in S^{\rightarrow}$$

Of course the sort $(s_1 \times \cdots \times s_n \rightarrow s)$ represents the type of functions with arguments in the cartesian product $s_1 \times \cdots \times s_n$ 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 sort must be downward-closed, that is if $(s_1 \times \cdots \times s_n \rightarrow s_{n+1}) \in S$, then all $s_i \in S$.

Since we are interested in the specification of partial functions, the application must be a partial function, even if the other operations of the signature are total (in which case we can speak of *total* specification of *partial* higher-order functions). For instance in the above example of the specification of a kernel of a programming language semantics, the application of commands (respectively Boolean expressions) to their input, the current state, was denoted by `CEval` (`BEval`).

The intuition that the elements of a functional sort $(s_1 \times \cdots \times s_n \rightarrow s_{n+1}) \in S$ 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 case of unary functions) is

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

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

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

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. [7]).

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, in [69, 74, 75] where the same problem is tackled for total types, the same results as for total conditional types are achieved. 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. [9]).

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 [7] for an analysis and exposition of the problem of higher-order partial types.

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 $\mathtt{Bool} =$
sorts $\mathtt{bool}, \mathtt{BoolExps}, \mathtt{dummy}$
opns $\cdot : \to \mathtt{dummy}$
$\mathtt{T}, \mathtt{F} : \to \mathtt{bool}$
$\mathtt{true}, \mathtt{false} : \to \mathtt{BoolExps}$

$\mathtt{BEval} : \mathtt{BoolExps} \times \mathtt{dummy} \to \mathtt{bool}$
$\mathtt{and}, \mathtt{or} : \mathtt{BoolExps} \times \mathtt{BoolExps} \to \mathtt{BoolExps} \ldots$
axioms $\mathtt{BEval}(\mathtt{true}, \cdot) \stackrel{e}{=} \mathtt{T}$
$\mathtt{BEval}(\mathtt{false}, \cdot) \stackrel{e}{=} \mathtt{F}$
$\mathtt{BEval}(x, \cdot) \stackrel{e}{=} \mathtt{F} \Rightarrow \mathtt{and}(x, y) \stackrel{e}{=} \mathtt{false}$
$\mathtt{BEval}(x, \cdot) \stackrel{e}{=} \mathtt{T} \Rightarrow \mathtt{and}(x, y) \stackrel{e}{=} y \ldots$
$\mathtt{BEval}(x, \cdot) \stackrel{e}{=} \mathtt{T} \Rightarrow \mathtt{or}(x, y) \stackrel{e}{=} \mathtt{true}$
$\mathtt{BEval}(x, \cdot) \stackrel{e}{=} \mathtt{F} \Rightarrow \mathtt{or}(x, y) \stackrel{e}{=} y \ldots$

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 non interesting 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 [10], 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 $[1 \ldots n]$ into the carriers, here *partial* n-tuples, that are partial functions from the range $[1 \ldots n]$ 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 $f(a)$ is defined, for some constant a and unary function f, then we have that a is defined or $f(x)$ is defined for whatever value of x (including the undefined). Thus $D(f(a))$ is equivalent to $D(a) \vee D(f(x))$.

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 [10], giving necessary and sufficient conditions for the existence of initial models.

A completely different point of view on non-strictness is presented in [25], 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 $\mathtt{zero} * t$ can be simplified to `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. Assume, indeed, that we have an arbitrary first order-formula φ; then $\imath x : s.\varphi$ (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 $\imath x : s.\varphi$ 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:[9]

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

The description operator is characterized by the axiom

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

which has to be added to the calculus.

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

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

[9]Syntax of terms and formulas as well as their semantics (i.e. $\nu^{\#}$ and $\nu \Vdash \_$) have to be defined in parallel now.

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

In absence of a division operation, the term $\imath z : real.x \stackrel{e}{=} y * z$ is not equivalent to a term without $\imath$. Therefore, Corollary 3.3.19 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 if 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 formulas containing the term. Thus valuations of terms *and* of formulas have to be partial. The latter causes the need for a third truth-value, say $\bot$, which is assigned to formulas containing a non-denoting term. Then, non-strict predicates also may yield $\bot$ when applied to denoting terms. The connectives can be extended to deal with $\bot$ in several ways. A natural choice is Kleene's three-valued logic [57], which is guided by a strictness (resp. continuity) principle (cf. [27]).

$\wedge$	t	$\bot$	f
t	t	$\bot$	f
$\bot$	$\bot$	$\bot$	f
f	f	f	f

$\vee$	t	$\bot$	f
t	t	t	t
$\bot$	t	$\bot$	$\bot$
f	t	$\bot$	f

$\neg$	
t	f
$\bot$	$\bot$
f	t

$\Rightarrow$	t	$\bot$	f
t	t	$\bot$	f
$\bot$	t	$\bot$	$\bot$
f	t	t	t

It is used, for example, in the specification languages SPECTRUM [19] and VDM [56]. 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 when incorporated in the three-valued world. For example, the definedness predicate

D	
t	t
$\bot$	f
f	t

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

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

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 $\bot$.

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

See [27] 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

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{spec}\ ThreeValued =} \\
\textbf{sorts} & \mathtt{Bool3} \\
\textbf{preds} & \stackrel{3}{=} : \mathtt{Bool3} \times \mathtt{Bool3} \\
\textbf{opns} & \mathtt{t}, \mathtt{f}, \bot : \rightarrow \mathtt{Bool3} \\
 & \neg^3 : \mathtt{Bool3} \rightarrow \mathtt{Bool3} \\
 & \wedge^3, logor^3, \Rightarrow^3 : \mathtt{Bool3} \times \mathtt{Bool3} \rightarrow \mathtt{Bool3} \\
\textbf{axioms} & \forall x : \mathtt{Bool3}. x \stackrel{e}{=} \mathtt{t} \vee x \stackrel{e}{=} \mathtt{f} \vee x \stackrel{e}{=} \bot \\
 & \mathtt{t} \vee^3 \bot \stackrel{e}{=} \mathtt{t} \\
 & \ldots \\
 & (x \stackrel{3}{=} y) \stackrel{e}{=} \mathtt{t} \Leftrightarrow x \stackrel{e}{=} y \\
 & (x \stackrel{3}{=} y) \stackrel{e}{=} \mathtt{f} \Leftrightarrow (D(x) \wedge D(y) \wedge \neg x \stackrel{e}{=} y) \\
 & (x \stackrel{3}{=} y) \stackrel{e}{=} \bot \Leftrightarrow (\neg D(x) \vee \neg D(y)) \\
 & \ldots
\end{array}
$$

A quantified formula $\forall X.\varphi$ is expressed by first translating φ inductively to a term $\stackrel{3}{\varphi}$: $\mathtt{Bool3}$ and then taking $\forall \stackrel{3}{X}.\varphi$ to be

$$
\begin{array}{l}
\iota b : \mathtt{Bool3}.\ ((\forall X.\, \stackrel{3}{\varphi} \stackrel{e}{=} \mathtt{t}) \Rightarrow b \stackrel{e}{=} \mathtt{t}) \\
\qquad \wedge((\forall X. \neg \stackrel{3}{\varphi} \stackrel{e}{=} \bot \wedge \exists X.\, \stackrel{3}{\varphi} \stackrel{e}{=} \mathtt{f}) \Rightarrow b \stackrel{e}{=} \mathtt{f}) \\
\qquad \wedge((\exists X.\, \stackrel{3}{\varphi} \stackrel{e}{=} \bot) \Rightarrow b \stackrel{e}{=} \bot)
\end{array}
$$

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

Contents

Bibliography

[1] J. Adámek, H. Herrlich, G. Strecker. *Abstract and Concrete Categories*. Wiley, New York, 1990.

[2] J. Adámek, J. Rosický. *Locally Presentable and Accessible Categories*. Cambridge University Press, 1994.

[3] H. Andréka, I. Németi. Generalization of the concept of variety and quasi-variety to partial algebras through category theory. *Dissertationes Mathematicae* **204**, 1983.

[4] V. Antimirov, A. Degtyarev. Consistency and semantics of equational definitions over predefined algebras. In *CPRS 92*, Lecture Notes in Computer Science. Springer Verlag, 1992.

[5] V. Antimirov, A. Degtyarev. Consistency of equational enrichments. In A. Voronkov, editor., *Logic Programming and Automated Reasoning 92, Lecture Notes in Computer Science/LNAI* **624**, 313–402. Springer Verlag, 1992.

[6] K. R. Apt, E.R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer Verlag, 1991.

[7] E. Astesiano, M. Cerioli. Partial higher-order specifications. *Fundamenta Informaticae* **16**(2), 101–126, 1992.

[8] E. Astesiano, M. Cerioli. Relationships between logical frames. In *Recent Trends in Data Type Specification: 8th Workshop on Specification of Abstract Data Types – Selected Papers*, number 655 in Lecture Notes in Computer Science, 126–143, Berlin, 1993. Springer Verlag.

[9] E. Astesiano, M. Cerioli. Free objects and equational deduction for partial conditional specifications. *Theoretical Computer Science* **152**(1), 91–138, 1995.

[10] E. Astesiano, M. Cerioli. Non-strict don't care algebras and specifications. *Mathematical Structures in Computer Science* **6**(1), 85–125, 1996.

[11] L. Bachmair, H. Ganzinger. Rewrite techniques for transitive relations. In *Proc. 9th IEEE Symposium on Logic in Computer Science*, 384–393. IEEE Computer Society Press, 1994. Short version of TR MPI-I-93-249.

[12] M. Barr, C. Wells. *Toposes, Triples and Theories*, *Grundlehren der mathematischen Wissenschaften* **278**. Springer Verlag, 1985.

[13] M. J. Beeson. Proving programs and programming proofs. In R. B. Marcus et al., editor., *Logic, Methodology and Philosophy of Science VII*, 51–82. North Holland, Amsterdam, 1986.

[14] K. Benecke, H. Reichel. Equational partiality. *Algebra Universalis* **16**, 219–232, 1983.

[15] H.L. Bentley, N. Murthy. Essentially algebraic categories of partial algebras. *Quaestiones Mathematicae 13*, 361–384, 1990.

[16] J.A. Bergstra, J.V. Tucker. Algebraic specifications of computable and semicomputable data types. *Theoretical Computer Science*, 137–181, 1987.

[17] J.A. Bergstra, J.V. Tucker. The inescapable stack: an exercize in algebraic specification with total functions. Technical Report P8804, University of Amsterdam; Programming Research Group, 1988.

[18] G. Bernot, M. Bidoit, C. Choppy. Abstract data types with exception handling: an initial approach based on the distinction between exceptions and errors. *Theoretical Computer Science* **46**(1), 13–45, 1986.

[19] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hußmann, D. Nazareth, F. Regensburger, O. Slotosch, K. Stø len. The requirement and design specification language spectrum: An informal introduction, version 1.0, part I. Technical Report TUM-19311, TU München, 1993.

[20] M. Broy, C. Pair, M. Wirsing. A systematic study of models of abstract data types. *Theoretical Computer Science* **33**, 137–174, 1984.

[21] M. Broy, M. Wirsing. Partial abstract types. *Acta Informatica* **18**, 1982.

[22] P. Burmeister. Partial algebras — survey of a unifying approach towards a two-valued model theory for partial algebras. *Algebra Universalis* **15**, 306–358, 1982.

[23] P. Burmeister. *A Model Theoretic Oriented Approach to Partial Algebras.* Akademie Verlag, Berlin, 1986.

[24] 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.

[25] M. Cerioli. A lazy approach to partial algebras. In G. Reggio E. Astesiano, A. Tarlecki, editors., *Recent Trends in Data Type Specification: 10th Workshop on Types – Selected Papers*, number 906 in Lecture Notes in Computer Science, 188–202, Berlin, 1995. Springer Verlag.

[26] M. Cerioli, J. Meseguer. May i borrow your logic? (transporting logical structures along maps). *Theoretical Computer Science*, 1996. To appear.

[27] J. P. Cleave. *A Study of Logics.* Oxford University Press, 1991.

[28] M. Coste. Localisation, spectra and sheaf representation. In M.P. Fourman, C.J. Mulvey, D.S. Scott, editors., *Application of Sheaves, Lecture Notes in Mathematics* **753**, 212–238. Springer Verlag, 1979.

[29] P.L. Curien. Partiality, cartesian closed categories and toposes. *Information and Computation* **80**, 50–???, 1989.

[30] Nachem Dershowitz, Jean-Pierre Jouannaud. Rewrite systems. In J. van Leeuwen, editor., *Handbook of Theoretical Computer Science*, 243–320. Elsevier Science Publ. B.V., 1990.

[31] H. Ehrig, B. Mahr. *Fundamentals of Algebraic Specifications 1: Equations and Initial semantics, EATCS Monographs on Theoretical Computer Science* **6**. Springer Verlag, New-York, 1985.

[32] H. Ehrig, B. Mahr. *Fundamentals of Algebraic Specifications 2, EATCS Monographs on Theoretical Computer Science* **21**. Springer Verlag, New-York, 1990.

[33] W. A. Farmer. A partial functions version of Church's simple type theory. *Journal of Symbolic Logic* **55**, 1269–1291, 1991.

[34] S. Feferman. A new approach to abstract data types, I informal development. *Mathematical Structures in Computer Science* **2**, 193–229, 1992.

[35] P. Freyd. Aspects of topoi. *Bull. Austral. Math. Soc.* **7**, 1–76, 1972.

[36] D. M. et al. Gabbay. *Handbook of Logic in Artificial Intelligence and Logic Programming, Vol. 1: Logical Foundations.* Oxford Science Publications, 1993.

[37] M. Gogolla. *Über partiell geordnete Sortenmengen und deren Anwendung zur Fehlerbehandlung in abstrakten Datentypen.* PhD thesis, Braunschweig, 1986.

[38] M. Gogolla. On parametric algebraic specifications with clean error handling. In *Proceedings TAPSOFT'87*, number 249 in Lecture Notes in Computer Science, 81–95, Berlin, 1987. Springer Verlag.

[39] J. Goguen, J. Thatcher, Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In R. Yeh, editor., *Current Trends in Programming Methodology*, 80–149. Prentice-Hall, 1976.

[40] J. A. Goguen, J. Meseguer, J.P. Jouannaud. Operational semantics of order-sorted algebra. In Wilfried Brauer, editor., *Proceedings, 1985 International Conference on Automata, Languages and Programming, Lecture Notes in Computer Science* **194**. Springer Verlag, 1985.

[41] J.A. Goguen. Order sorted algebras: Exceptions, error sorts, coercion and overloaded operators. Semantics and theory of computation report no. 14, University of California, Los Angeles., 1978.

[42] J.A. Goguen, J. Meseguer. Eqlog: Equality, types, and generic modules for logic programming. In D. DeGroot, G. Lindstrom, editors., *Logic Programming. Functions, Relations and Equations*, 295–363. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[43] J.A. Goguen, J. Meseguer. Remarks on remarks on many-sorted equational logic. *EATCS Bulletin* **30**, 66–73, 1986.

[44] J.A. Goguen, J. Meseguer. Order-sorted algebra solves the constructor selector, multiple representation and coercion problems. In *Proceedings, Second Symposium on Logic in Computer Science*, 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.

[45] J.A. Goguen, J. Meseguer. Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science* **105**, 217–273, 1992.

[46] J.A. Goguen, J. Thatcher, E. G. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In R. Yeh, editor., *Current Trends in Programming Methodology*, 80–149. Prentice-Hall, 1978.

[47] J.A. Goguen, T Winkler. Introducing OBJ3. Research report SRI-CSL-88-9, Computer Science Lab., SRI International, 1988.

[48] J. W. Gray. Categorical aspects of data type constructors. *Theoretical Computer Science* **50**, 103–135, 1987.

[49] H. Hermes. *Introduction to mathematical logic.* Springer Verlag, 1973.

[50] H. Herrlich, G.E. Strecker. *Category Theory, an Introduction.* Heldermann Verlag, Berlin, 1979.

[51] H.-J. Hoehnke. On partial algebras. In *Universal Algebra (Proc. Coll. Esztergom 1977), Colloq. Math. Soc. J. Bolyai* **29**, 373–412. North Holland, Amsterdam, 1981.

[52] G. Huet, D. Oppen. Equations and rewrite rules: a survey. In *Formal Language Theory: Perspectives and Open Problems.* Academic Press, New York, 1980.

[53] G. Jarzembski. Weak varieties of partial algebras. *Algebra Universalis* **25**, 247–262, 1988.

[54] G. Jarzembski. Programs in partial algebras — a categorical approach. In D.H. et al. Pitt, editor., *Category Theory and Computer Science, Lecture Notes in Computer Science* **530**, 140–150. Springer Verlag, 1991.

[55] G. Jarzembski. Programs in partial algebras. *Theoretical Computer Science* **115**, 131–149, 1993.

[56] C. B. Jones. *Systematc Software Development Using VDM.* Prentice Hall, 1990.

[57] S.C. Kleene. *Introduction to Metamathematics.* North Holland, 1952.

[58] Jan Willem Klop. Term rewriting systems. In S. Abramsky, Dov M. Gabbay, T.S.E. Maibaum, editors., *Handbook of Logic in Computer Science,* volume 2, 1–116. Oxford University Press, 1992.

[59] P. Knijnenburg, F. Nordemann. Partial hyperdoctrines: categorical models for partial function logic and hoare logic. *Mathematical Structures in Computer Science* **4**, 117–146, 1994.

[60] H.J. Kreowski. Partial algebras flow from algebraic specifications. In T. Ottmann, editor., *Proc. ICALP 87, Lecture Notes in Computer Science* **267**, 521–530. Springer Verlag, 1987.

[61] H.J. Kreowski, T Mossakowski. Equivalence and difference of institutions: Simulating horn clause logic with based algebras. To appear.

[62] J.W. Lloyd. *Foundations of Logic Programming.* Springer Verlag, 1987.

[63] S. MacLane. *Categories for the Working Mathematician.* Springer Verlag, 1971.

[64] V. Manca, A. Salibra. Equational calculi for many-sorted algebras with empty carriers. In *Proceedings of Mathematical Foundations of Computer Science 1990*, number 452 in Lecture Notes in Computer Science, Berlin, 1990. Springer Verlag.

[65] V. Manca, A. Salibra. Soundness and completeness of the birkhoff equational calculus for many-sorted algebras with possibly empty carriers. *Theoretical Computer Science* **94**(1), 101–124, 1992.

[66] V. Manca, A. Salibra, G. Scollo. Equational type logic. *Theoretical Computer Science* **77**, 131–159, 1990. Special Issue dedicated to AMAST'89.

[67] V. Manca, A. Salibra, G. Scollo. On the expressiveness of equational type logic. In C.M.I. Rattray, R.G. Clark, editors., *The Unified Computation Laboratory.* Oxford University Press, 1992.

[68] J. Meinke, J.V. Tucker, editors. *Many-sorted Logic and its Applications.* Wiley, 1993.

[69] K. Meinke. Universal algebra in higher types. *Theoretical Computer Science* **100**(2), 385–417, 1992.

[70] J. Meseguer. General logics. In *Logic Colloquium '87*, 275–329, Amsterdam, 1989. North Holland.

[71] J. Meseguer, J.A. Goguen. Initiality, induction and computability. In M. Nivat, J. Reynolds, editors., *Algebraic Methods in Semantics*, 459–540. Cambridge University Press, Cambridge, 1985.

[72] J. Meseguer, J.A. Goguen. Order-sorted algebra solves the constructor selector, multiple representation and coercion problems. *Information and Computation* **103**(1), 114–158, March 1993.

[73] E. Moggi. *The Partial Lambda-Calculus.* PhD thesis, University of Edinburgh, 1988. Available as CST-53-88.

[74] B. Möller. Algebraic specification with higher-order operations. In *Proceedings of IFIP TC 2 Working Conference on Program Specification and Transformation*, Amsterdam, 1987. North Holland.

[75] B. Möller, A. Tarlecki, M. Wirsing. Algebraic specification with built-in domain constructions. In M. Dauchet, M. Nivat, editors., *Proceedings of CAAP'88*, number 299 in Lecture Notes in Computer Science, 132–148, Berlin, 1988. Springer Verlag.

[76] T. Mossakowski. Parameterized recursion theory – a tool for the systematic classification of specification methods. In M. Nivat, C. Rattray, T. Rus, G. Scollo, editors., *Algebraic Methodology and Software Technology (AMAST'93)*, Workshops in Computing, 139–146. Springer Verlag, 1993. Also submitted to Theoretical Computer Science.

[77] T. Mossakowski. Equivalences among various logical frameworks of partial algebras. Bericht 4/95, Universität Bremen, 1995.

[78] T. Mossakowski. A hierarchy of institutions separated by properties of parameterized abstract data types. In E Astesiano, G. Reggio, A. Tarlecki, editors., *Recent Trends in Data Type Specification: 10th Workshop on Types – Selected Papers, Lecture Notes in Computer Science* **906**, 389–405. Springer Verlag, Berlin, 1995.

[79] T. Mossakowski, A. Tarlecki, W. Pawlowski. Combining and representing logical systems. Presented at Logic Colloquium 96, 1996.

[80] P. Mosses. Unified algebras and institutions. In *Proceedings of 4th Annual IEEE Symposium on Logic in Computer Science*, 304–312, 1989.

[81] A. Oberschelp. Untersuchungen zur mehrsortigen quantorenlogik. *Mathematische Annalen* **145**, 297–333, 1962.

[82] P. Padawitz. *Computing in Horn Clause Theories.* Springer Verlag, Berlin, 1988.

[83] 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*, number 332 in Lecture Notes in Computer Science, 208–234, Berlin, 1987. Springer Verlag.

[84] H. Reichel. *Initial Computability, Algebraic Specifications, and Partial Algebras.* Akademie Verlag, Berlin, 1986.

[85] J. Rosický. Semi-initial completions. *Journal of Pure and Applied Algebra* **40**, 177–183, 1986. correction ibid. 46:109, 1987.

[86] D. S. Scott. Identity and existence in intuitionistic logic. In M.P. Fourman, C.J. Mulvey, D.S. Scott, editors., *Application of Sheaves, Lecture Notes in Mathematics* **753**, 660–696. Springer Verlag, 1979.

[87] J.R. Shoenfield. *Mathematical Logic.* Addison-Wesley, Reading, Massachusetts, 1967.

[88] J. Sl ominski. *Peano-algebras and quasi-algebras*, *Dissertationes Mathematicae (Rozprawy Mat.)* **62**. 1968.

[89] A. Tarlecki. On the existence of free models in abstract algebraic institutions. *Theoretical Computer Science* **37**(3), 269–304, 1985.

[90] A. Tarlecki. Quasi-varieties in abstract algebraic institutions. *Journal of Computer and System Science* **33**, 333–360, 1986.

[91] J. Thatcher, E. G. Wagner, J. B. Wright. Specification of abstract data types using conditional axioms. Technical Report RC 6214, IBM Yorktown Heigths, 1981.

[92] M. Wirsing. Algebraic specification. In *Handbook of Theoretical Computer Science.* North Holland, 1990.

[93] Han Yan. *Theory and Implementation of Sort Constraints for Order Sorted Algebra.* PhD thesis, Oxford University, 1993.