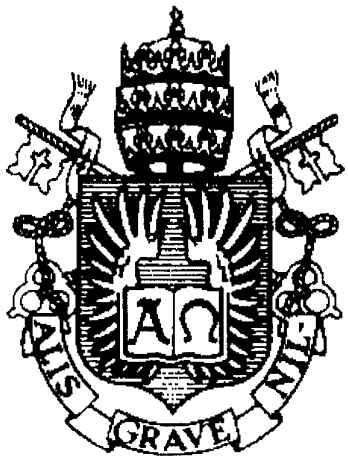
PUC



ISSN 0103-9741

Monografias em Ciência da Computação
nº 26/94

Logical Specifications: 2. Specifications as Presentations

Paulo A. S. Veloso
Thomas S. E. Maibaum

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900
RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA ISSN 0103-9741

Monografias em Ciência da Computação, N° 26/94

Editor: Carlos J. P. Lucena August, 1994

Logical Specifications:
2. Specifications as Presentations *

Paulo A. S. Veloso

Thomas S. E. Maibaum **

* Research partly sponsored by the Brazilian agencies CNPq, RHAE, FAPERJ, Ministério da Ciência e Tecnologia da Presidência da República Federativa do Brasil, British agency SERC and European Community agencies.

** Dept. of Computing, Imperial College of Science, Technology and Medicine, London, UK.

In charge of publications:
Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC Rio — Departamento de Informática
Rua Marquês de São Vicente, 225 — Gávea
22453-900 — Rio de Janeiro, RJ
Brasil
Tel. +55-21-529 9386 Telex +55-21-31048 Fax +55-21-511 5645
E-mail: rosane@inf.puc-rio.br

LOGICAL SPECIFICATIONS: 2. SPECIFICATIONS AS PRESENTATIONS

Paulo A. S. VELOSO and **Thomas S. E. MAIBAUM**
{e-mail: veloso@inf.puc-rio.br and tsem@doc.ic.ac.uk}

Abstract. The logical approach to formal specifications regards specifications as axiomatic presentations of theories in standard first-order logic. We present this view of specifications as theory presentations, exploring some of its methodological consequences. Particular attention is paid to underdetermined specification and the role of error or exceptions. We start by reviewing some syntactical and semantical concepts pertaining to languages and specifications. We then discuss how a specification generates its theory as well as the roles of equality and induction. Errors and underdetermined specifications are motivated and discussed in view of their role in developing programs and reasoning about them. Finally we introduce liberally constrained specifications as a more realistic concept of "good" specification, motivated by what is specified and available knowledge about it. A collection of example specifications is also provided. This report is a draft of the second section of a handbook chapter. Other reports cover the remaining sections.

Key words: Formal specifications, program development, logical approach, axiomatic presentations, equality, induction, errors, undetermined specifications,lLiberal specifications.

Resumo. O enfoque lógico para especificações formais trata especificações como apresentações axiomáticas de teorias na lógica usual de primeira ordem. Examinamos este enfoque de especificações como apresentações de teorias, explorando algumas de suas conseqüências metodológicas Dá-se particular atenção a especificações subdeterminadas bem como ao papel de erros ou exceções. Começamos revendo alguns conceitos sintáticos e semânticos relativos a linguagens e especificações. Passamos então a discutir como uma especificação gera sua teoria, bem como os papéis da igualdade e da indução. Questões de erros e especificações subdeterminadas são motivadas e discutidas tendo em vista seus papéis no desenvolvimento e certificação de programas. Finalmente introduzimos especificações com restrições liberais como um conceito mais realista de "boa" especificação, motivados pelo que é especificado e pelo conhecimento disponível sobre ele. Também se fornece uma coleção de especificações como exemplos. Este relatório é um esboço da segunda seção de um capítulo de um manual. Outros relatórios cobrem as demais seções.

Palavras chave: Especificações formais, desenvolvimento de programas, enfoque lógico, apresentações axiomáticas, igualdade, indução, erros, especificações subdeterminadas, especificações liberais.

NOTE

This report is a draft of the second section of a chapter in a forthcoming volume of the Handbook of Logic in Computer Science

Other reports, corresponding to the remaining sections, have been issued or are in preparation. The plan of the chapter - and series of reports Logical Specifications - is as follows.

1. Introduction and Overview MCC 13/94, June, 1994
2. Specifications as Presentations
3. Extensions of Specifications
4. Interpretation of Specifications
5. Implementation of Specifications
6. Parameterised Specifications
7. Conclusion: Retrospect and Prospects.

The chapter - and series of reports - is intended to provide an account of the logical approach to formal specification development.

Any comments or criticisms will be greatly appreciated.

The next report in this series is planned to be Logical Specifications: 3. Extensions of Specifications, with the following contents

3 EXTENSIONS OF SPECIFICATIONS

3.1 Introduction

3.2 Specification construction and hidden symbols

3.3 Extensions of languages and presentations

3.4 Conservative and eliminable extensions

3.5 Extensions by predicates

3.6 Extensions by operations

3.7 Extensions by constants

3.8 Extensions by sorts

3.9 Applications of extensions

3.10 Example specifications

ACKNOWLEDGEMENTS

Research reported herein is part of an on-going research project. Partial financial support from British, European Community and Brazilian agencies is gratefully acknowledged. The hospitality and support of the institutions involved have been very helpful. Collaboration with Martin R. Sadler, Sheila R. M. Veloso and José L. Fiadeiro was instrumental in sharpening many ideas. The authors would like to thank the following for many fruitful discussions on these and related topics: Carlos J. P. de Lucena, Samit Khosla, Atendolfo Pereda Bórquez, Douglas R. Smith, Haydée W. Poubel and M. Claudia Meré. Special thanks go to Tarcísio H. C. Pequeno and Roberto Lins de Carvalho

CONTENTS *

* See the preceding note for an explanation of the numbering system

List of Figures

List of Example Specifications

List of Results

2 Specifications as Presentations [1]

A specification 'describes' (properties of) some objects. We shall emphasise specifications formulated within the formalism of first-order logic. A specification will be an axiomatic presentation of a theory in many-sorted first-order logic. The properties described by such specification will be the consequences of the presentation.

We proceed in this section to present this view of specifications as theory presentations, exploring some of its methodological consequences. Particular attention is paid to underdetermined or (logically) incomplete specifications - in contrast to loose semantics - and the role of error or exceptions.

The semantics of a specification is generally given in terms of the models in which the axioms of the specification are satisfied. (Differences between different specification theories start arising when restrictions on the class of models allowed are imposed: initial models, finitely generated/reachable models, freely generated, etc.). In our setting, a specification is a theory presentation and the semantics of the specification (i. e. the meaning which we wish to assign to it) is the theory generated by that presentation. This distinction is methodologically important for the theory of specification. When constructing specifications of programs or systems, we construct presentations. It is in this construction and the validation which we apply to the result that the difference emerges.

We shall generally employ the usual terminology and notation for logical concepts, for which the reader is referred to standard textbooks, for instance (Enderton 1972; Ebbinghaus *et al.* 1984; Shoenfield 1967; van Dalen 1989). These will be briefly reviewed in the sequel.

The structure of this section is as follows. We start in 2.1 (Introduction) by presenting a simple example, a specification of Boolean with Neg(ation) and Less, indicating what a specification consists of. Then we briefly review in 2.2 some syntactical concepts related to First-order Languages, introducing some notation for terms, formulae, etc., and in 2.3 semantical concepts - like structure, model, value of term, theory, (logical) consequence, elementary equivalence. In 2.4 we examine Theories, Presentations and Specifications, discussing the theory generated by specification, equivalence of specifications and the role of equality. We introduce Inductive Sorts and Specifications by means of inductive schemas in 2.5, which are illustrated by some some simple examples: Naturals with zero and successor, Stacks of Elements. Our view of Errors and Underdetermined Operations is motivated in 2.6

---

[1] See the preceding note for an explanation of the terminology 'chapter', 'section', etc., as well as for the numbering system.

by discussing ideas such as computing values, overspecifying and abstraction, which is completed with our examination in 2.7 of Reasoning about Programs and the role of complete, sufficiently complete and liberal specifications, as well as the usefulness of undetermined operations for program refinement. Finally in 2.7 we introduce Liberally Constrained Specifications as a more realistic concept of "good" specification, motivated by what is specified and available knowledge about it. Our Example Specifications of this section are collected in 2.9.

2.1 Introduction

A specification consists of declarations and axioms. The declarations give syntactical information concerning its (extra-logical) symbols, akin to procedure headings which characterise syntactically correct procedure invocations. The axioms are intended to provide the available information concerning the behaviour of these symbols.

A simple example (see Spec. 2.1: BOOL_NEG_LESS in 2.9) will indicate what a specification consists of. The declaration part lists its sorts as well as its operations, constants and predicates, together with information concerning their arguments ans results, if any.

Most of the syntactical information given by the declarations of a specification can be graphically displayed in diagrams with appropriate conventions (Goguen *et al.* 1978). Figure 2.1 gives such a diagram for the specification of Boolean with Neg(ation) and Less given as Spec. 2.1: BOOL_NEG_LESS in 2.9.

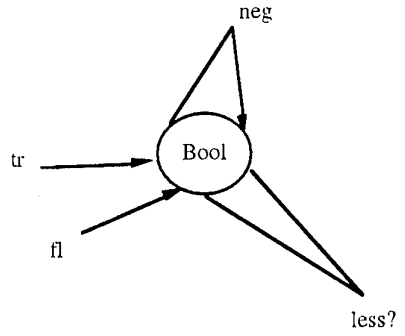


Figure 2.1: Syntactical diagram for Spec. 2.1: BOOL_NEG_LESS

2.2 Languages

We shall now indicate our basic notation and terminology concerning the first-order languages we shall use in specifications. These concern mainly sorts and alphabet, terms and formulae.

A first-order formalism deals with two kinds of symbols: the logical symbols and the extra-logical ones. The logical symbols have fixed interpretations, and are shared by all languages. The extra-logical symbols are open to interpretation, and are peculiar to a specific

language. Since the logical symbols are shared by all languages, with the same meaning, each language can be characterised by its alphabet of extra-logical symbols.

The *logical symbols* will be the usual *propositional connectives* $\neg, \wedge, \vee, \rightarrow$ and $\leftrightarrow$, as well as the *quantifiers* $\forall$ and $\exists$.

The *extra-logical symbols* can be partitioned into two sets: a set S of *sorts* and an alphabet A. The *alphabet* consists of two disjoint sets: a set R of *predicate symbols* and a set F of *operation symbols*. Each one of these sets, which may be empty, comes equipped with a declaration function, assigning, to each predicate symbol $r \in R$ its profile of sorts, and to each operation symbol $f \in F$ its profile of argument and result sorts. (Notice that constant symbols are treated as nullary operation symbols, even though we shall usually display them separately in presenting example specifications, due to their special importance.)

We also assume the presence of an *equality symbol* $\approx_s$, for each sort s. These special binary predicate symbols are usually not explicitly declared in presenting example specifications

In addition to the above symbols, one also needs a set of *variables*. This is a set V, disjoint from $S \cup A$, which comes partitioned into denumerably infinite sets V_s, called the set of *variables of sort* s, for each $s \in S$.

With a language one can build expressions. We are more interested in the well-formed expressions, of which we have two kinds: terms and formulae. Terms are intended to denote objects of domains and formulae serve to express properties of such objects.

A *term* is an expression built from variables with operation symbols so as to respect their declarations. The *sort* of a term is the sort of its result. A *name* is a ground term, i. e., one without variables.

The purpose of formulae is expressing properties, rather than denoting objects directly. They can be classified according to their construction. An *atomic formula* is an expression obtained by applying a predicate symbol to appropriate terms according to their declaration, which includes the equalities between two terms of the same sort. The *formulae* are constructed from the atomic ones by proper usage of connectives and quantifiers. A *sentence* is a formula without free variables.

We often display these objects as $L = \langle Srt(L), Alph(L) \rangle$ and $Alph(L) = \langle Prd(L), Opr(L) \rangle$ or $L = \langle Srt(L), Prd(L), Opr(L), Var(L) \rangle$. We shall use the following notations [1]

Var(L)[s] for the sets of variables of sort s;

---

[1] We often omit reference tothe language L when it is clear form the context.

$Prd(L)[s_1 \ldots s_m]$ and $Opr(L)[s_1 \ldots s_n \rightarrow s]$ for the set of predicate and of operation symbols with the given profile;

$Trm(L)[s]$, $Cnt(L)[s]$ and $Nm(L)[s]$ for the set of terms, constants and of names of sort s;

Atfm(L), Frml(L) and Sent(L) for the sets of atomic formulae, formulae and sentences.

When it is convenient and safe we shall use a simplified notation for free variables and *substitution*. We often say "given formula $\varphi(v)$" meaning that φ no variable, except possibly v, has free occurrence in φ. In this context $\varphi(t)$ denotes the result of replacing each, if any, free occurrence of v in φ by term t, by resorting to suitable alphabetic variant to avoid collision of variables. Similarly, "given formula $\varphi(v_1,\ldots,v_m)$" is intended to mean that all variables with free occurrences in φ are among $v_1,\ldots,v_m$. And $\varphi(t_1,\ldots,t_m)$ denotes the result of the simultaneous replacement of each, if any, free occurrence of v_k in φ by term t_k, with suitable precautions concerning collision of variables. A similar convention is used for terms: $t(v_1,\ldots,v_n)$ and $t(t_1,\ldots,t_n)$.

In order to improve legibility and simplify the notation, we shall freely omit parentheses or use brackets and the like, when convenient and safe.

2.3 Structures and models

Even though we favour a property-oriented approach, structures and models can be used as tools, both for conveying intuition and for establishing results. So, we now briefly review usual concepts related to structures and satisfaction, introducing appropriate notations.

As usual, a structure for a language provides realisations for its extra-logical symbols. A *structure* $\mathfrak{M}$ for language L amounts to an assignment of a realisation to each extra-logical symbol - so as to respect the syntactical declarations (Shoenfield 1967, p. 18; Enderton 1972, p. 79), in the sense that sorts are realised as non-empty sets and operations and predicate symbols as functions and relations, respectively, between the corresponding realisations of sorts. A *normal structure* is one where each equality symbol is realised by an identity relation. An *assignment* into a structure maps each variable over a sort to an element of the realisation of this sort.

As usual, an assignment extends inductively to a mapping of terms to values; and satisfaction is defined for atomic formulae, and then extended inductively to all formulae so as to capture the logical meaning of each connective and quantifier.

Given a structure $\mathfrak{M}$ for language L, we shall use the notations

$\mathfrak{M} \models \varphi\ [a_1,\ldots,a_m]$ to denote that $\mathfrak{M}$ satisfies formula φ under assignment $a(v_k)=a_k$, k=1,...,m;

$\mathfrak{M}[\varphi]$ for the *realisation* $\{<a_1,\ldots,a_m> / \mathfrak{B} \models \varphi\ [a_1,\ldots,a_m]\}$ of formula φ in $\mathfrak{M}$;

for a term t: $\mathfrak{M}[t](a_1,\ldots,a_n)$ for its *value* and $\mathfrak{M}[t]$ for its *realisation* $<a_1,\ldots,a_n> \rightarrow \mathfrak{M}[t](a_1,\ldots,a_n)$, in $\mathfrak{M}$;

for a sentence σ, $\mathfrak{M} \models \sigma$ or $\mathfrak{M} \in \mathrm{Mod}(\sigma)$ for $\mathfrak{M}$ is a *model* of σ;

Given a structure $\mathfrak{M}$ for language L, its *theory* is the set $\mathrm{Th}(\mathfrak{M}) := \{\sigma \in \mathrm{Sent}(L) / \mathfrak{M} \models \sigma\}$ and for a class **K** of structures for language L, its theory Th(**K**) consists of the sentences of L holding in each structure $\mathfrak{M} \in \mathbf{K}$. Structures $\mathfrak{A}$ and $\mathfrak{B}$ for language L are called *elementarily equivalent*, denoted by $\mathfrak{A} \equiv \mathfrak{B}$, iff they have the same theory: $\mathrm{Th}(\mathfrak{A}) = \mathrm{Th}(\mathfrak{B})$. A class **K** of structures for L is an *elementary class* (notation $\mathbf{K} \in \mathrm{EC}(L)$) iff it is closed under elementary equivalence: $\mathfrak{B} \in \mathbf{K}$ whenever $\mathfrak{A} \in \mathbf{K}$ and $\mathfrak{A} \equiv \mathfrak{B}$. Thus, $\mathbf{K} \in \mathrm{EC}(L)$ iff $\mathbf{K} = \mathrm{Mod}(\mathrm{Th}(\mathbf{K}))$.

A sentence σ of L is a (*logical*) *consequence* of a set Γ of sentences of L, denoted by $\Gamma \models \sigma$, iff $\mathrm{Mod}(\Gamma) \subseteq \mathrm{Mod}(\sigma)$ (every model of Γ is a model of σ). Thus, $\Gamma \models \sigma$ iff $\sigma \in \mathrm{Cn}(\Gamma)$, where $\mathrm{Cn}(\Gamma) := \mathrm{Th}(\mathrm{Mod}(\Gamma))$ is the *set of consequences* of Γ.

2.4 Theories, presentations and specifications

A specification generates a theory, consisting of the properties the specification guarantees. This is what we regard as the content of the specification. We now examine the basic concepts of our approach related to considering a specification as a theory presentation.

As usual a *theory* over language L is a set of T sentences of L closed under consequence ($\sigma \in T$ whenever $T \models \sigma$), and we call Γ a *set of axioms for theory* T iff $\mathrm{Cn}(\Gamma)=T$ (Enderton 1972; Ebbinghaus *et al.* 1984; Shoenfield 1967; van Dalen 1989).

By a *specification* we mean a *theory presentation*, i. e. a pair $P=<L,G>$, consisting of a first-order language L and a set G of sentences of L (its *axiomatisation*). Given a specification P, we use Lng(P) and Axm(P) to refer to its underlying language and set of axioms, respectively.

In order to describe the theory generated by a specification we need to back up a little. Recall that the equality symbols are assumed to be present, even though we may not bother to list them explicitly. Similarly, the corresponding equality properties may be assumed, even if we do not bother to include them in the set of axioms. Accordingly, the *theory* Cn[P] *generated* by a specification P consists of the consequences of its set of axioms augmented by the equality axioms. We shall say that specifications P' and P" are *equivalent*, denoted $P' \equiv P"$, iff

they have the same language (Lng(P')=Lng(P") and generate the same theory (Cn[P']=Cn[P"]).

The set Eq_axm(L) of *equality axioms of a language* L=<S,R,F> consists of the axioms stating that each equality has the properties of an equivalence preserved by all the operation and predicate symbols of L. We form the *equality completion* G∪Eq_axm(L) and say that a sentence σ of L is a *consequence* of P (notation P⊨σ) iff G∪Eq_axm(L)⊨σ, and accordingly the *theory of specification* P is Cn[P] := Cn(G∪Eq_axm(L)). Similarly, for a structure $\mathfrak{M}$ for L, $\mathfrak{M}$∈ Mod[P] is taken to mean $\mathfrak{M}$∈ Mod(G∪Eq_axm(L)).

2.5 Inductive sorts and specifications

Many objects involved in computing are inductive. Simple examples that come to mind are naturals, stacks, etc. So, let us indicate how ths idea appears in our property-oriented approach. The idea is that inductive sorts come with information for generating inductive axioms wihich are used in augmenting the axiomatisation, much in the spirit of the equality completion.

The idea of 'inductive' has various, somewhat related, senses. One sense is that of constructors: its elements are constructed from some basic ones by means of some constructor operations. This somewhat model-oriented view has the effect of providing an induction principle: a property that holds for the basic elements and holds for the results of constructor operations whenever it holds for the arguments will hold for all elements. This induction principle, sometimes called generator induction (Guttag 1977), is more appropriate for our property-oriented view.

By an *inductive sort* we mean one with an *inductive schema*. This inductive schema aims at capturing the inductive nature of the sort. Since such schemas have a simple general form we may resort to a device similar to the one employed for equality: we just indicate the information needed for generating the inductive principles. This information is given by formulae: a basis formula β for the basic elements and a set $\{\theta_1,\dots,\theta_k\}$ of formulae corresponding to the inductive step, so that we can form the *set* Ind(β;$\{\theta_1,\dots,\theta_k\}$) of *inductive axioms of sort* s *with respect to basis* β *and step* $\{\theta_1,\dots,\theta_k\}$ consisting of the *inductive axioms* Ind[φ](β;$\{\theta_1,\dots,\theta_k\}$) for each formula φ with a single free variable of sort s. The information of an inductive axiom, in terms of realisations, is "if φ includes β and is closed under $\theta_1,\dots,\theta_k$ then it exhausts the domain".

These inductive axioms are used, like the equality axioms, to augment the given axioms, for deriving consequences. So, for a

specification with inductive sorts, the theory of specification P consists of the consequences of its *inductive completion* $G \cup Eq\_axm(L) \cup Ind(\beta;\{\theta_1,\ldots,\theta_k\})$.

A simple example (see Spec 2.2: NAT_ZR_SUCC in 2.9) will indicate what a specification with inductive sorts may look like.

The item Nat: $Ind(x \approx_{Nat} zero;\{succ(x)x \approx_{Nat} y\})$ is intended to state that sort Nat is inductively constructed from zero by succ, in that it gives the information for generating the inductive axioms $\varphi(zero) \wedge (\forall x:Nat)[\varphi(x) \rightarrow \varphi(succ(x))] \rightarrow (\forall y:Nat)\varphi(y)$. We shall now describe how this is done. We have two formulae $x \approx_{Nat} zero$ and $succ(x)x \approx_{Nat} y$. From the former we form the basis sentence $(\forall x:Nat)[x \approx_{Nat} zero \rightarrow \varphi(x)]$, equivalent to $\varphi(zero)$, and from the latter we form the closure sentence $(\forall x,y:Nat)[\varphi(x) \wedge succ(x) \approx_{Nat} y \rightarrow \varphi(y)]$, equivalent to $(\forall x:Nat)[\varphi(x) \rightarrow \varphi(succ(x))]$. These basis and closure sentences contribute to forming the inductive axiom, as expected. By relying on such inductive axioms, one can derive consequences such as $(\forall y:Nat)[\neg y \approx_{Nat} zero \rightarrow (\exists x:Nat)y \approx_{Nat} succ(x)]$ or $(\forall y:Nat)[\neg y \approx_{Nat} succ(y)]$.

More generally, consider a formula φ with a single free variable of sort s, as well as formulae β, whose only free variable u is of sort s, and θ, with free variables u and v of sort s. We form the *basis sentence* $Bs(\beta,\varphi)$ as $(\forall u:s)[\beta(u) \rightarrow \varphi(u)]$, and the *closure sentence* $Cl(\varphi,\theta)$ as $(\forall u,v:s)[\varphi(u) \wedge \theta(u,v) \rightarrow \varphi(v)]$. Then, the *inductive axiom* $Ind[\varphi](\beta;\{\theta_1,\ldots,\theta_k\})$ of φ *with respect to basis* β *and step* $\{\theta_1,\ldots,\theta_k\}$ is the sentence $[Bs(\beta,\varphi) \wedge Cl(\varphi,\theta_1) \wedge \ldots \wedge Cl(\varphi,\theta_k) \rightarrow (\forall w:s)\varphi(w)]$. Finally, the *set* $Ind(\beta;\{\theta_1,\ldots,\theta_k\})$ of *inductive axioms of sort* s *with respect to basis* β *and step* $\{\theta_1,\ldots,\theta_k\}$ consists of the inductive axioms $Ind[\varphi](\beta;\{\theta_1,\ldots,\theta_k\})$ for each formula φ with a single free variable of sort s.

As another example, consider the case of stacks of elements with the usual operations push, pop and top (see Spec 2.3: STACK[ELEMENT] in 2.9). We shall comment on some aspects of this specification later on.

The inductive item for sort Stk gives as basis β the formula $u \approx_{Stk} crt$ and as step θ the formula $(\exists x:Elm)push(u,x) \approx_{Stk} v$. So, we have as basis sentence $Bs(\beta,\varphi)$, $(\forall u:Stk)[u \approx_{Stk} crt \rightarrow \varphi(u)]$, equivalent to $\varphi(crt)$, and as closure sentence $Cl(\varphi,\theta)$, $(\forall u,v:Stk)[\varphi(u) \wedge (\exists x:Elm)push(u,x) \approx_{Stk} v \rightarrow \varphi(v)]$, equivalent to $(\forall u:Stk)[\varphi(x) \rightarrow (\forall u:Stk)(\forall x:Elm)\varphi(push(u,x))]$. Thus, the inductive axiom $Ind[\varphi](\beta;\{\theta\})$ is equivalent to $\{\varphi(crt) \wedge (\forall u:Stk)[\varphi(u) \rightarrow (\forall x:Elm)\varphi(push(u,x))]\} \rightarrow (\forall w:Stk)\varphi(w)]$. A sentence in the inductive closure of this specification is $(\forall w:Stk)[w \approx_{Stk} crt \vee w \approx_{Stk} push(pop(w),top(w))]$.

Our specification NAT_ZR_SUCC for naturals with zero and successor in figure 2.2 involves two axioms and one inductive axiom shema. An alternative specification might consist of the two axioms together with the infinitely many axioms $(\forall x{:}Nat)(\neg x \approx_{Nat} succ^n(x))$, for each $n>0$ (here $succ^n(x)$ stands for the term $succ(\ldots succ(x)\ldots)$ with n occurrences of succ). They are equivalent specifications (Enderton 1972, p.178, 183), and this illustrates the fact that equivalent presentations for the same theory may be quite different, preference for one or another depending on several factors connected to the use at hand [1].

2.6 Errors and underdetermined operations

We shall now discuss some aspects of our approach concerning undetermined specifications. This is a central aspect of our liberal specifications.

The aim of most approaches to specification is to state only enough to describe the artefact being specified. This aim is sometimes discussed in the terminology of "what vs. how", "abstracting from implementation details", "representation independence" and so on (Guttag 1977, 1980; Goguen *et al.* 1978; Goguen 1977). It is the process of refinement or development which is to add the details of 'how', the details of a particular representation or implementation. The only meaning which this addition of detail could possibly have is the extension of the properties ascribed to the functions and relations used in constructing the specification. If there is no addition of new properties, but only the introduction of a new representation, it may be argued that no new information of any significance has been provided - just a change in naming conventions (Maibaum *et al.* 1991; Maibaum and Turski 1984).

Recall that an axiomatisation G is said to *decide* a sentence σ of its language iff $G \models \neg\sigma$ whenever $G \not\models \sigma$. Also, G is called *complete* iff it decides every sentence of its language. (An inconsistent axiomatisation is, of course, trivially complete; a *maximally consistent* axiomatisation is one that is both consistent and complete.)

A specification is called *complete* if for any sentence of its language, either it or its negation is derivable from the axioms. Complete specifications are generally very difficult to build. From the point of view of programming, they are also inappropriate, for they do not leave room for subsequent design decisions.

So, completeness is not a desirable property of a good abstract specification. Some approaches weaken this requirement to sufficient

---

[1] We examine some issues related to this point when considering liberally constrained specifications in 2.8.

completeness, which has to do with computing values in terms of constructors (Guttag 1977, 1980; Guttag and Horning 1978).

We shall now examine some aspects of the idea of computing values in terms of constructors and how it is captured by means of the inductive axioms. For this purpose, let us now examine more closely the specifications NAT_ZR_SUCC for naturals and STACK[ELEMENT] for stacks in 2.9 (see Spec. 2.2 and 2.3).

Part of the idea of constructor operations - at least from a model-oriented standpoint - is that every element, being constructed, can be denoted by a constructor term. In the case of naturals, the constructor terms are the numerals succ(... succ(zero)...). The analogue of such numerals for stacks would the terms $t(x_1,\ldots,x_{n-1},x_n)$ of the form $push(push(\ldots push(crt,x_1)\ldots,x_{n-1}),x_n)$. Now, Spec 2.3 has among its axioms, $(\forall u:Stk)(\forall x:Elm)[pop((push(u,x))\approx_{Stk}u$, called pP, and $(\forall u:Stk)(\forall x:Elm)[top((push(u,x))\approx_{Elm}x$, called tP. So, for any such term $t(x_1,\ldots,x_{n-1},x_n)$, with n>0, $\{pP\}\models pop[t(x_1,\ldots,x_{n-1},x_n)]\approx_{Stk}t(x_1,\ldots,x_{n-1})$ and $\{tP\}\models top[t(x_1,\ldots,x_{n-1},x_n)]\approx_{Elm}x_n$. Thus, we can say that axioms pP and tP enable one to compute the value of pop and top, respectively, on such constructor terms with n>0, by converting them into a 'normal form'. But, we still do not know how to compute them for the case n=0.

We contend that one should not worry about computing the values pop(crt) and top(crt) from the specification, because these are model-laden questions which are not relevant. Notice that it is quite natural to add axiom $pop(crt)\approx_{Stk}crt$, but we do not have a similar natural counterpart for top(crt). The usual approaches for the latter involve partial operations or error elements (Goguen *et al.* 1978; Guttag and Horning 1978).

In a nutshell, what is wrong with errors is their existence: life would be much easier without them. The aim of our approach is making life easier in so far as formal specifications are concerned. Our logical approach to formal specifications is directly motivated by program development and more adequate for this purpose. This approach is based on a property-oriented viewpoint, which permits, but does not demand, downplaying issues concerning errors.

The starting point is the following dichotomy:

(i) Programs and specifications are syntactical entities.

(ii) Errors/exceptions are semantical entities.

Let us elaborate these points. Point (i) does not appear to need much clarification. A program is a syntactical specification of a semantical object, the function it computes. The introduction gives further detail on this point.

Now let us turn to point (ii). Prototypical cases of errors in specifications arise in asking for the contents of an empty object, e.g., the top of the empty stack or the head of the empty list. We consider the latter: head(nil). Of course, there is no "natural" value to assign to it.

One way of dealing with head(nil) is saying that the operation realising head is partial, in particular it is not defined on the object denoted by nil. Along this line we find, e.g., the approach of (Broy and Wirsing 1983). A different alternative considers the operation realising head as total but the value of head(nil) is a special object denoted by error. The approaches of (Guttag 1977, 1980) and (Goguen *et al.* 1978; Goguen 1977) follow basically variations of this line. The main problem with the latter alternative is error propagation, which clutters up the specifications, whereas the former has to face the fact that not every term denotes an object, complicating some logical issues.

Notice that both alternatives arise from a semantical question: what is the value (denoted by) head(nil) (in a realisation). We offer a third alternative, mirroring the feeling that there is no "natural" value to assign to head(nil). We propose to leave the value of head(nil) open. Let us clarify this proposal. In a realisation, the value of head(nil) is not to be undefined (the operation realising head is to be total), nor is it an "abnormal" object denoted by error. Rather, our specification will not enable us to compute a unique primitive term to denote the value of head(nil). Thus, we do not require our specification to be sufficiently complete in the sense of Guttag and Horning (1978). Hence we will have several realisations of lists differing on the value they assign to head(nil).

Summing up, we call errors a semantical concern because they arise from a semantical question "What is the value of a term?", which only makes sense in the context of a given realisation. Now our proposal of "Life without errors" looks nice. But is it feasible and useful? Our proposal embodies a "non-uniqueness" viewpoint: a specification describes a class of (not necessarily isomorphic) realisations. But, again, this is our viewpoint couched in semantical terms. Syntactically, our proposal amounts to dealing with specifications that do not have to be complete or sufficiently complete.

Now, let us examine more closely our contention that we should not require that specifications enable prediction of values for operations (Veloso and Maibaum 1984). We claim that a specification should be required to predict only the relevant properties of its operations, and that the stronger requirement concerning all values may lead to overspecifying the operations, which is contrary to the idea of abstraction. For this claim we can offer support, related to mathematics and to programming.

The first line of support comes from an analogy with axiomatic theories in mathematics. When a mathematician studies structures, like groups or vector spaces, he is interested in the general properties of the operations, and not in computing the value of each closed term, which is not possible without some extra information.

From the viewpoint of programming we offer two considerations. First, error propagation does not adequately model computing practice: a program does not remain in an error state forever. When a program enters an error state, either its execution is aborted or an exception-handling routine is called. It will be seen that our approach does not force error propagation axioms into the specifications.

The second programming consideration might be deemed somewhat utopian. Good programs should not by themselves enter into error states. Indeed, a good programmer would not ask for the head of a list without first testing whether it is empty. Of course during the process of developing and tuning a program, this test may still not be present. Then what happens with a program that inadvertently tries to compute head (nil)? This depends on our specification. If we left head(nil) open then something will come out; only our specification does not predict what. On the other hand, we may include in our specification head(nil)=error (with or without error propagation) and then error will come out.

So, for programming, the contention is that the values of some terms may be irrelevant for the understanding of a program. We illustrate this point with an algorithm for determining the union of sets. (This example will examined with more detail in 2.7.)

We are given two sets S and T and are required to determine their union. A quite natural approach to this problem is based on the idea of transferring elements from one of the sets to the other while the former is not empty. In more detail we iterate the transfer, consisting of selecting an element of S, removing it from S, and inserting it into T, while S is not empty.

It is apparent that we have a correct algorithm for union. If we begin with S=M and T=N, then at the end we have T=M$\cup$N (and the algorithm will terminate if M is finite). What do we need to know about for arriving at this conclusion? Apparently we need to know that removal and insertion behave as expected. And what about selecting, does it matter how the element is selected? This question appears to be totally irrelevant for the understanding of this algorithm: all that matters is that an element of S is selected provided that S is not empty. The question concerning the value of the selection operation on an empty set is besides the point, because the algorithm never tries to do so. We may say it is an error, that selection is not defined for the empty set, or just

be silent about this case and assert only what is required, which is the alternative we prefer.

"All that we need to know about selection" would be asserted in an axiom stating that, provided a set is not empty, the element selected belongs to it. Thus, even if we know that S={a,b}, we are still unable to determine the exact element selected, whether a or b. All that we can assert is that it is one of a or b. This is an example of an undetermined, or underspecified, operation

Notice that an undetermined operation should not be confused with a nondeterministic one. For, sel(ection) is an operation symbol and hence must in any model be realised by a total function from sets to elements. Thus, given a specific set in such a model two different applications of (the realisation of) sel to this set will yield exactly the same result. Proof theoretically, we can derive from the specification only a disjunction like

$$sel[ins(ins(void,a),b))]\approx_{Elm}a \vee sel[ins(ins(void,a),b))]\approx_{Elm}b;$$

without being able to derive either disjunct. But, this is not an indication of nondeterminism. It is an indication of underdetermined specification. For whatever reason, we are unable, or unwilling, to pin down any more precisely what the value of sel applied to a set should be. The decision is being left to be made at a later time (when we have more information about the application or at some point in refinement when we wish to make a design decision which involves knowing more about sel). We may in some cases never need to decide completely about some detail if this decision is not required to correctly implement our specification.

Undetermined operations can be very useful for specifying, and reasoning about, programs and specifications at the appropriate level of abstraction. We provide the information that is required at this level and postpone giving more information to some later level of refinement. For instance, when we have decided to represent sets by sequences, we may decide to refine choose to select the first element.

Another example of the usefulness of undetermined operations is provided by sorting algorithms. The idea behind quicksort amounts to repeatedly splitting a sequence by comparison of its elements with the pivot. Some details about how the pivot is to be selected may postponed to a later stage, when their impact on efficiency can be more clearly assessed.

Summing up, we do not require our specifications to be complete, in the logical sense, or even sufficiently complete, in the sense of Guttag and Horning (1978). A specification is required to provide properties of its symbols; and the information provided may or may not be adequate for answering a specific question, depending on its level of abstraction.

2.7 Reasoning about programs

We shall now examine more closely the adequacy of liberal specifications for developing, and reasoning about, programs. We shall give special attention to the issue of underdetermined operations in not sufficiently complete specifications.

We have claimed that knowledge about the values of some terms may be irrelevant for the understanding of a program. This is one of the basic ideas underlying the usefulness of liberal specifications in that they provide the required information without being forced into considering irrelevant issues. But, are they really adequate for reasoning about programs?

In order to give a better indication of the adequacy of liberal specifications, we examine more closely the our example of union of sets.

Consider the formal specification SET[ELEMENT] of sets of elements given in 2.9 as Spec 2.4. Some properties (with implicit universal quantifications) derivable from this specification are

1.[¬empty?(ins(s,x))];

2. [blng(x,s)→(∀y:Elm)[(blng(y,s)∨blng(y,t))↔
↔(blng(y,rem(s,x))∨blng(y,ns(t,x)))]

The specification for our desired program is the following formula is_union(s,t,r) (expressing union in terms of belonging):

(∀y:Elm)[blng(y,r)↔(blng(y,s)∨blng(y,t))] (is_union)

The intuitive idea of transferring elements from one set to the other is expressed in the following invariant is_trnsf(s,t,m,n) (stating that s∪t=m∪n):

(∀y:Elm)[(blng(y,s)∨blng(y,t))↔(blng(y,m)∨blng(y,n))] (is_trnsf)

Now, consider the annotated program segment in figure 2.2.

```
{program segment for union of sets}
var S,T: Set {variables S and T over sort Set};
var X:Elm {variable X over sort Set};
while ¬empty?(S) do
   begin {here ¬empty?(S) holds}
     X:=sel(S) {X is some element from S}
     S:=rem(S,X) {element X is removed from S}
     T:=ins(T,X) {element X is inserted into T}
   end_while
{at completion of the loop empty?(S) holds}.
```

Fig. 2.2: Program segment for union of sets

We shall now see how we can reason about this program segment by relying on the information provided by the specification. Let us assume that execution begins with S=M and T=N. We wish to conclude that at the end we have T=M∪N, i. e. is_union(M,N,T).

Let us first consider partial correctness. For this purpose we first check the invariance of is_trnsf(S,T,M,N) and from it derive the desired conclusion. Clearly, at the starting point, since we assume S=m and T=n, we have is_trnsf(S,T,M,N). Now, to check that is_trnsf(S,T,m,n) is preserved by the iteration we establish the sentence

$$\neg\text{empty?}(S)\wedge X\approx_{Set}\text{sel}(S)\rightarrow(\text{is\_trnsf}(S,T,M,N)\rightarrow \rightarrow\text{is\_trnsf}(\text{ins}(S,X),\text{rem}(T,X),M,N)$$

(with implicit universal quantification), which follows form our axioms in view of property 2 above. We can then conclude that if and when the execution of the loop is completed we will have both is_trnsf(S,T,M,N) and empty?(S), whence is_union(M,N,T).

Termination is guaranteed only for a finite set M. Since finiteness cannot be expressed within first-order logic, we have basically two approaches for termination. They are both based on the idea that removal of an element belonging to a set produces a subset. The first one establishes that it produces a proper subset, and the second one that the resulting set can be denoted by a proper subterm. In either case we have a binary relation which is known to be well founded on finite sets.

In the first, subset, approach we consider the following formulae sbst(s,t) and prsb(s,t) (expressing respectively $s\subseteq t$ and $s\subset t$)

$$(\forall x{:}\text{Elm})[\text{blng}(x,s)\rightarrow\text{blng}(x,t)] \qquad \text{(sbst)}$$

$$\text{sbst}(s,t)\wedge\neg\text{sbst}(t,s) \qquad \text{(prsb)}$$

Then, we derive from our specification (without inductive axioms)

$$(\forall s{:}\text{Set})(\forall x{:}\text{Elm})[\text{blng}(x,s)\rightarrow\text{prsb}(\text{rem}(s,x),x)].$$

In the second, subterm, approach we derive from our specification (with inductive axioms)

$$(\forall s{:}\text{Set})(\forall x{:}\text{Elm})[\text{blng}(x,s)\rightarrow s\approx\text{ins}(\text{rem}(s,x),x)].$$

It is important to notice that in either approach we do not need information about sel beyond that provided by its axiom [1].

2.8 Liberally constrained specifications

Let us know examine more closely the question of what one wishes to describe when one writes a specification. This has to do with correctness and adequacy of specifications.

---

[1] More details on erors and related issues are planned to appear in the forthcoming report Logical Specifications: 3. Extensions of Specifications.

Some approaches take the view that a specification specifies a class of structures (Goguen *et al.* 1978; Guttag and Horning 1978). We might adapt this model-oriented viewpoint to our property-oriented approach simply by saying that a specification describes the properties of a class of structures. In any case the problem is one of axiomatising, by a convenient set of axioms, either a set of sentences or class of structures described, say, by set-theoretical means.

The crucial question is how one gives the object(s) to be specified. We contend that it is unrealistic to expect that one has, before specifying, reliable knowledge about the entire class of structures or the set of properties that will be specified. We therefore suggest a more realistic approach: 'specifications with liberal constraints'.

The idea is that at the start of the specification, we do not have perfect knowledge either of the models or of the properties of the final product. But, we do know, and insist upon

a certain class **Req_Mod** of required models (for instance, 'standard' models),

a set Req_Prop of required properties (for instance, associativity of an operation);

and we are willing to accept any specification that satisfies these liberal constraints, and only those.

As an example of liberally constrained specifications, consider specifying the ordering of the natural numbers in a language with a single binary predicate symbol lt. Let us assume that we have very little a-priori knowledge, so that we take the class **Req_Mod** of required models to consist only of the standard model $\mathfrak{N} = \langle \mathbf{N}, <\rangle$ and as required properties only that we have a linear ordering, i. e. Req_Prop=$\{\iota,\tau,\lambda\}$ where ι,τ, and λ are the following sentences expressing irreflexivity, transivity and linearity: $(\forall x:Nat)\neg lt(x,x)$, $(\forall x,y,z:Nat)[lt(x,y)\wedge lt(y,z)\rightarrow lt(x,z)]$ and $(\forall x,y:Nat)[lt(x,y)\vee x\approx_{Nat} y\vee lt(y,x)]$. Notice that $\mathfrak{N}\models \iota\wedge\tau\wedge\lambda$, so $\mathfrak{N}\in$ Mod(Req_Prop).

Some specifications that meet these constraints are given by the following axiomatisations:

LOF = $\{\iota,\tau,\lambda\}\cup\{\phi\}$, where ϕ is $(\exists x:Nat)(\forall y:Nat)\neg lt(y,x)$;

LO+∞ = $\{\iota,\tau,\lambda\}\cup\{+\infty\}$, where $+\infty$ is $(\forall x:Nat)(\exists y:Nat)lt(x,y)$;

LOD = $\{\iota,\tau,\lambda\}\cup\{\delta\}$, where δ is $\neg(\forall x,y:Nat)[lt(x,y)\rightarrow(\exists z:Nat)(lt(x,z)\wedge lt(z,y)]$;

LOF+∞ = LOF$\cup$LO+∞ = Req_Prop$\cup\{\phi,+\infty\}$;

LOFD = LOF$\cup$LOD = Req_Prop$\cup\{\phi,\delta\}$;

LO+∞D = LO+$\infty\cup$LOD = Req_Prop$\cup\{+\infty,\delta\}$;

LOF+∞D = LOF$\cup$LO+$\infty\cup$LOD = Req_Prop$\cup\{\phi,+\infty,\delta\}$;

We thus have several specifications meeting these constraints, as displayed in figure 2.3, which indicates the leeway one has.

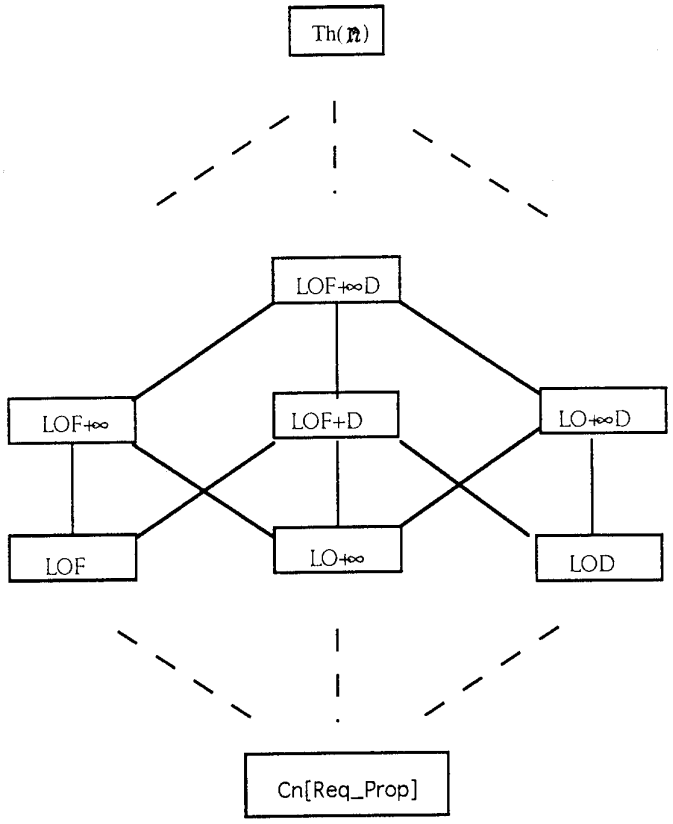


Fig. 2.3: Liberally constrained specifications

Consider a language L, as well as a class **K** of structures of L and a set Σ of sentences of L. By a *specification liberally constrained* to *required models* **K** and *required properties* Σ we mean any specification P=<L,G> such that **K**⊆Mod[P] and Σ⊆Cn[P], and we shall use the notation Spc(Σ,**K**] for the set {P=<L,G>/**K**⊆Mod[P] & Σ⊆Cn[P]} of specifications liberally constrained to required models **K** and required properties Σ.

Notice that axiomatisations of a class **K** of structures or of a set Σ of sentences are special cases of this liberal notion of specification. Indeed.

Spc(T,T] = {P=<L,G>/T=Cn[P]} is the set of axiomatisations for class for the theory T=Cn[T], and

Spc(Th(**K**),**K**] = {P=<L,G>/**K**=Mod[P]} is the set of axiomatisations for the elementary class **K**=Mod(Th(**K**)) of structures.

As might be expected, the required models and required properties must have some compatibility for the existence of liberally constrained specifications.

Proposition Existence of liberally constrained specifications

Given a language L, consider a class **K** of structures of L and a set Σ of sentences of L.

a) $Spc(\Sigma,\mathbf{K})=Spc(\Sigma,\varnothing]\cap Spc(\varnothing,\mathbf{K}]$,

b) $\Sigma\subseteq Th(\mathbf{K})$ iff $Spc(\Sigma,\mathbf{K}]\neq\varnothing$ iff $\mathbf{K}\subseteq Mod(\Sigma)$.

Proof.

a) Clear.

b) If $\Sigma\subseteq Th(\mathbf{K})$ then $P=<L,Th(\mathbf{K})>\in Spc(\Sigma,\mathbf{K}]$.

If $P=<L,G>\in Spc(\Sigma,\mathbf{K}]$ then $\mathbf{K}\subseteq Mod[P]=Mod(Cn[P])\subseteq Mod(\Sigma)$.

If $\mathbf{K}\subseteq Mod(\Sigma)$ then $\Sigma\subseteq Cn[\Sigma]=Th(Mod(\Sigma))\subseteq Th(\mathbf{K})$.

QED

One advantage of this liberal notion of specification, over the usual ones, is the amount of freedom one has. This can be seen by examining their theories $Cn(\Sigma,\mathbf{K}] := \{T/\mathbf{K}\subseteq Mod(T)\ \&\ \Sigma\subseteq Cn[T]=T\}$.

Theorem Lattice of liberally constrained specifications

Consider a class **K** of structures of language L and a set Σ of sentences of L. If $Spc(\Sigma,\mathbf{K}]\neq\varnothing$ then the set $Cn(\Sigma,\mathbf{K}]$ of theories liberally constrained to required models **K** and required properties Σ forms a complete lattice under inclusion, with least element $Cn[\Sigma]$ and top element $Th(\mathbf{K})$.

Proof.

Clearly, for any $T\in Cn(\Sigma,\mathbf{K}]$, $Cn[\Sigma]\subseteq T$, for $\Sigma\subseteq T$, and $T\subseteq Th(\mathbf{K})$, for $\mathbf{K}\subseteq Mod(T)$.

The empty family $\varnothing$ has $\inf(\varnothing)=Th(\mathbf{K})$ and $\sup(\varnothing)=Cn[\Sigma]$, both in $Cn(\Sigma,\mathbf{K}]$.

Now consider a nonempty family $\mathcal{F}\subseteq Cn(\Sigma,\mathbf{K}]$. We claim that $\mathcal{F}$ has infimum $\cap\mathcal{F}$ and supremum $Cn[\cup\mathcal{F}]$, both in $Cn(\Sigma,\mathbf{K}]$.

Clearly, $\cap\mathcal{F}$ is the smallest theory containing every theory in $\mathcal{F}$; so $\Sigma\subseteq\mathcal{F}$.
Since $\mathcal{F}\neq\varnothing$, we have some $T\in Cn(\Sigma,\mathbf{K}]$ with $\cap\mathcal{F}\subseteq T$, so $\mathbf{K}\subseteq Mod(T)\subseteq Mod(\cap\mathcal{F})$.
Thus, $\cap\mathcal{F}\in Cn(\Sigma,\mathbf{K}]$, as claimed.

Also, $Cn[\cup\mathcal{F}]$ is the largest theory contained in every theory in $\mathcal{F}$.
Since $\mathcal{F}\neq\varnothing$, we have some $T\in Cn(\Sigma,\mathbf{K}]$ with $T\subseteq Cn[\cup\mathcal{F}]$, so $\Sigma\subseteq T\subseteq Cn[\cup\mathcal{F}]$.
Now, for every $T\in\mathcal{F}\subseteq Cn(\Sigma,\mathbf{K}]$, $\mathbf{K}\subseteq Mod(T)$ and $T\subseteq\cup\mathcal{F}$. Thus,
$\mathbf{K}\subseteq Mod(\cup\mathcal{F})=Mod(Cn[\cup\mathcal{F}])$. Hence, $Cn[\cup\mathcal{F}]\in Cn(\Sigma,\mathbf{K}]$, as claimed.

QED

2.9 Example Specifications

We now present some simple examples of specifications. These examples will show some desirable features of a formalism for presenting specifications.

First the specifications already mentioned. These are Boolean with Neg(ation) and Less (BOOL_NEG_LESS), Naturals with zero and successor (NAT_ZR_SUCC), Stacks of Elements (STACK[ELEMENT]) and Sets of Elements (SET[ELEMENT]).

Spec. 2.1. BOOL_NEG_LESS: Boolean with Neg(ation) and Less

SPEC BOOL_NEG_LESS {Specification of Boolean with Neg(ation) and Less}
  DECLARATIONS {Description of symbols}
    Sorts {List of sorts}
      Bool {The (only) sort is Bool}
    Operations {List of (non-nullary) operations}
      neg (Bool)→Bool {neg is from Bool into Bool}
    Constants {List of constants}
      tr,fl: Bool {tr and fl are constants of Bool}
    Predicates {List of predicates besides $\approx_{Bool}$}
      less? (Bool,Bool) {less? over Bool and Bool}
  AXIOMS {List of axioms}
    $(\forall x{:}Bool)(x\approx_{Bool}tr \vee x\approx_{Bool}fl)$ {tr and fl exhaust Bool},
    $\neg tr\approx_{Bool}fl$ {tr and fl distinct},
    $(\forall x{:}Bool)(\neg neg(x)\approx_{Bool}x)$ {neg changes values},
    $neg(tr)\approx_{Bool}fl$ {negaton of tr is fl},
    $(\forall x,y{:}Bool)[less?(x,y)\leftrightarrow(x\approx_{Bool}fl \wedge x\approx_{Bool}tr)]$ {definition of less?}.
END_SPEC BOOL_NEG_LESS

Spec. 2.2. NAT_ZR_SUCC: Naturals with zero and successor

SPEC NAT_ZR_SUCC {Specification of Naturals with zero and successor}
  DECLARATIONS
    Sorts
      Nat {The (only) sort is Nat}
    Operations
      succ (Nat)→Nat {succ transforms Nat to Nat}
    Constants
      zero: Nat {zero is a constant of Nat}
    Predicates {No predicates, other than $\approx_{Nat}$}
  AXIOMS
    $(\forall x{:}Nat)\neg zero\approx_{Nat}succ(x)$ {zero not in the range of succ},
    $(\forall x,y{:}Nat)[succ(x)\approx_{Nat}succ(y)\rightarrow x\approx_{Nat}y]$ {succ injective},
    $Nat{:}Ind(x\approx_{Nat}zero;\{succ(x)x\approx_{Nat}y\})$ {Nat inductive on zero and succ}.

THEOREMS {Sample consequences}

$(\forall x{:}Nat)[\varphi(x)\rightarrow\varphi(succ(x))]$ {inductive schema},

$(\forall y{:}Nat)[\neg y\approx_{Nat}zero\rightarrow(\exists x{:}Nat)y\approx_{Nat}succ(x)]$ {inductive reachability},

$(\forall y{:}Nat)[\neg y\approx_{Nat}succ(y)]$ {succ has no fixpoint}.

END_SPEC NAT_ZR_SUCC

Spec. 2.3. STACK[ELEMENT]: Stacks of Elements

SPEC STACK[ELEMENT] {Specification of Stacks of Elements}

DECLARATIONS

Sorts

Stk, Elm {The sorts are Stk and Elm}

Operations

push (Stk,Elm)→Stk {push gives Stk from Stk and Elm}

pop (Stk)→Stk {pop transforms Stk to Stk}

top (Stk)→Elm {top gives Elm from Stk}

Constants

crt: Stk {crt is a constant of Stk}

Predicates

is_null? (Stk) {is_null? is over Stk}

AXIOMS

$(\forall u{:}Stk)(\forall x{:}Elm)(\neg crt\approx_{Stk}push(u,x))$ {crt not in the range of push},

$(\forall u{:}Stk)(\forall x{:}Elm)[pop((push(u,x))\approx_{Stk}u$ {axm pP},

$(\forall u{:}Stk)(\forall x{:}Elm)[top((push(u,x))\approx_{Elm}x$ {axm tP},

$(\forall u{:}Stk)(is\_null?(u)\leftrightarrow u\approx_{Stk}crt)$,

$Stk{:}Ind(u\approx_{Stk}crt;\{(\exists x{:}Elm)push(u,x)\approx_{Stk}v\})$ {Stk inductive on crt and push}

THEOREMS {Sample consequences}

$\{\varphi(crt)\wedge(\forall u{:}Stk)[\varphi(u)\rightarrow(\forall x{:}Elm)\varphi(push(u,x))]\}\rightarrow(\forall w{:}Stk)\varphi(w)]$,

Let $t(x_1,\ldots,x_{n-1},x_n):=push(push(\ldots push(crt,x_1)\ldots,x_{n-1}),x_n)$ {Let block}

with $x_1,\ldots,x_{n-1},x_n{:}Elm$

$top[t(x_1,\ldots,x_{n-1},x_n)]\approx_{Elm}x_n$ {value of top},

$pop[t(x_1,\ldots,x_{n-1},x_n)]\approx_{Stk}t(x_1,\ldots,x_{n-1})$ {value of top},

End_Let $t(x_1,\ldots,x_{n-1},x_n)$ {End let block};

$(\forall w{:}Stk)[w\approx_{Stk}crt\vee w\approx_{Stk}push(pop(w),top(w))]$.

COMMENTS {General remarks}

Specification of pop and top are underdetermined: the values of pop(crt) and top(crt) are left open.

END_SPEC STACK[ELEMENT]

Spec. 2.4. SET[ELEMENT]: Sets of Elements

SPEC SET[ELEMENT] {Specification of Sets of Elements}

DECLARATIONS

Sorts

Set, Elm {The sorts are Set and Elm}

Operations

ins, rem (Set,Elm)→Set {ins and rem give Set from Elm & Stk}

sel (Set)→Elm {sel gives Elm to Set}

Constants

void: Set {void is a constant of Set}

Predicates

empty? (Set) {empty? is over Set}

blng (Elm,Set) {blng is between Elm & Set}

AXIOMS

$(\forall s,t:Set)(\forall x:Elm)$ {Global quantification (for all axioms)}

$[empty?(s)\leftrightarrow s\approx_{Set}void]$ {void vs. empty?},

$[empty?(s)\leftrightarrow(\forall y:Elm)\neg blng(y,s)]$ {empty? vs. blng},

$(\forall y:Elm)[blng(y,ins(s,x))\leftrightarrow(y\approx_{Elm}x\vee blng(y,s))]$ {behaviour of ins},

$(\forall y:Elm)[blng(y,rem(s,x))\leftrightarrow(\neg y\approx_{Elm}x\wedge blng(y,s))]$ {behaviour of rem},

$[\neg empty?(s)\rightarrow blng(sel(s),s)]$ {sel on nonempty set},

$[s\approx_{Set}t\leftrightarrow(\forall y:Elm)(blng(y,s)\leftrightarrow blng(y,t))]$ {$\approx_{Set}$ vs. blng}

$Set:Ind(empty?(s);\{(\exists x:Elm)s\approx_{Set}ins(s,x)\})$ {Stk inductive on crt and push}

THEOREMS {Sample consequences}

$\{\varphi(void)\wedge\wedge(\forall s:Set)[\varphi(s)\rightarrow(\forall x:Elm)\varphi(ins(s,x))]\}\rightarrow(\forall t:Set)\varphi(t)$ {schema},

Quant $(\forall s:Set)(\forall x:Elm)$ {Quantification block}

$empty?(s)\vee(\exists t:Set)(\exists x:Elm)s\approx_{Set}(ins(t,x))$ {inductive reachability},

$ins(ins(s,x),x)\approx_{Set}ins(s,x)$ {idempotent ins},

$ins(ins(s,x),y)\approx_{Set}ins(ins(s,y),x)$ {ins commutes},

$[\neg empty?(ins(s,x))]$

$blng(x,ins(s,x))$

$\neg blng(x,rem(s,x))$

End_Quant $(\forall s:Set)(\forall x:Elm)$ {End quantification block};

Let t:=ins(ins(void,x),y)) with x,y:Elm {Let block}

$sel(t)\approx_{Elm}x\vee sel(t)\approx_{Elm}y$ {undetermined sel},

$pop[t(x_1,\ldots,x_{n-1},x_n)]\approx_{Stk}t(x_1,\ldots,x_{n-1})$ {value of top},

End_Let $t(x_1,\ldots,x_{n-1},x_n)$ {End let block};

$(\forall s:Set)(\forall x:Elm)rem(ins(s,x),x)\approx_{Set}rem(s,x),$

$(\forall s{:}Set)(\forall x{:}Elm)rem(ins(s,x),x) \approx_{Set} rem(s,x)$.

COMMENTS {General remarks}

Specification of sel is underdetermined:
can say sel[ins(ins(void,a),b))] is aor b, but not which one!
$sel[ins(ins(void,a),b))] \approx_{Elm} b$.

END_SPEC SET[ELEMENT]

We now present a specification for a simple version of the integers: with zero, successor and predecessor ordered by <.

Spec. 2.5. INT: Integers with zero, successor and predecessor ordered by <

SPEC INT {Integers with zero, successor and predecessor ordered by <}

DECLARATIONS

Sorts
Int

Operations
sc, prd (Int)→Int

Constants
zr: Int

Predicates
Int<Int {< infix between Int & Int}

AXIOMS

$(\forall x,y{:}Int)[x<y \rightarrow \neg y<x]$ {< antisymmetric},
$(\forall x,y,z{:}Int)[(x<y \wedge y<z) \rightarrow x<z]$ {< transitive},
$(\forall x,y{:}Int)[x<y \vee x \approx_{Int} y \vee y<x]$ {< linear},
$(\forall x,y{:}Int)[x<sc(y) \leftrightarrow x<y \vee x \approx_{Int} y]$ {< vs. sc},
$(\forall x,y{:}Int)[prd(x)<y \leftrightarrow x<y \vee x \approx_{Int} y]$ { prd vs.< },
$(\forall x{:}Int)[prd(sc(x)) \approx_{Int} x \wedge sc(prd(x)) \approx_{Int} x]$ { sc & prd inverse},

THEOREMS {Sample consequences}

$(\forall x{:}Int)\neg x<x$ {< ireflexive},
$(\forall x,y{:}Int)[\neg x<y \rightarrow (x \approx_{Int} y \vee y<x)]$,
$(\forall x{:}Int)[x<sc(x) \wedge prd(x)<x]$,
$(\forall x,y{:}Int)[x<y \rightarrow (sc(x)<sc(y) \wedge prd(x)<prd(y))]$ {< transitive},

For n>0 and x:Elm let {For block}
$sc^n(x):=sc(\ldots(sc(x))\ldots)$ & $prd^n(x):=prd(\ldots(prd(x))\ldots)$ [both n times]
$x<sc^n(x) \wedge prd^n(x)<x$ {no sc or prd loops},
End_For $sc^n(x)$ & prd^n {End for block};

END_SPEC INT

A data type provided by several programming languages is integers with the usual arithmetic operations and predicates. It can be obtained by extending INT. (Extensions will be examined more closely in the next section.)

Spec. 2.6. INT_ARITHM: Integers with arithmetic operations and predicates

SPEC INT_ARITHM := EXT of INT by {Extension of INT}
  DECLARATIONS {Description of new symbols}
    Sorts {No new sort}
    Operations {List of new (non-nullary) operations}
      $|Int| \rightarrow Int$ {outfix |n| = absolute value}
      $Int\ +,*,-\ Int \rightarrow Int$ {binary infix +,*,-}
    Constants {No new constant}
    Predicates {List of new predicates}
      $Int \leq Int$ {< infix between Int & Int}
      int_div (Int,Int,Int,Int) {int_div (m,n,q,r): quotient q, remainder r}
  AXIOMS {List of new axioms}
    Quant $(\forall x,y:Int)$ {Quantification block}
      $[x \leq y \leftrightarrow (x < y \vee x \approx_{Int} y)]$ {definition of ≤},
      $\{x+zr \approx_{Int} x \wedge [x+sc(y) \approx_{Int} sc(x+y) \wedge x+prd(y) \approx_{Int} prd(x+y)\}$ {+},
      $\{x-zr \approx_{Int} x \wedge [x-sc(y) \approx_{Int} prd(x+y) \wedge x-prd(y) \approx_{Int} sc(x-y)\}$ {-},
      $\{x*zr \approx_{Int} zr \wedge [x*sc(y) \approx_{Int} x*y+x \wedge x*prd(y) \approx_{Int} x*y-x\}$ {*},
      $\{|x| \approx_{Int} y \leftrightarrow [(zr \leq x \wedge y \approx_{Int} x) \vee (x < zr \wedge y \approx_{Int} -x)]\}$ {definition of |•|}
    End_Quant $(\forall x,y:Int)$ {End quantification block};
    $(\forall x,y,u,w:Int)[int\_div(x,y,u,w) \leftrightarrow (zr \leq u \wedge u < |y| \wedge x \approx_{Int} y*u+w)]$ {int-div}
END_SPEC INT_ARITHM

2.10 References

Arbib, M. and Mannes, E. (1975). *Arrows, Structures and Functors : the Categorical Imperative*. Academic Press, New York.

Barwise, J. ed. (1977). *Handbook of Mathematical Logic*. North-Holland, Amsterdam.

Bauer, F., L. and Wössner, H. (1982). *Algorithmic Language and Program Development*. Springer Verlag, Berlin.

Darlington, J. (1978). A synthesis of several sorting algorithms. *Acta Informatica*, **11** (1), 1-30.

Ebbinghaus, H. D., Flum, J. and Thomas, W. (1984). *Mathematical Logic*. Springer-Verlag, Berlin.

Enderton, H. B. (1972). *A Mathematical Introduction to Logic.* Academic Press; New York.

Ehrich, H.-D. (1982). On the theory of specification, implementation and parameterization of abstract data types. *J. ACM*, **29** (1), 206-227.

Ehrig, H. and Mahr, B. (1985) *Fundamentals of Algebraic Specifications, 1: Equations and Initial Semantics.* Springer-Verlag, Berlin.

Ledgard, H. and Taylor, R. W. (1977) Two views on data abstraction. *Comm. Assoc. Comput. Mach.*, **20** (6), 382-384.

Guttag, J. V (1977). Abstract data types and the development of data structures. *Comm. Assoc. Comput. Mach.*, **20** (6), 396-404.

Maibaum, T. S. E. (1986). The role of abstraction in program development. In Kugler, H.-J. ed. *Information Processing '86.* North-Holland, Amsterdam, 135-142.

Maibaum, T. S. E., Sadler, M. R. and Veloso, P. A. S. (1984). Logical specification and implementation. In Joseph, M. and Shyamasundar R. eds. *Foundations of Software Technology and Theoretical Computer Science.* Springer-Verlag, Berlin, 13-30.

Maibaum, T. S. E. and Turski, W. M. (1984). On what exactly is going on when software is developed step-by-step. *tProc. 7h Intern. Conf. on Software Engin.* IEEE Computer Society, Los Angeles, 528-533.

Maibaum, T. S. E, Veloso, P. A. S. and Sadler, M. R. (1985). A theory of abstract data types for program development: bridging the gap?. In Ehrig, H., Floyd, C., Nivat, M. and Thatcher, J. eds. *Formal Methods and Software Development; vol. 2: Colloquium on Software Engineering.* Springer-Verlag, Berlin, 214-230.

Maibaum, T. S. E, Veloso, P. A. S. and Sadler, M. R. (1991). A logical approach to specification and implementation of abstract data types. Imperial College of Science, Technology and Medicine, Dept. of Computing Res. Rept. DoC 91/47, London.

Manna, Z. (1974). *The Mathematical Theory of Computation.* McGraw-Hill, New York.

Parnas, D. L. (1979). Designing software for ease of extension and contraction. *IEEE Trans. Software Engin.*, **5** (2), 128-138.

Shoenfield, J. R. (1967). *Mathematical Logic.* Addison-Wesley, Reading.

Smith, D. R. (1985).The Design of Divide and Conquer Algorithms. *Science Computer Programming*, **5** 37-58.

Smith, D. R. (1990). Algorithm theories and design tactics". *Science of Computer Programming.*, **14**, 305-321.

Smith, D. R. (1992). Constructing specification morphisms. Kestrel Institute, Tech. Rept. KES.U.92.1, Palo Alto.

Turski, W. M and Maibaum, T. S. E. (1987). *The Specification of Computer Programs*. Addison-Wesley, Wokingham, .

van Dalen, D. (1989). *Logic and Structure* (2nd edn, 3rd prt). Springer-Verlag, Berlin.

Veloso, P. A. S. (1984). Outlines of a mathematical theory of general problems. *Philosophia Naturalis*, **21** (2/4), 354-362.

Veloso, P. A. S. (1985). On abstraction in programming and problem solving. *2nd Intern. Conf. on Systems Research, Informatics and Cybernetics*. Baden-Baden.

Veloso, P. A. S. (1987). *Verificação e Estruturação de Programas com Tipos de Dados*. Edgard Blücher, São Paulo.

Veloso, P. A. S. (1987). On the concepts of problem and problem-solving method. *Decision Support Systems*, **3** (2), 133-139.

Veloso, P. A. S. (1988). Problem solving by interpretation of theories. In Carnielli, W. A. ; Alcântara, L. P. eds. *Methods and Applications of Mathematical Logic*. American Mathematical Society, Providence, . 241-250.

Veloso, P. A. S. (1992). On the modularisation theorem for logical specifications: its role and proof. PUC - RJ, Dept. Informática Res. Rept. MCC 17/92, Rio de Janeiro.

Veloso, P. A. S., Maibaum, T. S. E. and Sadler, M. R. (1985). Program development and theory manipulation. In *Proc. 3rd Intern. Workshop on Software Specification and Design*. IEEE Computer Society, Los Angeles, 228-232.

Veloso, P. A. S. and Maibaum, T. S. E. (1992). On the Modularisation Theorem for logical specifications. Imperial College of Science, Technology & Medicine, Dept. of Computing Res. Rept. DoC 92/35, London.

Veloso, P. A. S. and Veloso. S. R. M. (1981). Problem decomposition and reduction: applicability, soundness, completeness. In Trappl, R.; Klir, J. ; Pichler, F. eds. *Progress in Cybernetics and Systems Research*. Hemisphere, Washington, DC, 199-203.