JOURNAL OF COMPUTER AND SYSTEM SCIENCES **34**, 150–178 (1987)

On Observational Equivalence and Algebraic Specification*

DONALD SANNELLA AND ANDRZEJ TARLECKI[†]

Department of Computer Science, University of Edinburgh, Edinburgh EH9 3JZ, United Kingdom

Received August 22, 1985; revised February 24, 1986

The properties of a simple and natural notion of observational equivalence of algebras and the corresponding specification-building operation are studied. We begin with a definition of observational equivalence which is adequate to handle reachable algebras only, and show how to extend it to cope with unreachable algebras and also how it may be generalised to make sense under an arbitrary institution. Behavioural equivalence is treated as an important special case of observational equivalence, and its central role in program development is shown by means of an example. © 1987 Academic Press, Inc.

1. INTRODUCTION

Probably the most exciting potential application of formal specifications is to the formal development of programs by gradual refinement from a high-level specification to a low-level "program" or "executable specification" as in OBJ [30] or HOPE [10]. Each refinement step embodies some design decisions (such as choice of data representation) under the requirement that behaviour must be preserved. If each refinement step can be proved correct, then the program which results is guaranteed to satisfy the original specification.

This paper studies what is meant by 'behaviour" in the context of algebraic specifications. Intuitively, the behaviour of a program is determined just by the answers which are obtained from computations the program may perform. We may say (informally) that two Σ-algebras are *behaviourally equivalent* with respect to a set *OBS* of *observable sorts* if it is not possible to distinguish between them by evaluating Σ-terms which produce a result of observable sort. For example, suppose Σ contains the sorts *nat*, *bool*, and *bunch* and the operations *empty*: $\to$ *bunch*, *add*: *nat*, *bunch* $\to$ *bunch* and $\in$: *nat*, *bunch* $\to$ *bool* (as well as the usual operations on *nat* and *bool*), and suppose A and B are Σ-algebras with

$$|A_{bunch}| = \text{the set of finite sets of natural numbers}$$

$$|B_{bunch}| = \text{the set of finite lists of natural numbers}$$

* An extended abstract of this paper appeared in "Proceedings, 10th Colloq. on Trees in Algebra and Programming, Joint Conf. on Theory and Practice of Software Development, Berlin, March 1985," Lect. Notes in Comput. Sci. Vol. 185, pp. 308–322, Springer, New York/Berlin, 1985.

[†] Present address: Institute of Computer Science, Polish Academy of Sciences, Warsaw.

0022-0000/87 $3.00

with the operations and the remaining carriers defined in the obvious way (but B does *not* contain operations like *cons*, *car*, and *cdr*). Then A and B are behaviourally equivalent with respect to $\{bool\}$, since every term of sort *bool* has the same value in both algebras (the interesting terms are of the form $m \in add(a_1, ..., add(a_n, empty) ...))$. Note that A and B are not isomorphic.

In the above we assume that the only observations (or experiments) we are allowed to perform are to test whether the results of computations are equal. In this paper we deal with the more general situation in which observations may be arbitrary logical formulae. We discuss a notion of *observational equivalence* in which two algebras are observationally equivalent if they both give the same answers to any observation from a prespecified set. Similar ideas have appeared in [41], and a related approach to observational equivalence of concurrent processes was studied in [12].

Observational equivalence (or more specifically, behavioural equivalence) seems to be a concept which is fundamental to programming methodology. For example:

Data Abstraction

A practical advantage of using abstract data types in the construction of programs is that the implementation of abstractions by program modules need not be fixed. A different module using different algorithms and/or different data structures may be substituted without changing the rest of the program provided that the new module is behaviourally equivalent to the module it replaces (with respect to the non-encapsulated types). ADJ [31] have suggested that "abstract" in "abstract data type" means "up to isomorphism"; we understand it to mean "up to behavioural equivalence."[1]

Program Specification

One way of specifying a program is to describe the desired input/output behaviour in some concrete way, e.g., by constructing a very simple program which exhibits the desired behaviour. Any program which is behaviourally equivalent to the sample program with respect to the primitive types of the programming language satisfies the specification. This is called an *abstract model specification* [37]. In general, specifications under the usual algebraic approaches are not abstract enough; it is either difficult, as in Clear [8] or impossible, as in the initial

[1] It is not our intention to quibble over terminology here. We only wish to suggest that the use of the word "abstract" in "abstract data type," meaning "independent of representation" according to [31], is more accurately reflected by the notion of behavioural equivalence than by isomorphism as was suggested there. This seems to be consistent with the use of the term in languages like CLU [36] (where abstract data types are called *clusters*). In [28, 29] it has been suggested that "abstract data type" is an appropriate term for an isomorphism class of algebras while "abstract machine" refers to a behavioural equivalence class of algebras. Then a CLU cluster would correspond to an abstract machine. Since the motivation is really to capture algebraically the idea embodied in CLU clusters, we are in agreement with Goguen and Meseguer although we choose to use a different terminology.

algebra approach of [31] and the final algebra approach of [52] to specify sets of natural numbers in such a way that both A and B above are models. The kernel specification language ASL [47] provides a specification-building operation **abstract** which when applied to a specification SP relaxes interpretation to all those algebras which are observationally equivalent to a model of SP with respect to the given set of "equational" observations. With a properly chosen set of observations, this gives *behavioural abstraction.*

Stepwise Refinement

A formalisation of stepwise refinement requires a precise definition of the notion of refinement, i.e., of the *implementation* of one specification by a lower level specification. In the context of a specification language which includes an operation like behavioural abstraction, it is possible to adopt a very simple definition of implementation (see Sect. 5 for details). This notion of implementation has two very desirable properties (vertical and horizontal composability, see [25]) which permit the development of programs from specifications in a gradual and modular fashion. An alternative approach which illustrates the same point is to use a definition of implementation which implicitly involves behavioural equivalence, as in [28, 48].

This paper establishes a number of basic definitions and results concerning observational equivalence in an attempt to provide a sound foundation for its application to problems such as those indicated above. We begin by treating in Section 2 the case in which observations are logical formulae containing no free variables. We define observational equivalence of algebras and prove that it has properties such as antimonotonicity (with respect to the set of observations) and coherence with translation of algebras and formulae along signature morphisms. We define in terms of observational equivalence a specification-building operation (**abstract**) which performs observational abstraction and prove that it is, e.g., idempotent and satisfies certain identities. We try to characterise the effect that **abstract** has on the model class of a specification; in the case of first-order logic and specifications having a simple form we can characterise it exactly. It turn out that observations without free variables are sufficient if we are only interested in reachable algebras, and if we are willing to use the infinitary logic $L_{\infty\infty}$ then such observations are sufficient to deal with all algebras.

We generalise this material in two different dimensions; Section 3 discusses observations which contain free variables (to handle "junk" in unreachable algebras without resorting to infinitary logic) and Section 6 shows how the definitions can be generalised to make sense under an arbitrary logical system (or *institution* [26]). Almost all the results of Section 2 continue to hold under both of these generalisations. Section 4 deals with the problem of proving theorems about structured specifications in the context of observational abstraction, and gives an inference rule for reasoning about specifications built using the **abstract** operation. Section 5 discusses behavioural equivalence as an important special case of observational equivalence. A simple notion of implementation is defined, and we demonstrate the role of behavioural equivalence in program development by carrying out one refinement step in the development of a fragment of an optimising compiler from its specification.

We assume that the reader is familiar with the basic algebraic notions presented in, e.g., [31] (cf. [9]) as well as basic notions of logic as in, e.g., [18] including some infinitary logic, see [35].[2] In addition, Section 6 assumes some knowledge of category theory, see [1 or 38].

2. Observational Equivalence: The Ground Case

What is an observation on an algebra? In the axiomatic framework, the most natural choice is to take logical formulae as observations; the result of an observation on an algebra is just the truth or falsity of the formula in the algebra. The kind of formulae we use dictates the kinds of observations we are allowed to make on algebras. On the other hand, the kinds of observations we want to make on algebras dictates the kind of formulae we need, that is, the logic we should use.

For example, if we want only to examine results of computations, the natural choice is equations which allow us to compare the values of terms. Another natural choice is first-order predicate calculus which allows us to distinguish between, e.g., closed and open intervals of rationals (the observation/formula $\forall x.\exists y.x < y$ yields true in the latter and false in the former). Another choice is an infinitary logic such as $L_{\omega_1\omega}$ which allows us to check, e.g., reachability of algebras (that is, whether all elements of the algebra are values of ground terms). Note that the latter two kinds of observations are not computationally based; they are at a more abstract level, i.e., they describe algebras rather than computations in algebras. Still another kind of formulae is necessary if we want to deal with problems of concurrency or other non-functional "facets" of programming languages (see [40]), but we do not consider such issues in this paper.

For the moment, we do not want to commit ourselves to any particular logic, and so we leave the notion of "formula" undefined. (In fact, all our definitions work in an even more general setting; see Sect. 6.) The reader may feel more comfortable imagining that we are talking about first-order logic.

We use the term "formula" rather than "sentence" to indicate the possible presence of free variables to name elements which are not values of ground terms ("junk"). Free variables introduce some complications which we postpone to the next section. We will assume for the remainder of this section that formulae contain no free variables; we call these *ground observations* or *sentences*. The following definition corresponds directly to the definition of *elementary equivalence* in [41].

[2] In fact, as far as infinitary logic is concerned we use only the convention that if α is a regular infinite cardinal and β is a cardinal either 0 or $\omega \leqslant \beta \leqslant \alpha$, then $L_{\alpha\beta}$ denotes the language of first-order logic extended by allowing formulae to include conjunctions and disjunctions of sets of formulae of cardinality $<\alpha$ and quantification over sets of variables of cardinality $<\beta$. So, $L_{\omega_1\omega}$ includes countable $(<\omega_1)$ conjunctions and disjunctions but quantification is allowed only for finite $(<\omega)$ sets of variables.

DEFINITION. Let Σ be a signature, Φ a set of Σ-sentences, and let A, B be Σ-algebras. A and B are *observationally equivalent with respect to* Φ, written $A \equiv_{\Phi} B$, if for any $\varphi \in \Phi$, $A \models \varphi$ iff $B \models \varphi$.

Easy Facts

FACT 1. *For any signature Σ and set Φ of Σ-sentences, $\equiv_{\Phi}$ is an equivalence relation on the class of Σ-algebras.*

FACT 2. *For any signature Σ and sets Φ, Φ' of Σ-sentences, $\Phi \supseteq \Phi'$ implies $\equiv_{\Phi} \subseteq \equiv_{\Phi'}$.*

FACT 3. *For any signature Σ, family $\{\Phi_i\}_{i \in I}$ of sets of Σ-sentences, and Σ-algebras A, B, $A \equiv_{\Phi_i} B$ for all $i \in I$ implies $A \equiv_{\Phi} B$, where $\Phi = \bigcup_{i \in I} \Phi_i$.*

Two algebras are observationally equivalent w.r.t. Φ if they satisfy exactly the same sentences of Φ. Note that this remains true if we consider not only the sentences of Φ but also their negations and conjunctions, possibly infinite or empty ($\bigwedge\varnothing$ is true). We can also add to Φ sentences equivalent to the ones already in Φ, and so everything which is definable in terms of negation and conjunction as well (disjunctions, implications, etc.). For any set Φ of Σ-sentences, let $\mathrm{Cl}(\Phi)$ denote the closure of Φ under negation, conjunction, and equivalence, insofar as the logic in use allows.

FACT 4. $\equiv_{\Phi} = \equiv_{\mathrm{Cl}(\Phi)}$.

Note that this implies that the premise $\Phi \supseteq \Phi'$ in Fact 2 may be replaced by the weaker condition $\mathrm{Cl}(\Phi) \supseteq \Phi'$.

A *signature morphism* $\sigma: \Sigma \to \Sigma'$ is a renaming of the sorts and operations in Σ to those of Σ' which preserves the argument and result sorts of operations. This induces in a natural way a translation of Σ-terms to Σ'-terms and of Σ-sentences to Σ'-sentences; if φ is a Σ-sentence, then $\sigma(\varphi)$ denotes its translation to a Σ'-sentence. A signature morphism $\sigma: \Sigma \to \Sigma'$ also induces a *σ-reduct* functor translating Σ'-algebras to Σ-algebras; if A' is a Σ'-algebra then $A'|_{\sigma}$ denotes its σ-reduct. For the exact definitions of these notions see, e.g., [44, Sect. 2]. These translations satisfy the following condition (see [26] and also Section 6):

SATISFACTION CONDITION. For any Σ-sentence φ and Σ'-algebra A', $A'|_{\sigma} \models \varphi$ iff $A' \models \sigma(\varphi)$.

This gives immediately

FACT 5. *For any signature morphism $\sigma: \Sigma \to \Sigma'$, set Φ of Σ-sentences and Σ'-algebras A', B', $A' \equiv_{\sigma(\Phi)} B'$* iff *$A'|_{\sigma} \equiv_{\Phi} B'|_{\sigma}$, where $\sigma(\Phi) = \{\sigma(\varphi) \mid \varphi \in \Phi\}$.*

This says that observational equivalence is coherent with translation along signature morphisms. We can also show that observational equivalence is preserved under combination of "independent" algebras.

Let $\Sigma 1$ and $\Sigma 2$ be disjoint signatures and let $\Sigma 1 + \Sigma 2$ be their (disjoint) union.[3] For any $\Sigma 1$-algebra $A1$ and $\Sigma 2$-algebra $A2$, let $\langle A1, A2\rangle$ be the unique $(\Sigma 1 + \Sigma 2)$-algebra such that $\langle A1, A2\rangle|_{\iota 1} = A1$ and $\langle A1, A2\rangle|_{\iota 2} = A2$, where $\iota 1$ and $\iota 2$ are the inclusions of $\Sigma 1$ and $\Sigma 2$ (respectively) into $\Sigma 1 + \Sigma 2$. Note that all $(\Sigma 1 + \Sigma 2)$-algebras are of this form.

FACT 6. *For any disjoint signatures $\Sigma 1$, $\Sigma 2$, sets of $\Sigma 1$-sentences $\Phi 1$ and $\Sigma 2$-sentences $\Phi 2$, $\Sigma 1$-algebras $A1$, $B1$ and $\Sigma 2$-algebras $A2$, $B2$,*

$$\langle A1, A2\rangle \equiv_{\iota 1(\Phi 1)\cup \iota 2(\Phi 2)} \langle B1, B2\rangle \textit{ iff } A1 \equiv_{\Phi 1} B1 \textit{ and } A2 \equiv_{\Phi 2} B2.$$

Proof. ($\Leftarrow$) By Fact 5, if $A1 \equiv_{\Phi 1} B1$ then $\langle A1, A2\rangle \equiv_{\iota 1(\Phi 1)} \langle B1, B2\rangle$. Similarly, if $A2 \equiv_{\Phi 2} B2$ then $\langle A1, A2\rangle \equiv_{\iota 2(\Phi 2)} \langle B1, B2\rangle$, and so by Fact 3, $\langle A1, A2\rangle \equiv_{\iota 1(\Phi 1)\cup \iota 2(\Phi 2)} \langle B1, B2\rangle$.

($\Rightarrow$) By Fact 2, $\langle A1, A2\rangle \equiv_{\iota 1(\Phi 1)\cup \iota 2(\Phi 2)} \langle B1, B2\rangle$ implies $\langle A1, A2\rangle \equiv_{\iota 1(\Phi 1)} \langle B1, B2\rangle$, and so by Fact 5, $A1 \equiv_{\Phi 1} B1$. By the same argument, $A2 \equiv_{\Phi 2} B2$. ▌

A specification describes a collection of models of the same signature. To formalise this, for any specification SP let $\mathrm{Sig}[SP]$ denote its signature and $\mathrm{Mod}[SP]$ denote the class of its models, which are $\mathrm{Sig}[SP]$-algebras. The notion of observational equivalence give rise to a very powerful specification-building operation:

DEFINITION. For any specification SP and set Φ of $\mathrm{Sig}[SP]$-sentences

$$\mathrm{Sig}[\textbf{abstract } SP \textbf{ w.r.t. } \Phi] = \mathrm{Sig}[SP]$$

$$\mathrm{Mod}[\textbf{abstract } SP \textbf{ w.r.t. } \Phi] = \{A \mid A \equiv_{\Phi} B \text{ for some } B \in \mathrm{Mod}[SP]\}.$$

Informally, **abstract** SP **w.r.t.** Φ is a specification which admits any model which is observationally equivalent to some model of SP. This provides a way of abstracting away from certain details of a specification (see [44, 47]).

Easy Facts

FACT 7. *For any specification SP* and set 8 of $\mathrm{Sig}[SP]$-*sentences,*

$$\mathrm{Mod}[SP] \subseteq \mathrm{Mod}[\textbf{abstract } SP \textbf{ w.r.t. } \Phi].$$

FACT 8. *For any specification SP* and set 8 of $\mathrm{Sig}[SP]$-*sentences,*

$$\mathrm{Mod}[\textbf{abstract } (\textbf{abstract } SP \textbf{ w.r.t. } \Phi) \textbf{ w.r.t. } \Phi] = \mathrm{Mod}[\textbf{abstract } SP \textbf{ w.r.t. } \Phi].$$

[3] Or *coproduct* in categorical terms; this footnote is for the benefit of Sect. 6.

FACT 9. *For any specification* SP and sets 8, Φ' *of* $\mathrm{Sig}[SP]$*-sentences,*

$$\mathrm{Cl}(\Phi) \supseteq \Phi' \textit{ implies } \mathrm{Mod}[\textbf{abstract } SP \textbf{ w.r.t. } \Phi] \subseteq \mathrm{Mod}[\textbf{abstract } SP \textbf{ w.r.t. } \Phi'].$$

FACT 10. *For any specifications* SP, SP' *such that* $\mathrm{Sig}[SP] = \mathrm{Sig}[SP']$ *and set* Φ *of* $\mathrm{Sig}[SP]$*-sentences,* $\mathrm{Mod}[SP] \subseteq \mathrm{Mod}[SP']$ *implies*

$$\mathrm{Mod}[\textbf{abstract } SP \textbf{ w.r.t. } \Phi] \subseteq \mathrm{Mod}[\textbf{abstract } SP' \textbf{ w.r.t. } \Phi].$$

Using the above facts we may derive simple identities which allow us to transform specifications involving **abstract**. For example:

FACT 11. *For any specification* SP and sets 8, Φ' *of* $\mathrm{Sig}[SP]$*-sentences,*

(a) $\mathrm{Mod}[\textbf{abstract } SP \textbf{ w.r.t. } \Phi \cup \Phi'] \subseteq \mathrm{Mod}[\textbf{abstract } (\textbf{abstract } SP \textbf{ w.r.t. } \Phi) \textbf{ w.r.t. } \Phi']$
$\subseteq \mathrm{Mod}[\textbf{abstract } SP \textbf{ w.r.t. } \mathrm{Cl}(\Phi) \cap \mathrm{Cl}(\Phi')]$.

(b) $\mathrm{Cl}(\Phi) \supseteq \Phi'$ *implies*

$$\begin{aligned}\mathrm{Mod}[\textbf{abstract } SP \textbf{ w.r.t. } \Phi'] &= \mathrm{Mod}[\textbf{abstract } (\textbf{abstract } SP \textbf{ w.r.t. } \Phi) \textbf{ w.r.t. } \Phi'] \\ &= \mathrm{Mod}[\textbf{abstract } (\textbf{abstract } SP \textbf{ w.r.t. } \Phi') \textbf{ w.r.t. } \Phi].\end{aligned}$$

Proof Example (*first part of* (b)).

$\mathrm{Mod}[\textbf{abstract } SP \textbf{ w.r.t. } \Phi'] \subseteq \mathrm{Mod}[\textbf{abstract } (\textbf{abstract } SP \textbf{ w.r.t. } \Phi) \textbf{ w.r.t. } \Phi']$
by Facts 7 and 10;

$\mathrm{Mod}[\textbf{abstract } SP \textbf{ w.r.t. } \Phi'] \supseteq \mathrm{Mod}[\textbf{abstract } SP \textbf{ w.r.t. } \Phi]$
by Fact 9, since $\mathrm{Cl}(\Phi) \supseteq \Phi'$;

so

$\mathrm{Mod}[\textbf{abstract } SP \textbf{ w.r.t. } \Phi'] = \mathrm{Mod}[\textbf{abstract } (\textbf{abstract } SP \textbf{ w.r.t. } \Phi') \textbf{ w.r.t. } \Phi']$
by Fact 8,

$\supseteq \mathrm{Mod}[\textbf{abstract } (\textbf{abstract } SP \textbf{ w.r.t. } \Phi) \textbf{ w.r.t. } \Phi']$
by Fact 10. ▌

The following counterexample shows that the second equality in (b) above need not hold if $\mathrm{Cl}(\Phi) \not\supseteq \Phi'$:

COUNTEREXAMPLE. Consider the signature Σ having one sort and three constants a, b, c. Let SP be a specification with $\mathrm{Sig}[SP] = \Sigma$ such that $\mathrm{Mod}[SP] = \{A \mid A \models a = b = c\}$. Let $\Phi = \{a = b, b = c\}$ and $\Phi' = \{a = c\}$. Now, $\mathrm{Mod}[\textbf{abstract } SP \textbf{ w.r.t. } \Phi] = \mathrm{Mod}[SP]$ and so any algebra in $\mathrm{Mod}[\textbf{abstract } (\textbf{abstract } SP \textbf{ w.r.t. } \Phi) \textbf{ w.r.t. } \Phi']$ satisfies $a = c$. On the other hand, $\mathrm{Mod}[\textbf{abstract } SP \textbf{ w.r.t. } \Phi']$ contains, for example, algebras in which $a = c$ but $a \neq b$

($\neq c$). Thus Mod[**abstract** (**abstract** SP **w.r.t.** Φ') **w.r.t.** Φ] contains algebras in which all three constants have different values. This shows

$$\mathrm{Mod}[\textbf{abstract}\ (\textbf{abstract}\ SP\ \textbf{w.r.t.}\ \Phi)\ \textbf{w.r.t.}\ \Phi'] \neq \mathrm{Mod}[\textbf{abstract}\ (\textbf{abstract}\ SP\ \textbf{w.r.t.}\ \Phi')\ \textbf{w.r.t.}\ \Phi].$$

Note that this counterexample also shows that the inclusions in (a) above may be proper.

Every algebraic specification language provides an operation for specifying the class of models of a given signature which satisfy a given set of axioms, that is,

DEFINITION. For any signature Σ and set Δ of Σ-sentences, $\langle \Sigma, \Delta \rangle$ is a *basic specification* and

$$\mathrm{Sig}[\langle \Sigma, \Delta \rangle] = \Sigma$$
$$\mathrm{Mod}[\langle \Sigma, \Delta \rangle] = \{A \mid A \text{ is a } \Sigma\text{-algebra and } A \models \Delta\}.$$

For any signature Σ and class K of Σ-algebras, let $\mathrm{Th}(K)$ denote the set of all Σ-sentences which hold in K. Note that $K \subseteq \mathrm{Mod}[\langle \Sigma, \mathrm{Th}(K) \rangle]$ but the converse inclusion is true only for classes K definable by basic specifications.

FACT 12. *For any specification SP* with $\mathrm{Sig}[SP] = \Sigma$ *and set* Φ *of* Σ*-sentences,*

$$\mathrm{Mod}[\textbf{abstract}\ SP\ \textbf{w.r.t.}\ \Phi] \subseteq \mathrm{Mod}[\langle \Sigma, \mathrm{Th}(\mathrm{Mod}[SP]) \cap \mathrm{Cl}(\Phi) \rangle].$$

Proof. Let $A \in \mathrm{Mod}[\textbf{abstract}\ SP\ \textbf{w.r.t.}\ \Phi]$, i.e., $A \equiv_\Phi B$ for some $B \in \mathrm{Mod}[SP]$. Obviously, $B \models \mathrm{Th}(\mathrm{Mod}[SP])$ and so for any $\varphi \in \mathrm{Th}(\mathrm{Mod}[SP]) \cap \mathrm{Cl}(\Phi)$, $B \models \varphi$. Hence, by Fact 4, $A \models \varphi$ so $A \in \mathrm{Mod}[\langle \Sigma, \mathrm{Th}(\mathrm{Mod}[SP]) \cap \mathrm{Cl}(\Phi) \rangle]$. ∎

In the following, we try to further characterise how **abstract** works for classes of models definable by basic specifications. For any signature Σ and set Δ of Σ-sentences, let $\Delta^* = \mathrm{Th}(\mathrm{Mod}[\langle \Sigma, \Delta \rangle])$ be the *closure of* Δ *under consequence.*

FACT 13. *In first-order logic, for any signature* Σ *and sets* Δ, Φ *of* Σ*-sentences,*

$$\mathrm{Mod}[\textbf{abstract}\ \langle \Sigma, \Delta \rangle\ \textbf{w.r.t.}\ \Phi] = \mathrm{Mod}[\langle \Sigma, \Delta^* \cap \mathrm{Cl}(\Phi) \rangle].$$

Proof. ($\subseteq$) Obvious by the previous fact.

($\supseteq$) Since for unsatisfiable Δ the containment holds (because false $\in \mathrm{Cl}(\Phi)$), we assume that Δ has a model. Let $A \in \mathrm{Mod}[\langle \Sigma, \Delta^* \cap \mathrm{Cl}(\Phi) \rangle]$. Consider $\Psi = \{\varphi \in \mathrm{Cl}(\Phi) \mid A \models \varphi\}$. We want to show that $\Delta^* \cup \Psi$ has a model. Suppose not, i.e., that $\Delta^* \cup \Psi$ is not satisfiable. Then by the compactness theorem of first-order logic and by the fact that Ψ is closed under conjunction there is a $\psi \in \Psi$ such that

$\Delta^* \cup \{\psi\}$ has no model. Hence $\Delta^* \models \neg \psi$, which, since Δ^* is closed under logical consequence implies that $\neg\psi \in \Delta^*$. Thus $\neg\psi \in \Delta^* \cap \mathrm{Cl}(\Phi)$ ($\mathrm{Cl}(\Phi)$ is closed under negation) and so $A \models \neg\psi$, which contradicts $\psi \in \Psi$.

This proves that $\Delta^* \cup \Psi$ has a model, say B. Let $\varphi \in \Phi$. If $A \models \varphi$ then $\varphi \in \Psi$ and so $B \models \varphi$. If $A \not\models \varphi$, i.e., $A \models \neg\varphi$, then $\neg\varphi \in \Psi$ and so $B \models \neg\varphi$, i.e., $B \not\models \varphi$. Thus $A \equiv_\Phi B$, which shows that $A \in \mathrm{Mod}[$**abstract** $\langle\Sigma, \Delta\rangle$ **w.r.t.** $\Phi]$, since $B \in \mathrm{Mod}[\langle\Sigma, \Delta\rangle]$. ▌

The above fact may be interpreted as a statement that $\mathrm{Cl}(\Phi)$ gives a complete characterisation of observational equivalence with respect to Φ. Another formulation of this is

FACT 14. *In first-order logic, for any signature Σ and sets $\Phi1$, $\Phi2$ of Σ-sentences, $\equiv_{\Phi1} = \equiv_{\Phi2}$ iff $\mathrm{Cl}(\Phi1) = \mathrm{Cl}(\Phi2)$.*

Proof. ($\Leftarrow$) Follows directly from Fact 4.

($\Rightarrow$) Assume that $\mathrm{Cl}(\Phi1) \neq \mathrm{Cl}(\Phi2)$. By symmetry, we may assume that there is $\varphi \in \mathrm{Cl}(\Phi1)$ such that $\varphi \notin \mathrm{Cl}(\Phi2)$. To complete the proof it is enough to show that

$$\mathrm{Mod}[\textbf{abstract } \langle\Sigma, \{\varphi\}\rangle \textbf{ w.r.t. } \Phi1] \neq \mathrm{Mod}[\textbf{abstract } \langle\Sigma, \{\varphi\}\rangle \textbf{ w.r.t. } \Phi2].$$

By Fact 13, $\mathrm{Mod}[$**abstract** $\langle\Sigma, \{\varphi\}\rangle$ **w.r.t.** $\Phi1] = \mathrm{Mod}[\langle\Sigma, \{\varphi\}^* \cap \mathrm{Cl}(\Phi1)\rangle] = \mathrm{Mod}[\langle\Sigma, \{\varphi\}\rangle]$ (since $\varphi \in \mathrm{Cl}(\Phi1)$). On the other hand, $\mathrm{Mod}[$**abstract** $\langle\Sigma, \{\varphi\}\rangle$ **w.r.t.** $\Phi2] = \mathrm{Mod}[\langle\Sigma, \{\varphi\}^* \cap \mathrm{Cl}(\Phi2)\rangle]$. We show that there is $A \in \mathrm{Mod}[\langle\Sigma, \{\varphi\}^* \cap \mathrm{Cl}(\Phi2)\rangle]$ such that $A \not\models \varphi$. Suppose otherwise, i.e., that $(\{\varphi\}^* \cap \mathrm{Cl}(\Phi2)) \cup \{\neg\varphi\}$ is not satisfiable. Then by the compactness theorem for first-order logic, since $\{\varphi\}^* \cap \mathrm{Cl}(\Phi2)$ is closed under finite conjunctions, for some sentence $\psi \in \{\varphi\}^* \cap \mathrm{Cl}(\Phi2)$, $\psi \models \varphi$. Thus φ and ψ are equivalent ($\varphi \models \psi$ by the definition of $\{\varphi\}^*$) and so since $\psi \in \mathrm{Cl}(\Phi2)$, $\varphi \in \mathrm{Cl}(\Phi2)$ as well, which contradicts our assumptions. ▌

Ground observations are powerful enough if we are only interested in reachable (subalgebras of) algebras and we do not want to distinguish between isomorphic algebras, provided that our logic is at least capable of expressing ground equations (i.e., equations between terms without variables).

FACT 15. *For any signature Σ and Σ-algebras A, B, $A \equiv_{\mathrm{GEQ}(\Sigma)} B$ iff A and B have isomorphic reachable subalgebras, where* $\mathrm{GEQ}(\Sigma)$ *is the set of all* ground *Σ-equations.*

Proof. ($\Leftarrow$) Obvious.

($\Rightarrow$) Consider the function which maps the value of any ground Σ-term in A to the value of this term in B. It is easy to see that since $A \equiv_{\mathrm{GEQ}(\Sigma)} B$ this is well defined and, moreover, that this function is an isomorphism between the reachable subalgebras of A and B. ▌

3. Observational Equivalence: The General Case

In the last section we dealt with observational equivalence based on ground observations only (formally, on formulae without free variables). As Fact 15 indicates, this is quite satisfactory when we restrict our considerations to reachable algebras. If we want to deal with algebras containing "junk," things become more complicated.

Why do we bother about non-reachable algebras? First, when dealing with parameterised specifications it is usual to consider examples in which some sorts have no generators at all, but where we are interested in algebras having the associated carriers non-empty. This is shown by standard examples such as Stack-of-X, where X is an arbitrary set. Second, when we view algebras from different levels of abstraction we view them with respect to different sets of operations. It is then natural that an algebra which is reachable at a certain level of abstraction becomes non-reachable when viewed from a higher level. A more technical but related point is that there is no natural definition of the specification-building operation **derive** [8, 44] (which can be used to "forget" operations), if models of the result are required to be reachable. Finally, there are examples [47] in which unreachable elements can be useful in constructing specifications; an element which is unreachable at one stage of the construction can become reachable and useful at a later stage.

It should be noted that if the logic we are working in is sufficiently powerful then we can identify algebras up to isomorphism using only ground observations. According to Scott's theorem [49] this may be achieved using $L_{\omega_1\omega}$ for countable algebras. Using an even more powerful logic the same may be done for arbitrary algebras.

Fact 16. *For any signature Σ and Σ-algebras A, there is a Σ-sentence $\zeta(A)$ of $L_{\infty\infty}$ such that for any Σ-algebra B, $B \models \zeta(A)$ iff $A \cong B$.*

Proof. Assume that Σ has only one sort; the general case is similar but notationally more cumbersome. Consider the formula

$$\zeta(A) =_{\text{def}} \exists |A| . (\bigwedge \mathscr{D}(A) \wedge \forall x . \bigvee \{x = a \mid a \in |A|\}),$$

where $|A|$ is the carrier of A, and $\mathscr{D}(A)$ is the first-order diagram of the expansion of A to a $\Sigma(|A|)$-algebra with the natural interpretation of the new constants (in fact, it is enough to take all ground $\Sigma(|A|)$-equations and -inequations which are true in the expansion of A). If $B \cong A$ then obviously $B \models \zeta(A)$. To see this consider the isomorphism from A to B as the valuation of variables $|A|$ in B.

Conversely, assume that B satisfies $\zeta(A)$. Thus, there is a valuation $v \colon |A| \to |B|$ such that $B \models_v \bigwedge \mathscr{D}(A) \wedge \forall x . \bigvee \{x = a \mid a \in |A|\}$ ($B \models_v \varphi$ means B satisfies φ under the valuation v; we are going to use this notation throughout the paper.). It is easy to see that v is an isomorphism between A and B:

— v is surjective since $B \models_v \forall x. \bigvee \{x = a \mid a \in |A|\}$.

— v is 1-1: for $a_1, a_2 \in |A|$, if $a_1 \neq a_2$ then $a_1 \neq a_2 \in \mathscr{D}(A)$ and so $B \models_v a_1 \neq a_2$, that is, $v(a_1) \neq v(a_2)$.

— v satisfies the homomorphism property: for any n-argument operation $f \in \Sigma$, $a_1, ..., a_n \in |A|$, $a \in |A|$, if $f_A(a_1, ..., a_n) = a$ then $f(a_1, ..., a_n) = a \in \mathscr{D}(A)$ and so $B \models_v f(a_1, ..., a_n) = a$, that is, $f_B(v(a_1), ..., v(a_n)) = v(a)$. ▮

Note from the construction that in order to handle algebras of cardinality α it is enough to consider formulae with quantifiers binding α variables. It seems to be possible to sharpen this result (requiring only quantifiers binding less than α variables) using some kind of "back-and-forth" construction as in the proof of Scott's theorem in [2].

However, in practice it is desirable to avoid use of infinitary logic (although [39] argues for an approach to specification in which infinitary logic is central). What we are trying to do in the following is to obtain a balance between the power of the logic in use (the simpler the logic, the better) and the simplicity of the definition of observational equivalence.

It is obvious that using ground equations as observations we are not able to talk about junk at all. If we use equations with universally quantified variables, although we are able to say something about junk we cannot always distinguish between algebras which are intuitively not equivalent. For example, the two algebras

$$\begin{tikzcd} A: & \circ \arrow[r, "f"] & \circ \arrow[r, "f"] & \cdots & B: & \circ \arrow[r, "f"] & \circ \arrow[r, "f"] & \cdots \\ & \circ \arrow[r, "f"'] & \circ \arrow[r, "f"'] & \cdots & & \circ \arrow[loop right, "f"] \end{tikzcd}$$

cannot be distinguished by any equation with universally quantified variables (neither of them satisfy $\forall x. f(x) = x$) although if we go to first-order logic then the formula $\exists x. f(x) = x$ distinguishes between them. But even in the framework of first-order logic using only closed sentences, we are not able to deal with junk in a satisfactory way. We cannot even express such a basic property as is existence. For example, it is well-known that the standard model of arithmetic (the natural numbers) and non-standard models (the natural numbers with junk) satisfy exactly the same set of first-order sentences. Thus, there is no set of ground first-order observations which can distinguish between standard and non-standard models of arithmetic. To distinguish between these models using ground observations we need $L_{\omega_1\omega}$.

We are going to extend the definitions of the previous section by allowing free variables in observations. The idea is that these provide a way of referring to otherwise unnameable values. For example, it should be intuitively clear (and will be formalised below) that the observation $f(x) = x$ with free variable x distinguishes between the two algebras A and B above, and the set of observations $\{x = succ^n(0) \mid n \geq 0\}$ with free variable x distinguishes between standard and non-standard models of arithmetic. As in logic, we need a valuation of the free variables into the algebra under consideration to provide these names with interpretations.

Given a signature Σ, a set X of variables (of sorts in Σ), a set $\Phi(X)$ of Σ-formulae with free variables in X, and two Σ-algebras A, B there are a number of possible ways to define $A \equiv_{\Phi(X)} B$. Here are some which are **not** satisfactory:

(1) For all $\varphi \in \Phi(X)$, $A \models \forall X.\varphi$ iff $B \models \forall X.\varphi$.

(2) For all valuations $v_A: X \to |A|$ and $v_B: X \to |B|$ and all $\varphi \in \Phi(X)$, $A \models_{v_A} \varphi$ iff $B \models_{v_B} \varphi$.

(3) There exist valuations $v_A: X \to |A|$ and $v_B: X \to |B|$ such that for all $\varphi \in \Phi(X)$, $A \models_{v_A} \varphi$ iff $B \models_{v_B} \varphi$.

Comments. (1) is basically equivalent to the use of ground observations.

(2) is too strong; the relation it defines is not even reflexive.

(3) is too weak; v_A and v_B can be chosen to ignore the parts of A and B which are inconvenient. If A and B have isomorphic reachable subalgebras then they are observationally equivalent under this definition for any set of observations.

In [47, 44], $A \equiv_{\Phi(X)} B$ was defined as follows:

(4) There exist surjective valuations $v_A: X \to |A|$ and $v_B: X \to |B|$ such that for all $\varphi \in \Phi(X)$, $A \models_{v_A} \varphi$ iff $B \models_{v_B} \varphi$.

The justification for this definition is that v_A and v_B identify "matching parts" of A and B; each part of A must match some part of B and vice versa. But there are some problems with this definition. Technically, this relation is restricted to comparing algebras of cardinality less than or equal to that of X because of the surjectivity requirement on v_A and v_B. Also, we have to exclude algebras with empty carriers, (at least) on sorts in which X is non-empty; otherwise the valuations v_A and/or v_B cannot exist. Finally, in the "general" case in which models and the logic are arbitrary (see [44] and also Sect. 6) this definition is rather messy and inelegant because of the difficulty of formulating in abstract terms the requirement of surjectivity.

We are going to concentrate on still another definition of observational equivalence. We define the observational equivalence relation in terms of a preorder.

DEFINITION. For any signature Σ, set X of variables of sorts in Σ, set $\Phi(X)$ of Σ-formulae with free variables in X, and Σ-algebras A, B, A is *observationally reducible to B w.r.t. $\Phi(X)$*, written $A \leqslant_{\Phi(X)} B$, if for any valuation $v_A: X \to |A|$ there exists a valuation $v_B: X \to |B|$ such that for all $\varphi \in \Phi(X)$, $A \models_{v_A} \varphi$ iff $B \models_{v_B} \varphi$.

FACT 17. *For any signature Σ, set X of variables of sorts in Σ, and set $\Phi(X)$ of Σ-formulae with free variables in X, $\leqslant_{\Phi(X)}$ is a preorder on the class of Σ-algebras.*

DEFINITION. For any signature Σ, set X of variables of sorts in Σ, set $\Phi(X)$ of Σ-formulae with free variables in X, and Σ-algebras A, B, A and B are *observationally equivalent w.r.t.* $\Phi(X)$, written $A \equiv_{\Phi(X)} B$, if $A \leqslant_{\Phi(X)} B$ and $B \leqslant_{\Phi(X)} A$.

Note that if the set X is empty, the above definition of observational equivalence reduces to the definition in the ground case. But if X is nonempty then even if $\Phi(X)$ contains no free variables it may be the case that observational equivalence with respect to $\Phi(X)$ is not the same as (ground) observational equivalence with respect to Φ because of problems caused by empty carriers.

Although we are not going to restate all of them formally here again, Facts 2–6 of Section 2 hold for the preorder $\leqslant_{\Phi(X)}$ and Facts 1–6 hold for the equivalence $\equiv_{\Phi(X)}$. For example, Fact 3 may be reformulated for $\leqslant_{\Phi(X)}$ here as follows:

FACT 3′. *For any signature Σ, family of mutually disjoint sets $\{X_i\}_{i\in I}$ of variables of sorts in Σ, family $\{\Phi_i\}_{i\in I}$ of sets of Σ-formulae such that for $i\in I$, Φ_i has free variables in X_i, and Σ-algebras A, B, $A \leqslant_{\Phi_i(X_i)} B$ for all $i \in I$ implies $A \leqslant_{\Phi(X)} B$, where $\Phi = \bigcup_{i\in I} \Phi_i$ and $X = \bigcup_{i\in I} X_i$.*

However, because of the problems which empty carriers may cause, we have to be careful with the opposite direction of this implication, that is, when discharging variables. Fact 2 should be reformulated as follows:

FACT 2′. *For any signature Σ, set X of variables of sorts in Σ, sets $\Phi(X)$ and $\Phi'(X)$ of Σ-formulae with free variables in X, and Σ-algebras A, B,*

$$\Phi(X) \supseteq \Phi'(X) \text{ and } A \leqslant_{\Phi(X)} B \qquad \text{implies} \qquad A \leqslant_{\Phi'(X)} B.$$

Note that Φ and Φ' must formally have the same set X of free variables, even if the formulae in the smaller set Φ' do not use all of them. We can discharge such unnecessary variables only if X contains other variables of the same sorts, or if the algebras we are dealing with are guaranteed to have non-empty carriers of these sorts.

As in the previous section, we can define a specification-building operation **abstract** in terms of observational equivalence with exactly the same semantics.

DEFINITION. For any specification SP, set X of variables of sorts in Sig[SP] and set $\Phi(X)$ of Sig[SP]-formulae with free variables in X,

$$\mathrm{Sig}[\textbf{abstract}\ SP\ \textbf{w.r.t.}\ \Phi(X)] = \mathrm{Sig}[SP]$$

$$\mathrm{Mod}[\textbf{abstract}\ SP\ \textbf{w.r.t.}\ \Phi(X)] = \{A \mid A \equiv_{\Phi(X)} B \text{ for some } B \in \mathrm{Mod}[SP]\}.$$

Facts 7–12 still hold under this more general definition. Facts 13 and 14 do not hold, and Fact 15 is not relevant.

Note that we can give a sharper formulation of the facts which involve forming the closure Cl(Φ) of a set of formulae Φ. In the presence of free variables, besides conjunctions and negations it is tempting to allow the introduction of quantifiers here. We can redefine $\mathrm{Cl}(\Phi(X))$ to be the closure of $\Phi(X)$ under negation, conjunction (possibly infinite), equivalence, and uniform quantification, that is, $\varphi \in \mathrm{Cl}(\Phi(X))$ implies $\forall X.\varphi \in \mathrm{Cl}(\Phi(X))$ and $\exists X.\varphi \in \mathrm{Cl}(\Phi(X))$. To prove that all the facts are still true with this new definition of $\mathrm{Cl}(\Phi(X))$, we have to show

FACT 18. $\equiv_{\mathrm{Cl}(\Phi(X))} \supseteq \equiv_{\Phi(X)}$.

Proof. Let A and B be algebras with $A \equiv_{\Phi(X)} B$. The only problem may be with quantifiers, but for this note that since $A \equiv_{\Phi(X)} B$, for $\varphi \in \Phi(X)$ we have $A \models_{v_A} \varphi$ for all $v_A : X \to |A|$ iff $B \models_{v_B} \varphi$ for all $v_B : X \to |B|$. And so $A \models \forall X.\varphi$ iff $B \models \forall X.\varphi$. Since $\exists X.\varphi$ is equivalent to $\neg\forall X.\neg\varphi$, this completes the proof. ∎

Note that only uniform quantification is allowed above. The above fact does not hold if we allow quantification over a proper subset of the set of free variables. For example, suppose $\Sigma =$ **sorts** *rat*, *bool* **opns** $<$: *rat*, *rat* $\to$ *bool*. Let A and B be Σ-algebras corresponding to, respectively, open and closed intervals of rational numbers. Now consider $\Phi(X) = \{x < y \mid x, y \in X\}$. Obviously $A \equiv_{\Phi(X)} B$ but $A \models \forall x.\exists y.x < y$ while $B \not\models \forall x.\exists y.x < y$.

The uniform quantification allowed in the closure $\mathrm{Cl}(\Phi(X))$ is not enough to guarantee that Fact 13 holds for observational abstraction with respect to open formulae, as shown by the following counterexample.

COUNTEREXAMPLE. Let Σ be a signature with one sort and one unary operation f. Let X be a countably infinite set of variables. Consider

$$\Delta = \{\forall x.\exists y.f(y) = x\}$$

$$\Phi(X) = \{x = y, f(x) = y \mid x, y \in X\}$$

$$A\colon \cdots \xrightarrow{f} \circ \xrightarrow{f} \circ \xrightarrow{f} \cdots \qquad B\colon \circ \xrightarrow{f} \circ \xrightarrow{f} \circ \xrightarrow{f} \cdots$$

Now, $A \models \Delta$, $A \not\equiv_{\Phi(X)} B$, and $B \notin \mathrm{Mod}[$**abstract** $\langle \Sigma, \Delta \rangle$ **w.r.t.** $\Phi(X)]$, but $B \models \Delta^* \cap \mathrm{Cl}(\Phi(X))$. To prove the last of these statements we use the following well-known lemma.

LEMMA. *For any Σ-algebra A, set of variables X, valuation $v\colon X \to |A|$, and quantifier-free (first-order logic) Σ-formula $\varphi(X)$, $A \models_v \varphi(X)$ iff $[v'(X')]_A \models_{v'} \varphi(X')$, where $X' \subseteq X$ is the set of variables which actually occur in φ, v' is v restricted to X', $v'(X') = \{v'(x) \mid x \in X'\}$, and $[v'(X')]_A$ is the least subalgebra of A which contains $v'(X')$.*

Now, for any quantifier-free Σ-formula $\varphi(X)$, the set $X' \subseteq X$ of variables which actually appear in φ is finite, and so for any valuation $v\colon X \to |A|$, $[v'(X')]_A$ is isomorphic to B. Thus, if $A \models \forall X.\varphi(X)$ then $B \models \forall X.\varphi(X)$ as well, and if $A \models \exists X.\varphi(X)$ then $B \models \exists X.\varphi(X)$. Using this it is easy to prove that for any Σ-sentence ψ which involves only uniform quantification, if $A \models \psi$ then $B \models \psi$ as well. This shows that $B \models \Delta^* \cap \mathrm{Cl}(\Phi(X))$.

Fact 14 does not hold under the new definition of closure either, as the following counterexample demonstrates.

COUNTEREXAMPLE. Let Σ be as in the example above. Let $X =_{\mathrm{def}} \{x\}$. Consider

$$\Phi1(X) = \{\exists x.f(x) \neq x, \exists x.f(x) = x\}$$
$$\Phi2(X) = \Phi1(X) \cup \{f(x) = x\}.$$

$\mathrm{Cl}(\Phi1(X)) \neq \mathrm{Cl}(\Phi2(X))$, since $x = f(x) \notin \mathrm{Cl}(\Phi1(X))$. But it is easy to show that for any two Σ-algebras A and B, if $A \equiv_{\Phi1(X)} B$ then $A \equiv_{\Phi2(X)} B$ (the implication holds in the opposite direction by Fact 2).

4. Proofs in Structured Specifications

An important issue connected with specifications is theorem proving. We would like to be able to prove theorems about a specification, that is, that certain sentences of the underlying logic hold in every model of a specification. As suggested by Guttag and Horning [33], by proving that selected theorems hold we can understand specifications and gain confidence that they express what we want. Moreover, in order to do any kind of formal program development or verification (or even specification building, if parameterised specifications with requirements are to be used [51]) a theorem-proving capability is necessary.

In the context of structured specifications, we have to cope with two separate problems. First is how to prove theorems in theories of the underlying logic. Note that this task may be eased by the fact that our theories have structure, as this allows us to naturally disregard information which is probably irrelevant to what we are trying to prove. The other problem is dealing with the structure itself. What we need are inference rules for every specification-building operation which allow us to derive theorems about a combined specification from theorems about the components from which it was built. Note that the latter problem is not automatically reducible to the former because not all specifications are equivalent to (have the same class of models as) theories of the underlying logic [44], let alone theories with finite presentations as required for use by a theorem prover.

For simple specification-building operations appropriate inference rules are given in [43], for example,

$$\textit{thm} \text{ in } SP \;\Rightarrow\; \textit{thm} \text{ in } SP + SP',$$

where "*thm* in SP" means $\textit{thm} \in \mathrm{Th}(\mathrm{Mod}[SP])$, that is, that the sentence *thm* holds in all models of the specification SP; the specification-building operation $+$ combines the requirements on models imposed by its arguments (see [47] for details).

The **abstract** specification-building operation defined in Sections 2 and 3 is more difficult to handle. One problem is that in contrast to other specification-building operations it is not monotonic, in the sense that

$$\mathit{thm} \text{ in } SP \not\Rightarrow \mathit{thm} \text{ in } \textbf{abstract } SP \textbf{ w.r.t. } \ldots.$$

However, Fact 12 and its analogue for observations with free variables (see Sect. 3) says that the following inference rule is sound.

Inference Rule. *For any set $\Phi(X)$ of open formulae with variables in X,*

$$\mathit{thm} \text{ in } SP \text{ and } \mathit{thm} \in \mathrm{Cl}(\Phi(X)) \Rightarrow \mathit{thm} \text{ in } \textbf{abstract } SP \textbf{ w.r.t. } \Phi(X). \qquad (*)$$

Moreover, for the case of ground observations (i.e., when X is the empty set), Fact 13 shows that in some standard logics (e.g., first-order logic) the above rule is in a sense complete when used together with inference rules for the underlying logic and the other specification-building operations. Note also that Facts 7–11 provide us with some subsidiary inference rules; for example, Fact 9 implies

$$\mathit{thm} \text{ in } \textbf{abstract } SP \textbf{ w.r.t. } \Phi \text{ and } \Phi \subseteq \mathrm{Cl}(\Phi') \Rightarrow \mathit{thm} \text{ in } \textbf{abstract } SP \textbf{ w.r.t. } \Phi'.$$

A consequence of the inference rule $(*)$ is that a proof of *thm* in *SP* is also a valid proof of *thm* in **abstract** *SP* **w.r.t.** $\Phi(X)$ provided that there is a "cut" across the proof tree containing only observable sentences (i.e., sentences in $\mathrm{Cl}(\Phi(X))$). In other words, every path in the proof from *thm* to a fact in *SP* must contain at least one observable sentence.

For example, consider the following specification of sets:

Set = **sorts** elem, set
  opns $\varnothing: \to \text{set}$
    add: elem, set $\to$ set
  predicates $\in$: elem, set
  axioms $\forall x{:}\text{elem}, S{:}\text{set}.\ \text{add}(x, \text{add}(x, S)) = \text{add}(x, S)$
    $\forall x,y{:}\text{elem}, S{:}\text{set}.\ \text{add}(x, \text{add}(y, S)) = \text{add}(y, \text{add}(x, S))$
    $\forall x{:}\text{elem}. \neg(x \in \varnothing)$
    $\forall x{:}\text{elem}, S{:}\text{set}.\ x \in \text{add}(x, S)$
    $\forall x,y{:}\text{elem}, S{:}\text{set}.\ x \neq y \Rightarrow (x \in \text{add}(y, S) \Leftrightarrow x \in S)$

(for this example we use first-order predicate logic with equality). We abstract from Set with respect to an appropriate set of observable formulae:

SetAbs = **abstract** Set **w.r.t.** $\{x \in t \mid t \text{ is of the form add}(x_1, \ldots, \text{add}(x_n, \varnothing)\ldots) \text{ for } n \geq 0\}$.

We then enrich this specification by a function *choose* which selects some arbitrary element from any non-empty set:

SetChoose = **enrich** SetAbs **by**
opns choose: set $\to$ elem
axioms $\forall S$:set. $S \neq \varnothing \Rightarrow \text{choose}(S) \in S$

(the notation **enrich** *SP* **by** ... **axioms** *E* is an abbreviation for $SP + \langle \text{sig}(SP) \cup ..., E \rangle$).

Suppose we want to prove the following simple theorem in SetChoose:

$$\forall x, y\text{:elem. choose}(\text{add}(x, \text{add}(y, \varnothing))) \in \text{add}(y, \text{add}(x, \varnothing)).$$

A naive proof which ignores the use of **abstract** in SetAbs (i.e., imagining that SetChoose is built directly from Set instead of via SetAbs) might go as in Fig. 1.

To transform this into a valid proof (taking **abstract** into account) we have to transform the indicated subproofs P1 and P2 from proofs in Set to proofs in SetAbs. According to the discussion above, P1 is a proof in SetAbs since the underlined sentences are observable. This is not the case with P2. In fact, the con-

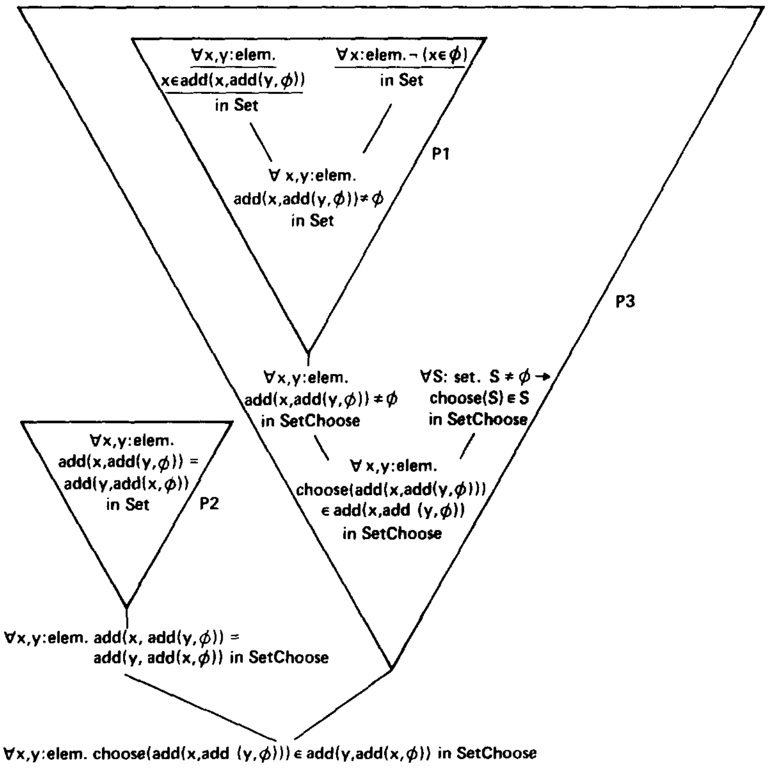
∀x,y:elem. x∈add(x,add(y,ϕ)) in Set
∀x:elem. ¬(x∈ϕ) in Set
∀ x,y:elem. add(x,add(y,ϕ))≠ϕ in Set
P1
P3
∀x,y:elem. add(x,add(y,ϕ)) ≠ ϕ in SetChoose
∀S: set. S ≠ ϕ → choose(S) ∈ S in SetChoose
∀ x,y:elem. choose(add(x,add(y,ϕ))) ∈ add(x,add (y,ϕ)) in SetChoose
∀x,y:elem. add(x,add(y,ϕ)) = add(y,add(x,ϕ)) in Set
P2
∀x,y:elem. add(x, add(y,ϕ)) = add(y, add(x,ϕ)) in SetChoose
∀x,y:elem. choose(add(x,add (y,ϕ))) ∈ add(y,add(x,ϕ)) in SetChoose

FIGURE 1

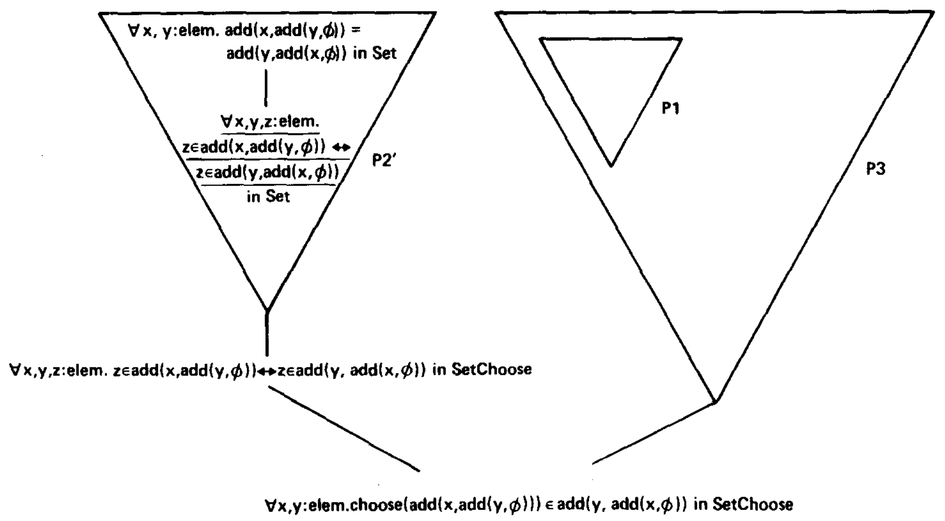
∀x, y:elem. add(x,add(y,∅)) = add(y,add(x,∅)) in Set
∀x,y,z:elem. z∈add(x,add(y,∅)) ↔ z∈add(y,add(x,∅)) in Set
P2′
P1
P3
∀x,y,z:elem. z∈add(x,add(y,∅))↔z∈add(y, add(x,∅)) in SetChoose
∀x,y:elem.choose(add(x,add(y,∅)))∈add(y, add(x,∅)) in SetChoose

FIGURE 2

clusion of P2 is not a theorem of SetAbs (or SetChoose) at all. Thus we must modify our proof as shown in Fig. 2 (note that the subproof P3 which includes P1 remains unaltered). P2′ is a valid proof in SetAbs, so the complete proof is valid in SetChoose.

5. Behavioural Equivalence—An Example

In Sections 2 and 3 we defined a very general and powerful notion of observational equivalence. In this section we look at a very important special case and we consider an example of its use. Namely, we restrict observations to equations between terms from some specified set; this gives an equivalence corresponding to the one used in the ASL specification language [47]. A proper choice of the set of terms gives *behavioural equivalence* as informally discussed in the Introduction.

Suppose that Σ is a signature and IN and OUT are subsets of the sorts of Σ. Now, consider all computations which take input from sorts IN and give output in sorts OUT; this set of computations corresponds to the set of Σ-terms of sorts OUT with variables of sorts IN. Consider the set $EQ_{OUT}(X_{IN})$ of equations between terms of the same sort in OUT having variables X_{IN} of sorts in IN. Two algebras are observationally equivalent with respect to $EQ_{OUT}(X_{IN})$ if they are behaviourally equivalent, that is, if they have matching input/output relations. Note that this covers the notions of behavioural equivalence with respect to a single set OBS of *observable* sorts which appear in the literature. For example, in [28, 42] we have $IN = \text{sorts}(\Sigma)$, $OUT = OBS$; in [29, 47, 48] $IN = OUT = OBS$; and in [5, 20, 34]

$IN = \varnothing$ and $OUT = OBS$. To denote the corresponding special case of **abstract** we use

$$\textbf{behaviour } SP \textbf{ with in } IN \textbf{ out } OUT =_{\text{def}} \textbf{abstract } SP \textbf{ w.r.t. } EQ_{OUT}(X_{IN}).$$

This corresponds to *behavioural abstraction* as defined in ASL [47].

As an example we are going to consider a simple language of expressions for arithmetical computation over the integers. This may be imagined as a small piece of a real programming language. We believe that the approach used below may be applied to other programming language constructs as well, leading toward the possible formal development of a compiler. Handling programming language features like recursion requires that we switch to the framework of partial algebras [7] or continuous algebras [32]; this is not a problem since as will be discussed in Section 6 our definitions and results extend smoothly to these cases.

We assume that we are given some standard specifications of the integers (Int) with the usual arithmetic operations and of identifiers (Ident). For this example, Ident need only contain a single sort called *ident*. The (abstract) syntax of expressions is given by the following specification (we use the notation of the Clear specification language [8][4]):

```
Expr = enrich Int + Ident by
           data sorts expr
                opns const: int → expr
                     var: ident → expr
                     plus, times: expr, expr → expr
                     cond: expr, expr, expr → expr
```

The use of **data** above means that any model of Expr is a free extension of a model of Int + Ident. That is, the sort *expr* contains (up to isomorphism) expressions built up using the newly introduced operations. We could achieve the same effect using a *hierarchy constraint* [4] (cf. [17, 46]) together with the appropriate inequations.

To describe the semantics of expressions we need the additional concept of an environment from which the values of variables may be retrieved. This is described by the following (loose) specification:

```
Env = enrich Int + Ident by
          sorts env
          opns lookup: env, ident → int
```

For the purpose of our example, no more that the existence of an operation *lookup* is required. Other parts of the language may need a more elaborate specification of environments, including, e.g., an operation to modify environments and axioms to

[4] But for the semantics of **derive**, see [47].

relate this operation to *lookup*. We also simplify our view of environments by assuming that *lookup* never produces errors. We could incorporate standard approaches to specifications with errors, see Section 6.

```
Eval = **enrich** Expr + Env **by**
        **opns**    eval: expr, env → int
        **axioms**  ∀n:int, ρ:env. eval(const(n), ρ) = n
                 ∀x:ident, ρ:env. eval(var(x), ρ) = lookup(ρ, x)
                 ∀e,e′:expr, ρ:env. eval(plus(e, e′), ρ) = eval(e, ρ) + eval(e′, ρ)
                 ∀e,e′:expr, ρ:env. eval(times(e, e′), ρ) = eval(e, ρ) × eval(e′, ρ)
                 ∀e,e′,e″:expr, ρ:env. eval(cond(e, e′, e″), ρ)
                                         = eval(e″, ρ) if eval(e, ρ) = 0
                                         = eval(e′, ρ) otherwise
```

(We use an obvious notation to simplify the syntax of conditional axioms.)

The models of Eval are just the models of Expr with the expected semantics provided by the operation *eval*. The *cond* construct has the semantics of **if** _ **then** _ **else** _, where 0 (as the value of the first argument) is interpreted as false and any other value is interpreted as true. Note that the models of Eval are pretty well determined; in fact, they are determined up to isomorphism given models of Ident and Env. Now imagine that we want to build a compiler which performs some source-level optimisation; for example, recognising that *times*(*const*(0), *e*) is just *const*(0). Such optimisations are not permitted by the specification above.

Two solutions to this dilemma are offered in the literature. First, [52] and [34] advocate the use of *final* models; if we adopt this approach (modifying the above specification appropriately) then every (final) model of Eval would satisfy $e = e'$ iff it satisfies $\forall \rho{:}env.\ eval(e, \rho) = eval(e', \rho)$, for all expressions e and e'. But this disallows non-optimal implementations, since it requires that all possible optimisations are performed. Much worse, the specified models are actually not attainable since the optimisation required is not computable (this follows from a result in [11]).

Second, as advocated in, e.g., [13, 15] the notion of *implementation* of one specification by another should take care of this problem. Algebras with some optimisations are not models of the specification above but models of a specification which implements it. Unfortunately, the formal notions of implementation which have been suggested are rather complicated, and especially so in the context of loose and parameterised specifications. (Note that the specification above may be viewed as parameterised by Ident.)

We adopt neither of these solutions. Instead, we argue that the specification Eval as given above is not really what we intend. When we specify a program, what we are really interested in is its behaviour, that is, the answers which we obtain when the program is applied to the various possible inputs. The specification Eval says more than that; it dictates the structure of internal data. We can obtain the class of models having the behaviour which Eval specifies (rather concretely) by applying the **behaviour** operation for the appropriate choice of input and output sorts:

Eval-we-really-want = **behaviour** Eval **with in** {int,ident,env} **out** {int}

The inference rule for **abstract** given in Section 4 may be applied here to show, e.g., that

$$\forall e,e'\text{:expr}, \rho\text{:env. eval(plus}(e, e'), \rho) = \text{eval(plus}(e', e), \rho)$$

is a theorem in Eval-we-really-want, since it is a theorem of Eval and is in the closure of the set of observations we are using here.[5]

The ability to specify classes of algebras up to behavioural equivalence (as in Eval-we-really-want) allows us to greatly simplify our formal view of what an implementation is. Proceeding from a specification to a program means making a series of design decisions, each of which amounts to a restriction on the class of models. Such design decisions are choice of data structures, choice of algorithms, and choice between alternatives which the specification leaves open. Thus, a simple but natural notion of implementation is as follows.

DEFINITION. A specification SP is *implemented by* a specification SP', written $SP \rightsquigarrow SP'$, if $\text{Mod}[SP'] \subseteq \text{Mod}[SP]$.

It is easy to see that the above implementation relation is transitive ($SP \rightsquigarrow SP'$ and $SP' \rightsquigarrow SP''$ implies $SP \rightsquigarrow SP''$), i.e., that it can be composed *vertically* (see [25]). This means that a specification can be refined gradually. Furthermore, this implementation relation can be composed *horizontally* [25] as well[6] [47] ($SP1 \rightsquigarrow SP1'$ and $SP2 \rightsquigarrow SP2'$ implies $SP1 + SP2 \rightsquigarrow SP1' + SP2'$ and similarly for the other specification-building operations). This means that specifications can be refined in a modular fashion. This is in contrast to the more complicated notions of implementation mentioned earlier for which these properties do not hold in general.

The following specification is an implementation of Eval-we-really-want:

```
Eval' =
  let Ev0 = enrich Eval by
            opns    optplus, opttimes: expr, expr → expr
                    optcond: expr, expr, expr → expr
            axioms  ∀e,e':expr. optplus(e, e')
                          = e'                         if e = const(0)
                          = e                          if e' = const(0)
                          = opttimes(const(2), e)      if e = e'
                          = plus(e, e')                otherwise
```

[5] For technical reasons (see [27]) the validity of this comment requires that there be constants of sort *ident*.

[6] This is the case provided that all specification-building operations are monotonic (with respect to model classes), which is the case for the specification-building operations defined in, e.g., Clear [8], LOOK [16], ASL [47], and for **abstract** and **behaviour** as defined above.

```
        ∀e,e':expr. opttimes(e, e')
                       = const(0)          if e = const(0)
                                           or e' = const(0)
                       = e'                if e = const(1)
                       = e                 if e' = const(1)
                       = times(e, e')      otherwise
        ∀e,e',e'':expr. optcond(e, e', e'')
                       = e'                if e = const(n) and n ≠ 0
                       = e''               if e = const(0)
                       = e'                if e' = e''
                       = cond(e, e', e'')  otherwise
in derive signature Eval
   from Ev0
   by const is const
      var is var
      plus is optplus
      times is opttimes
      cond is optcond
      eval is eval
```

Eval′ specifies the syntax and semantics of our expression language, requiring that certain source-level optimisations (constant folding) be carried out.

In order to prove that Eval′ implements Eval-we-really-want we have to show:

CLAIM. Mod[Eval-we-really-want] $\supseteq$ Mod[Eval′].

To prove this we have to show that any model of Eval′ is behaviourally equivalent to a model of Eval (with respect to input sorts $\{int, ident, env\}$ and output sort $\{int\}$).

Sketch of Proof. Let A' be a model of Eval′. By the definition of **derive** [47] $A' = A0|_\sigma$ for some model $A0$ of Ev0, where σ is the signature morphism described in the **derive**. Let $A = A0|_\iota$, where ι:Sig[Eval] $\rightarrow$ Sig[Ev0] is the signature inclusion. By definition of **enrich** [8] A is a model of Eval. We claim that A and A' are behaviourally equivalent with respect to input sorts $\{int, ident, env\}$ and output sort $\{int\}$. Note that by the construction of A, the parts of A and A' corresponding to Int, Ident, and Env are identical. Thus, it is enough to show that for any valuation v of variables of sorts $\{int, ident, env\}$ into the corresponding carriers of A (which are the same as those of A') and for any terms, t, t' of sort int having variables of sorts $\{int, ident, env\}$, $A \models_v t = t'$ iff $A' \models_v t = t'$. This may be reduced to proving that for any term t as above, its values in A and A' under v are the same. This is obvious if t is a Sig[Int]-term; the only other case is for t of the form $eval(t_{expr}, \rho)$, where ρ is a variable and t_{expr} is a Sig[Eval]-term of sort $expr$ (with variables of sorts $\{int, ident\}$). The value of this term in A is the same as in $A0$, and its value in A' is the same as the value of $eval(\sigma(t_{expr}), \rho)$ in $A0$. That

$A0 \models_v eval(t_{expr}, \rho) = eval(\sigma(t_{expr}), \rho)$ may be proved by an easy induction on t_{expr}. ∎

Note that this proof technique is quite general; it basically relies only on the fact that the specifications under consideration are *persistent enrichments* (see, e.g., [19]) of the specifications of their observable parts. The mechanical proof of a theorem similar to the final step of the above proof is described in [6].

A different way of proving that two algebras are behaviourally equivalent is suggested in [48]; in this approach, a relation (called a *correspondence*) between the corresponding carriers is set up explicitly and proved to satisfy a kind of homomorphism property.

A more detailed discussion of the application of the ideas presented in this section to the development of programs is given in [45].

6. Observational Equivalence in an Arbitrary Institution

In the previous sections we have been rather vague about what we mean by a "formula." We have mentioned formulae of equational logic, first-order logic, and infinitary logic. Moreover, although we have been using the standard notion of many-sorted algebra as in [31], this was mostly in order to take advantage of the reader's intuition; in fact, we made use of very few formal properties of algebras. This means that in place of the standard notion we could have used, for example, partial algebras [7] or continuous algebras [32]. We could even change both the notions of signature and of algebra to deal with errors [22, 23] or coercions [21, 24].

The notion of an *institution* [26] provides a tool for dealing with any of these different notions of a logical system for writing specifications. An institution comprises definitions of signature, model (algebra), sentence, and a satisfaction relation satisfying a few minimal consistency conditions. (For a similar but more logic-oriented approach see [3].) By basing our definitions (of observational equivalence, etc.) on an arbitrary institution we can avoid choosing particular definitions of these underlying notions and do everything at an adequately general level.

Definition. An *institution INS* consists of:

— A category $\mathbf{Sign}_{INS}$ (of signatures)

— A functor $\mathbf{Sen}_{INS}: \mathbf{Sign}_{INS} \to \mathbf{Set}$ (where $\mathbf{Set}$ is the category of all sets; $\mathbf{Sen}_{INS}$ gives for any signature Σ the set of Σ-sentences and for any signature morphism $\sigma: \Sigma \to \Sigma'$ the function $\mathbf{Sen}_{INS}(\sigma): \mathbf{Sen}_{INS}(\Sigma) \to \mathbf{Sen}_{INS}(\Sigma')$ translating Σ-sentences to Σ'-sentences)

— A functor $\mathbf{Mod}_{INS}: \mathbf{Sign}_{INS} \to \mathbf{Cat}^{\mathrm{op}}$ (where $\mathbf{Cat}$ is the category of all categories; $\mathbf{Mod}_{INS}$ gives for any signature Σ the category of Σ-models and for any signature morphism $\sigma:\Sigma\to\Sigma'$ the σ-reduct functor $\mathbf{Mod}_{INS}(\sigma):\mathbf{Mod}_{INS}(\Sigma')\to \mathbf{Mod}_{INS}(\Sigma)$ translating Σ'-models to Σ-models)

— A satisfaction relation $\models_{\Sigma,INS}\subseteq|\mathbf{Mod}_{INS}(\Sigma)|\times\mathbf{Sen}_{INS}(\Sigma)$ for each signature Σ

such that for any signature morphism $\sigma:\Sigma\to\Sigma'$ the translations $\mathbf{Mod}_{INS}(\sigma)$ of models and $\mathbf{Sen}_{INS}(\sigma)$ of sentences preserve the satisfaction relation, i.e., for any $\varphi\in\mathbf{Sen}_{INS}(\Sigma)$ and $M'\in|\mathbf{Mod}_{INS}(\Sigma')|$,

SATISFACTION CONDITION. $M'\models_{\Sigma',INS}\mathbf{Sen}_{INS}(\sigma)(\varphi)$ iff $\mathbf{Mod}_{INS}(\sigma)(M')\models_{\Sigma,INS}\varphi$

To be useful as the underlying institution of a specification language, an institution must provide some tools for "putting things together." Thus, in this paper we additionally require that the category **Sign** has pushouts and initial objects (i.e., is finitely cocomplete) and, moreover, that **Mod** preserves pushouts and initial objects (and hence finite colimits), i.e., that **Mod** translates pushouts and initial objects in **Sign** to pullbacks and terminal objects (respectively) in **Cat**. For a brief discussion of these requirements see [44]. For notational convenience we omit subscripts like *INS* and Σ whenever possible, and for any signature morphism $\sigma:\Sigma\to\Sigma'$ we denote $\mathbf{Sen}(\sigma)$ simply by σ and $\mathbf{Mod}(\sigma)$ by $\_|_{\sigma}$.

In [8] the semantics of the Clear specification language was defined in terms of an arbitrary institution (called there a "language"). More recently, in [44] a number of more basic but powerful specification-building operations were defined in the framework of an arbitrary institution.

We encounter no problems at all in generalising the contents of Section 2 (on ground observations) to an arbitrary institution. In fact, the definitions of observational equivalence and of the **abstract** operation with respect to a set of ground observations may be taken literally as they were formulated there. Moreover, Facts 1–12 still hold. (Facts 13 and 14 hold for institutions with some simple closure properties. Fact 15 may be generalised if we equip institutions with some notion of reachability along the lines of [50].)

In order to deal with the general case of observations containing free variables we have first to provide a notion of an open formula and a valuation of free variables in the framework of an arbitrary institution. Although sentences as they are used in the definition of an institution above are always closed, this may be done.

Namely, note that in the standard algebraic case, an open Σ-formula with variables X may be viewed as a (closed) sentence over the signature $\Sigma(X)$ resulting from Σ by adding the elements of X as constants. Then a valuation of variables X into a Σ-algebra A may be viewed as an expansion of A to a $\Sigma(X)$-algebra, which additionally contains interpretations of the constants X. This formulation can be extended to the framework of an arbitrary institution as follows (see [44] for a more detailed exposition).

Let Σ be a signature. Any pair $\langle\varphi,\theta\rangle$, where $\theta:\Sigma\to\Sigma'$ is a signature morphism and $\varphi \in \mathbf{Sen}(\Sigma')$, may be viewed as an open Σ-formula with variables "$\Sigma' - \theta(\Sigma)$." (Note the quotation marks—since $\Sigma' - \theta(\Sigma)$ makes no sense in an arbitrary institution, it is only meaningful as an aid to our intuition.) If M is a Σ-model, $M \in |\mathbf{Mod}(\Sigma)|$, then a valuation of variables "$\Sigma' - \theta(\Sigma)$" into M is a Σ'-model $M' \in |\mathbf{Mod}(\Sigma')|$ which is a θ-expansion of M, i.e., $M'|_\theta = M$.

Given an open Σ-formula $\langle \varphi, \theta \rangle$ we can define its universal closure, written $\forall \langle \varphi, \theta \rangle$, as a new Σ-sentence. A Σ-model satisfies $\forall \langle \varphi, \theta \rangle$ if each of its θ-expansions satisfies φ, i.e., for any $M \in |\mathbf{Mod}(\Sigma)|$,

$$M \models \forall \langle \varphi, \theta \rangle \quad \text{iff} \quad \text{for any } M' \in |\mathbf{Mod}(\Sigma')| \text{ such that } M'|_\theta = M,\ M' \models \varphi.$$

Obviously, other quantifiers (there exists, there exist infinitely many, there exists a unique, for almost all ...) may be introduced in the same manner as we have just introduced universal quantifiers. Moreover, one may similarly introduce logical connectives such as negation and conjunction (of formulae having the same "free variables") with the standard logical interpretation (cf. [3]). Thus, our definition of the Cl operator on sets of open formulae (see Sect. 3) works in any institution. Note also that these definitions give more generality than was implied by the discussion of the standard algebraic case above, since variables denoting operations or even sorts are permitted. Such variables could be forbidden if desired by restricting θ appropriately.

The above definitions allow us to generalise the contents of Section 3.

DEFINITION. For any signature Σ, signature morphism $\theta: \Sigma \to \Sigma'$, set $\Phi \subseteq \mathbf{Sen}(\Sigma')$ of open Σ-formulae, and Σ-models A, B, A is *observationally reducible to* B *w.r.t.* Φ *via* θ, written $A \leqslant^\theta_\Phi B$, if for every (valuation) $A' \in |\mathbf{Mod}(\Sigma')|$ with $A'|_\theta = A$ there exists (a valuation) $B' \in |\mathbf{Mod}(\Sigma')|$ with $B'|_\theta = B$, such that for all $\varphi \in \Phi$, $A' \models \varphi$ iff $B' \models \varphi$.

As in the standard case, this is a preorder and so we define observational equivalence as before.

DEFINITION. For any signature Σ, signature morphism $\theta: \Sigma \to \Sigma'$, set $\Phi \subseteq \mathbf{Sen}(\Sigma')$ of open Σ-formulae, and Σ-models A, B, A and B are *observationally equivalent w.r.t.* Φ *via* θ, written $A \equiv^\theta_\Phi B$, if $A \leqslant^\theta_\Phi B$ and $B \leqslant^\theta_\Phi A$.

Again, it is easy to verify that the facts in Section 3 concerning observational equivalence are still valid. Note, however, that if the sets of observations under consideration use different sets of free variables, as in the reformulation of Fact 3 given in Section 3, then the disjoint union of sets of variables must be replaced here by the colimit of the signature morphisms which introduce the free variables. The proofs then use the satisfaction condition and our requirement that **Mod** preserves colimits. An example of a proof using these ideas is the proof of the satisfaction condition for universally quantified formulae in the extended version of [44]. (Note that in order to deal with arbitrary (infinite) families of sets of observations for the generalisation of Fact 3, we have to require that the category **Sign** be cocomplete rather than only finitely cocomplete and that **Mod** preserves colimits rather than only finite colimits.)

The definition of the specification-building operation **abstract** is much as it was before:

DEFINITION. For any specification SP, signature morphism $\theta{:}\mathrm{Sig}[SP] \rightarrow \Sigma'$, and set $\Phi \subseteq \mathbf{Sen}(\Sigma')$ of open Sig[SP]-formulae,

$$\mathrm{Sig}[\textbf{abstract } SP \textbf{ w.r.t. } \Phi \textbf{ via } \theta] = \mathrm{Sig}[SP]$$

$$\mathrm{Mod}[\textbf{abstract } SP \textbf{ w.r.t. } \Phi \textbf{ via } \theta] = \{A \mid A \equiv_\Phi^\theta B \text{ for some } B \in \mathrm{Mod}[SP]\}.$$

As might be suspected by now, the relevant facts concerning **abstract** (Facts 7–12) still hold under this definition. Moreover, the inference rule given in Section 4 is sound here as well. The remarks concerning the notion of implementation in Section 5 carry over without change.

7. CONCLUSION

By exploring the properties of a primitive but powerful and general notion such as observational equivalence and then deriving the more directly useful concept of behavioural equivalence as a special case, we are following in the footsteps of earlier work on kernel specification-building operations [44, 47, 53, 54]. Our ultimate interest is not in the primitive notions themselves but rather in the useful higher level constructs which can be expressed in their terms. By carefully investigating the primitives we hope to gain insights which can be applied to the derived constructs. An example which justifies this approach is the **junk** specification-building operation of ASL [47] which is another useful special case of **abstract**:

$$\textbf{junk } SP \textbf{ on } S =_{\mathrm{def}} \textbf{abstract } SP \textbf{ w.r.t. } EQ_{\mathrm{sorts}(SP)}(X_{\mathrm{sorts}(SP)-S}).$$

(See Sect. 5 for the meaning of the EQ notation.) This gives those algebras which are the same as models of SP except that they may contain arbitrary junk in sorts S; see [47] for examples of its use. Since we have studied observational equivalence and **abstract** rather than the special case of behavioural abstraction and **behaviour**, everything we have done applies to **junk** as well.

We have not yet investigated thoroughly the interaction between **behaviour** and other specification-building operations, although a start in this direction is given by Facts 5 and 6 which give rise to the following identities:

$$\textbf{translate (abstract } SP \textbf{ w.r.t. } \Phi\textbf{) by } \sigma = \textbf{abstract (translate } SP \textbf{ by } \sigma\textbf{) w.r.t. } \sigma(\Phi)$$

$$\textbf{derive from (abstract } SP \textbf{ w.r.t. } \sigma(\Phi)\textbf{) by } \sigma = \textbf{abstract (derive from } SP \textbf{ by } \sigma\textbf{) w.r.t. } \Phi$$

$$(\textbf{abstract } SP \textbf{ w.r.t. } \Phi) + (\textbf{abstract } SP' \textbf{ w.r.t. } \Phi') = \textbf{abstract } (SP + SP') \textbf{ w.r.t. } \Phi \cup \Phi'$$

$$\text{if } \mathrm{Sig}[SP] \text{ and } \mathrm{Sig}[SP'] \text{ are disjoint signatures}$$

(See, e.g., [44] for the semantics of **translate**, **derive**, and +.) An issue we have not discussed is the connection between behavioural equivalence/abstraction and parameterisation of specifications. An approach to the problem of specifying software modules which integrates parameterisation and implementation is given in [14].

The material in this paper could provide the basis for high-level specification languages such as one in which every specification is surrounded by an implicit (and invisible) application of **behaviour** with respect to input and output sorts appropriate to the context. This follows the argument in the Introduction that a specification is only worthly to be called "abstract" if the class of its models is closed under behavioural equivalence. Such a language, obtained by extending the modularisation facilities of the programming language Standard ML, is presented in [45].

Acknowledgments

Our thanks to Rod Burstall for his support and encouragement and to Martin Wirsing and Rocco de Nicola for helpful comments on an earlier version. Support was provided by the Science and Engineering Research Council.

References

1. M. A. Arbib and E. G. Manes, "Arrows, Structures and Functors: The Categorical Imperative," Academic Press, New York/London, 1975.
2. J. Barwise, Back and forth through infinitary logic, *in* "Studies in Model Theory" (M. D. Morley, Ed.), Studies in Mathematics Vol. 8, pp. 5–34, Amer. Math. Assn., Washington, DC, 1973.
3. J. Barwise, Axioms for abstract model theory, *Ann. Math. Logic* **7** (1974), 221–265.
4. F. L. Bauer *et al.* (the CIP Language Group), "Report on a Wide Spectrum Language for Program Specification and Development," Report TUM-I8104, Technical Univ. Munich, 1981.
5. J. A. Bergstra and J. J. Meyer, I/O computable data structures, *SIGPLAN Notices* **16**, No. 4 (1981), 27–32.
6. R. S. Boyer and J. S. Moore, "A Computational Logic," Academic Press, New York/London, 1979.
7. M. Broy and M. Wirsing, Partial abstract types, *Acta Inform.* **18** (1982), 47–64.
8. R. M. Burstall and J. A. Goguen, The semantics of Clear, a specification language, *in* "Proceedings of Advanced Course on Abstract Software Specifications, Copenhagen," Lect. Notes in Comput. Sci. Vol. 86, pp. 292–332, Springer, New York/Berlin, 1980.
9. R. M. Burstall and J. A. Goguen, Algebras, theories and freeness: An introduction for computer scientists, *in* "Proceedings, 1981 Marktoberdorf NATO Summer School," Reidel, Dordrecht, 1982.
10. R. M. Burstall, D. B. MacQueen, and D. T. Sannella, HOPE: An experimental applicative language, *in* "Proceedings, 1980 LISP Conference, Stanford, California," pp. 136–143, 1980.
11. A. Church, An unsolvable problem of elementary number theory, *Amer. J. Math.* **58** (1936), 345–363.

12. R. DE NICOLA AND M. C. B. HENNESSY, Testing equivalences for processes, *Theoret. Comput. Sci.* **34** (1984), 83–133.
13. H.-D. EHRICH, "On the Theory of Specification, Implementation, and Parametrization of Abstract Data Types," Report 82, Abteilung Informatik, University of Dortmund, 1979; *J. Assoc. Comput. Mach.* **29**, No. 1 (1982), 206–227.
14. H. EHRIG, "An Algebraic Specification Concept for Modules" (draft version), Report 84–04, Institut für Software und Theoretische Informatik, Technical Univ. Berlin, 1984.
15. H. EHRIG, H.-J. KREOWSKI, B. MAHR, AND P. PADAWITZ, Algebraic implementation of abstract data types, *Theoret. Comput. Sci.* **20** (1982), 209–263.
16. H. EHRIG, J. W. THATCHER, P. LUCAS, AND S. N. ZILLES, Denotational and initial algebra semantics of the algebraic specification language LOOK, Draft report, IBM Research, 1982.
17. H. EHRIG, E. G. WAGNER, AND J. W. THATCHER, Algebraic specifications with generating constraints, *in* "Proceedings, 10th ICALP, Barcelona," Lect. Notes in Comput. Sci. Vol. 154, pp. 188–202, Springer, New York/Berlin, 1983.
18. H. B. ENDERTON, "A Mathematical Introduction to Logic," Academic Press, London/New York, 1972.
19. H. GANZINGER, Parameterized specifications: parameter passing and implementation, *TOPLAS* **3** (1983), 318–354.
20. V. GIARRATANA, F. GIMONA, AND U. MONTANARI, Observability concepts in abstract data type specification, *in* "Proceedings, 5th MFCS, Gdansk," Lect. Notes in Comput. Sci. Vol. 45, Springer, New York/Berlin, 1976.
21. M. GOGOLLA, "Algebraic Specifications with Partially Ordered Sorts and Declarations," Report 169, Abteilung Informatik, University of Dortmund, 1983.
22. M. GOGOLLA, K. DROSTEN, U. LIPECK, AND H.-D. EHRICH, Algebraic and operational semantics of specifications allowing exceptions and errors, *Theoret. Comput. Sci.* **34** (1984), 289–313.
23. J. A. GOGUEN, Abstract errors for abstract data types, *in* "Proceedings, IFIP Working Conf. on the Formal Description of Programming Concepts, New Brunswick, New Jersey," 1977.
24. J. A. GOGUEN, "Order Sorted Algebras: Exceptions and Error Sorts, Coercions and Overloaded Operators," Semantics and Theory of Computation Report No. 14, Dept. of Computer Science, UCLA, 1978.
25. J. A. GOGUEN AND R. M. BURSTALL "CAT, a System for the Structured Elaboration of Correct Programs from Structured Specifications," Technical report CSL-118, Computer Science Laboratory, SRI International, 1980.
26. J. A. GOGUEN AND R. M. BURSTALL, Introducing institutions, *in* "Proceedings Logics of Programming Workshop, Carnegie–Mellon Univ.," Lect. Notes in Comput. Sci. Vol. 164, pp. 221–256, Springer, New York/Berlin, 1984.
27. J. A. GOGUEN AND J. MESEGUER, Completeness of many-sorted equational logic, *SIGPLAN Notices* **16**, No. 7 (1981), 24–32; extended version *Houston J. Math.* **11** (1985), 307–334.
28. J. A. GOGUEN AND J. MESEGUER, Universal realization, persistent interconnection and implementation of abstract modules, *in* "Proceedings, 9th ICALP, Aarhus, Denmark," Lect. Notes in Comput. Sci. Vol. 140, pp. 265–281, Springer, New York/Berlin, 1982.
29. J. A. GOGUEN AND J. MESEGUER, "An Initiality Primer," Draft report, SRI International, 1983.
30. J. A. GOGUEN AND J. TARDO, An introduction to OBJ: A language for writing and testing software specifications, *in* "Specification of Reliable Software," pp. 170–189, IEEE, New York, 1979.
31. J. A. GOGUEN, J. W. THATCHER, AND E. G. WAGNER, "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types," IBM Research Report RC 6487, 1976; *in* "Current Trends in Programming Methodology, Vol. 4: Data Structuring" (R. T. Yeh, Ed.), pp. 80–149, Prentice–Hall, Englewood Cliffs, NJ, 1978.
32. J. A. GOGUEN, J. W. THATCHER, E. G. WAGNER, AND J. B. WRIGHT, Initial algebra semantics and continuous algebras, *J. Assoc. Comput. Mach.* **24**, No. 1 (1977), 68–95.
33. J. V. GUTTAG AND J. J. HORNING, Formal specification as a design tool, *in* "Proceedings, 7th ACM Symposium on Principles of Programming Languages, Las Vegas," pp. 251–261, 1980.

34. S. Kamin, Final data types and their specification, *TOPLAS* 5, No. 1 (1983), 97–121.
35. C. R. Karp, "Languages with Expressions of Infinite Length," North–Holland, Amsterdam, 1964.
36. B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler, and A. Snyder, "CLU Reference Manual," Lect. Notes in Comput. Sci. Vol. 144, Springer, New York/Berlin, 1981.
37. B. H. Liskov and V. Berzins, "An Appraisal of Program Specifications," Computation Structures Group Memo 141–1, Laboratory for Computer Science, MIT, 1977.
38. S. MacLane, "Categories for the Working Mathematician," Springer, New York/Berlin, 1971.
39. T. S. E. Maibaum, M. R. Sadler, and P. A. S. Veloso, "Logical Implementation," Technical report, Department of Computing, Imperial College, 1983.
40. P. D. Mosses, Abstract semantic algebras! *in* "Proceedings, IFIP TC2 Working Conf. on Formal Description of Programming Concepts II, Garmisch–Partenkirchen," North–Holland, Amsterdam, 1983.
41. P. Pepper, On the correctness of type transformations, talk at "2nd Workshop on Theory and Applications of Abstract Data Types, Passau," 1983.
42. H. Reichel, Behavioural Equivalence—a unifying concept for initial and final specification methods, *in* "Proceedings 3rd Hungarian Computer Science Conf., Budapest," pp. 27–39, 1981.
43. D. T. Sannella and R. M. Burstall, Structured theories in LCF, *in* "Proceedings 8th Colloq. on Trees in Algebra and Programming, L'Aquila, Italy," Lect. Notes in Comput. Sci. Vol. 159, pp. 377–391, Springer, New York/Berlin, 1983.
44. D. T. Sannella and A. Tarlecki, Building specifications in an arbitrary institution, *in* "Proceedings, Intl. Symposium on Semantics of Data Types, Sophia–Antipolis," Lect. Notes in Comput. Sci. Vol. 173, pp. 337–356, Springer, New York/Berlin, 1984.
45. D. T. Sannella and A. Tarlecki, Program specification and development in Standard ML, *in* "Proceedings, 12th ACM Symp. on Principles of Programming Languages, New Orleans," pp. 67–77, 1985.
46. D. T. Sannella and M. Wirsing, "Implementation of Parameterised Specifications," Report CSR-103-82, Dept. of Computer Science, University of Edinburgh; extended abstract in "Proceedings, 9th ICALP, Aarhus, Denmark," Lect. Notes in Comput. Sci. Vol. 140, pp. 473–488, Springer, New York/Berlin, 1982.
47. D. T. Sannella and M. Wirsing, "A Kernel Language for Algebraic Specification and Implementation," Report CSR-131-83, Dept. of Computer Science, University of Edinburgh; extended abstract in "Proceedings, Intl. Conf. on Foundations of Computation Theory, Borgholm, Sweden," Lect. Notes in Comput. Sci. Vol. 158, pp. 413–427, Springer, New York/Berlin, 1983.
48. O. Schoett, "A Theory of Program Modules, Their Specification and Implementation (extended abstract)," Report CSR-155-83, Dept. of Computer Science, University of Edinburgh, 1983.
49. D. Scott, Logic with denumerably long formulas and finite strings of quantifiers, *in* "Theory of Models," pp. 329–341, North–Holland, Amsterdam, 1965.
50. A. Tarlecki, On the existence of free models in abstract algebraic institutions, *Theoret. Comput. Sci.* **37** (1985), 269–304.
51. J. W. Thatcher, E. G. Wagner, and J. B. Wright, Data type specification: Parameterization and the power of specification techniques, *in* "SIGACT 10th Annual Symp. on the Theory of Computing, San Diego, California," 1978; *in TOPLAS* **4**, No. 4 (1982), 711–732.
52. M. Wand, Final algebra semantics and data type extensions, *J. Comput. System Sci.* **19** (1979), 27–44.
53. M. Wirsing, Structured algebraic specifications, *in* "Proceedings, AFCET Symp. on Mathematics for Computer Science, Paris, pp. 93–107, 1982.
54. M. Wirsing, "Structured Algebraic Specifications: A Kernel Language," Habilitation thesis, Technical Univ. Munich, 1983.