A Complete Calculus for the Multialgebraic and Functional Semantics of Nondeterminism

Michal Walicki*
michal@ii.uib.no

Sigurd Meldal
sigurd@ii.uib.no

University of Bergen
Department of Informatics

Keywords: Nondeterminism; Multialgebras; Equational logic

Abstract: The current algebraic models for nondeterminism focus on the notion of *possibility* rather than *necessity*, and consequently equate (nondeterministic) terms that one intuitively would not consider equal. Furthermore, existing models for nondeterminism depart radically from the standard models for (equational) specifications of deterministic operators. One would prefer that a specification language for nondeterministic operators be based on an extension of the standard model concepts, preferably in such a way that the reasoning system for (possibly nondeterministic) operators becomes the standard equational one whenever restricted to the deterministic operators – the objective should be to *minimize* the departure from the standard frameworks.

In this paper we define a specification language for nondeterministic operators and multialgebraic semantics. The first *complete* reasoning system for such specifications is introduced. We also define a transformation of specifications of nondeterministic operators into derived specifications of deterministic ones, obtaining a "computational" semantics of nondeterministic specification by adopting the standard semantics of the derived specification as the semantics of the original one. This semantics turns out to be a refinement of multialgebra semantics. The calculus is shown to be sound and complete also with respect to the new semantics.

1. Introduction

The notion of *nondeterminism* arises naturally in describing concurrent systems. Various approaches to the theory and specification of such systems, for instance, CCS [Milner 1980], CSP [Hoare 1985], process algebras [Bergstra 1986], event structures [Winskel 1988], include the phenomenon of nondeterminism. But nondeterminism is also a natural concept in describing sequential programs, either as a means of indicating a "don't care" attitude as to which among a number of computational paths will actually be utilized in a particular computation (e.g., [Dijkstra 1976]) or as a means of increasing the level of abstraction [Meldal 1989, Walicki 1994a]. The present work proceeds from the theory of *algebraic specifications* [Ehrig 1985, Wirsing 1990] and generalizes it so that it can be applied to describing nondeterministic operations. Our main concern is to make such an extension non-intrusive, i.e., such that it does not change the existing framework, and in particular, reduces to the standard deterministic theory when only deterministic operations are present in the specification.

In terms of satisfiability, allowing nondeterminism is quite distinct from underspecification. The latter, though admitting distinct models, still insists on a function always returning a specific value, given a particular list of arguments. In the case of nondeterministic operators this is no longer true.[1] Also, nondeterminism turns out to be a useful abstraction tool whenever there is a

* We gratefully acknowledge the financial support received from the Norwegian Research Council.

[1] In the interest of clarity we shall let the term *function* denote deterministic operator, and the term *operation* denote nondeterministic as well as deterministic operators.

hidden state or other components of a system description which are, methodologically or technically, inaccessible at a particular level of abstraction. As we illustrate in section 6 [also Walicki 1994a], underspecification does not provide an equally powerful abstraction mechanism.

The basic concept of classical algebraic specifications is the notion of *function*. Classical specifications describe programs which, when viewed abstractly, are input-output functions – a particular input uniquely determines the output value resulting from evaluating the program.[2] This does not hold for *nondeterministic* programs. The input to such a program does *not* uniquely determine the resultant output. Since a nondeterministic program cannot be modeled as an input-output function, the *set approach* [Hesselink 1988, Kaplan 1988, Kapur 1980, Maibaum 1977, Subrahmanyam 1981] generalizes the standard algebraic notion of programs as functions, and regards nondeterministic programs as *functions in a multialgebra*. A program then denotes a function which instead of returning single values returns *sets* of values. For a particular nondeterministic operator f and argument a, the term $f(a)$ denotes the set of *all* values possibly returned by f. Thus, in the set approach, a nondeterministic specification of programs is modeled by structures describing its *possible* behaviors.

For a sort T, let "$\sqcup.\_: Set(T) \to T$" denote an operator which makes a nondeterministic choice among the elements of some set of T-elements. For instance, $\sqcup.\{0, 1\}$ may return either of the values 0 and 1, and need not return the same value at every application.

In the set approach, the semantics of the term $\sqcup.\{0, 1\}$ is a *constant* – the term $\sqcup.\{0, 1\}$ denotes the set $\{0, 1\}$ of possible results. Consequently, the equality $\sqcup.\{0, 1\}=\sqcup.\{0, 1\}$ will hold, since both sides are interpreted as the same set-valued constant. Consider now the *evaluation of a program* making use of such an equality, for instance in the expression "***if*** $\sqcup.\{0, 1\}=\sqcup.\{0, 1\}$ ***then*** 5 ***else*** 6 ***fi***". According to our intuition the result of evaluating the expression could be either 5 or 6, depending on whether the two applications of $\sqcup.\{0, 1\}$ returned the same value or not. But according to the set semantics of equality the result will always be 5, because the term $\sqcup.\{0, 1\}$ denotes one and the same *set*-valued constant – the equality "$\sqcup.\{0, 1\}=\sqcup.\{0, 1\}$" is interpreted as a test of whether the set $\{0, 1\}$ equals the set $\{0, 1\}$, which is always true [Hesselink 1988, Kaplan 1988, Kapur 1980, Maibaum 1977, Subrahmanyam 1981]. Thus the set semantic interpretation of equality implies the *possibility* of two terms returning the same result, but it does not guarantee that the same results *actually* will be produced. This leads to particular difficulties in treating properly such notions as, for example, observability [Hennessy 1980, Nipkow 1987] and implementation [Qian 1993].

Equality in our view should be a *necessary* equality which must hold in every evaluation of a program (specification). *It does not correspond to set equality, but to identity of 1-element sets*. Thus the simple formula $t{=}s$ will be considered true (in a multistructure M) iff both t and s are interpreted as one and the same set which, in addition, *has only one element*. (To indicate this modified interpretation we will use the symbol "$\doteq$" instead of the usual identity "=".)

Besides problems with constructing an adequate algebraic model the set approach originally forced one to consider even simple equational specifications as full first order theories [Subrahmanyam 1981, Kapur 1980]. This, of course, led to reasoning systems whose efficiency and

[2] This observation extends to reactive systems as well, see [Kahn 1974].

simplicity could not match that of the existing systems for equational reasoning (see, e.g., [O'Donell 1977, Dershowitz 1990] for references).

The difficulty with the set approach as sketched above is its inability to distinguish between the equivalence of *terms* and the equivalence of *application of terms*. This distinction is of no consequence in the deterministic case, because there every term necessarily identifies a single value; evaluating a term more than once will always yield the same result. On the other hand, distinct applications of a nondeterministic operator will typically yield different results.

The distinction between terms (operations) and their applications is merely a technical reflection of the distinction between *programs* and *computations*, a distinction whose relevance became obvious with the advent of parallel and nondeterministic computing. When writing specifications, we are primarily interested in these as constraints on *computations*. Their importance as constraints on the acceptable programs is only a secondary concern, arising from the fact that a program identifies a set of possible computations.

We are not interested in *programs* (they are just pieces of text), but rather in what happens when they are evaluated. A specification serves the purpose of abstractly describing legal computations. In contrast to the *algebras of possibilities* represented by the set approach we offer *algebras of actual computations*. Any particular computation is an abstract computation in the sense of algebra – input-output functions. In this way we preserve the basics of classical algebraic theory, and we believe that this view is the more relevant in actual use.

This paper presents a function oriented view which refines the notion of equality in the multialgebraic semantics. The result of a (non-)deterministic operator is considered a single (rather than set) value, namely the result of a particular *application*. Distinct occurrences of a particular term will each denote a single value, but not necessarily the same one for each. The equality $\sqcup.\{0, 1\}=\sqcup.\{0, 1\}$ no longer holds. Instead, we will restore the meaning of equality as a necessary relation which must hold in any particular evaluation of a program.

In this new semantics, nondeterminism is reflected in the existence of distinct *computations* – each realizing distinct possibilities of nondeterministic operators. Simple facts such as "if x is a result of applying $\sqcup.\{0, 1\}$ then x=0 or x=1" are not only expressible and provable in this system – they have a natural interpretation in the computational model.[3] The semantics also satisfies a secondary concern of ours: that the specification formalism not only deals with nondeterminism in a sensible and intuitive way, but that it also preserves the notions and results of the standard algebraic approaches to deterministic constructs. This is important for two reasons: (1) The formalism behaves in the expected ways within the usual domains of formal specifications. Thus the specifier need not be exposed to new (and possibly exotic) semantic concepts as long as he stays within the deterministic domain of discourse. (2) The existing tool base, created to cater for standard (equationally defined) algebras [Dershowitz 1990, O'Donell 1977] can be retained. In

[3] This may seem trivial, but was in fact hard enough to require the introduction of full first order logic in the early deduction systems [Kapur 1980, Subrahmanyam 1981]. All classes of structures proposed as semantics of nondeterminism allow models in which such disjunctive equations are *not* valid, and can be satisfied only by making additional semantic restrictions [Hußmann 1990,Hußmann 1993,Maibaum 1977,Mosses 1989a].

extending the tool base to cover nondeterminism as well, one can hope for reusing much of the existing software.

At the same time we show that replacing the notion of set equality by the element equality, the two semantics are compatible – the multialgebraic semantics can be seen as a convenient abstraction of the function oriented one. The calculus introduced in the paper is sound and complete with respect to both semantics and thus allows one to reason at the more abstract level where terms are thought of as sets of possible results.

As the main results we present:

1) A modified view of equality in the multialgebraic semantics of nondeterminism.
2) A calculus, NEQ, which is sound and complete with respect to the multialgebraic semantics, and which behaves identically with the standard equational calculus [Birkhoff 1935] in the deterministic case.
3) A new – functional – semantics for nondeterministic specifications which refines the multialgebraic one.
4) Soundness and completeness of NEQ with respect to this new semantics.
5) A transformation procedure allowing one to interpret specifications of nondeterministic operators as underspecified deterministic functions.

The structure of the paper is: in section 2 we introduce a specification language $\mathcal{L}_{\text{neq}}$ which provides for nondeterministic operators. In section 3 the class of multimodels of a specification in $\mathcal{L}_{\text{neq}}$ is defined. In section 4 we show how specifications of nondeterministic operators in $\mathcal{L}_{\text{neq}}$ can be transformed into specifications of deterministic ones by translating them into the language $\mathcal{L}_{\text{deq}}$ which is the usual language of deterministic conditional equations extended with disjunction. We then define the (loose) functional semantics of the original specification as the class of standard models of the derived $\mathcal{L}_{\text{deq}}$ specification. Subsection 4.3 compares the two semantics. Section 5 introduces a calculus NEQ, discusses some of its features and states soundness and completeness of NEQ with respect to both multialgebraic and functional semantics. Section 6 contains an example specification of access types. Section 7 summarizes our results and comments briefly on some related work.

The proofs have been omitted from the paper – they are given in full in [Walicki 1993a], except for the soundness and completeness theorems which are proved in the appendix.

2. Writing specifications of nondeterministic operators

In this section we present the specification language $\mathcal{L}_{\text{neq}}$, in which the user is expected to write the specifications. We also explain informally the intended semantics of $\mathcal{L}_{\text{neq}}$ and discuss the relation between the interpretation of terms as *applications* of (possibly nondeterministic) operations and as sets of possible results.

2.1. Syntactic preliminaries

A specification in $\mathcal{L}_{\text{neq}}$ is a pair (Σ, Π), where the *signature* Σ is a pair of sets $(\mathbf{S}, \mathbf{F})$ of sorts $\mathbf{S}$ and operation symbols $\mathbf{F}$ (with arguments and result sorts in $\mathbf{S}$). There exists a denumerable set $\mathcal{V}$ of variables for every sort. For any syntactic entity (term, formula, set of formulae) χ, $\mathcal{V}[\chi]$ will denote the set of variables in χ. Letters from the end of the Latin alphabet, x, y, z are used for variables.

The set of terms over the signature Σ and a variable set X is denoted $W_{\Sigma, X}$. We always assume that, for every sort S, the set of ground terms of sort S, $(W_{\Sigma})_S$, is not empty.[4]

Π is a set of *clauses*. Each clause C is a set of literals (i.e., the ordering and multiplicity of the literals do not matter), written as $a_1, \ldots, a_n$ (possibly within "{ }"). A clause will be interpreted as disjunction of its literals and, sometimes, we will write it explicitly as $a_1 \vee \ldots \vee a_n$. Usually, we use uppercase Latin letters for single clauses, uppercase Greek letters for sets of clauses, and lowercase Latin letters a, e for literals. Comma indicates concatenation (set union) of clauses, and semicolon a collection. E.g., if C is $\{c_1, \ldots, c_n\}$, D is $\{d_1, \ldots, d_m\}$, and a is a literal (identified with an atomic clause for brevity), then "C ; D" denotes the conjunction of the two clauses, while "C, D" denotes the clause $\{c_1, \ldots, c_n, d_1, \ldots, d_m\}$. Similarly, "$C, a$" is the clause $\{c_1, \ldots, c_n, a\}$ and "C ; a" is the conjunction of C and $\{a\}$. "Π ; C" denotes the set (of clauses) Π extended with the clause C.

A literal (atomic clause) is either an *equation*, $t \doteq s$, an *inclusion*, $t \prec s$, or an *inequality*, $t \# s$, of terms $t, s \in W_{\Sigma, X}$. Equations and inclusions are called the *positive* literals (or atoms), and inequalities the *negative* literals.[5] We do not use a negaton of inclusion – as will be explained below, inequalities play the role of the complement of both equalities and inclusions. A clause $\{ t \# s,\ p \prec r \}$ is thought of as the conditional formula $\neg(t \# s) \Rightarrow\ p \prec r$, stating that p is included in r whenever t is not (necessarily) distinct from s. A clause with exactly one positive literal is called a *Horn formula*, and a Horn formula with no negative literals a *simple formula*.

All variables occurring in a clause are implicitly universally quantified over the whole clause. A clause is satisfied if, for every assignment to the variables, at least one of the literals is true.

For a specification SP=$(\Sigma,\ \Pi)$, $\mathcal{L}_{\text{neq}}$(SP) is the restriction of $\mathcal{L}_{\text{neq}}$ to $W_{\Sigma, \mathcal{V}}$.

2.2. $\mathcal{L}_{\text{neq}}$ specifications and informal semantics

Multialgebraic semantics [Hußmann 1993, Kapur 1980] interprets specifications in some form of power structures where the (nondeterministic) operations correspond to *set-valued functions*. This means that a (ground) term is interpreted as a set of *possibilities* – it denotes the set of *possible* results of the corresponding operation. We, on the other hand, want our formulae to express *necessary* facts, i.e., facts which have to hold in *every evaluation* of a program (specification). We think of each occurrence of a term of $\mathcal{L}_{\text{neq}}$ as referring to a *particular* (*arbitrary*) *application* of the operation which returns *one* of the possible results of the operation. Thus two occurrences of syntactically identical terms refer to two potentially distinct applications of the operation, and *syntactic identity of terms*

---

[4] We do not address the problem of empty sorts here and will present calculus which works under the assumption that sorts are not empty. We will usually give signatures with at least one constant for every sort but other ways of restricting the signatures [Goguen 1981, Huet 1980] or ensuring nonemptiness [Goguen 1981, Goguen 1987] can be used instead. It seems also that the most flexible approach which generalizes calculus by introducing explicit variables [Goguen 1981, Goguen 1982, Ehrig 1985] can be adapted to our framework.

[5] An equivalent (and the original) formulation of the syntax and the following results could use sequents A $\mapsto$ B, where all atoms in A are equalities (corresponding to $\#$) and all atoms in B equalities or inclusions. Because such a formulation complicates the calculus and proofs, we choose the present syntax of clauses.

does not, in general, *imply semantic identity*. Every two syntactic occurrences of a term t are thought of as *possibly distinct* applications of t. For nondeterministic terms this means that they may denote two distinct values.

This distinction between deterministic and nondeterministic terms is captured in the multialgebraic semantics by the modified notion of equality which we interpret (according to the discussion in section 1) as *necessary* equality. Although multialgebras interpret terms as sets of possible results, and hence two terms denoting the same set are equal *as sets*, we will make a clause $t \doteq t$ hold only if any two applications of t return the same result, i.e. if t is deterministic. $\doteq$ is then a *partial equivalence relation* and terms t for which $t \doteq t$ is valid are exactly the deterministic terms, denoted by $D_{SP,X}$. Variables will always be interpreted deterministically – all occurrences of a particular variable in a clause denote the same value (see definition 3.3). (For a discussion of the consequences of allowing variables to be interpreted nondeterministically, see [Walicki 1994a,Walicki 1994b].) We consider the loss of referential transparency implied by such a reinterpretation of equality to be an intrinsic feature of nondeterminism [e.g. Qian 1993]. And as a consequence our proof system only allows a restricted form of substitutivity.

Observe that operators with no arguments (which in theories of deterministic operations always denote *constants*) need not be constant at all in $\mathcal{L}_{neq}$. An operator such as $c\colon \to \mathrm{S}$ will have distinct meanings depending on whether the determinacy axiom "$c \doteq c$" is valid or not. If it follows from the specification then c denotes a usual deterministic constant. If not, then c corresponds to (possibly unbounded) nondeterministic choice from the set of elements of the sort S. This, together with inclusion and (limited) negation expressed by $\#$ allows us to utilize nondeterminism for defining and reasoning about subsort hierarchies at the term level (in a way reminiscent of unified algebras [Mosses 1989a]).

Interpreted in the multialgebras, "$s \prec t$" means that the possible results of s are contained in the set of possible results of t. This intuition is preserved when s and t are interpreted as particular applications – and not the whole sets of possible results – by the requirement that any result returned by (an arbitrary application of) s is also returned by some application of t.

There is a subtle difference between this and the set interpretation. Consider a user defined choice operation $\sqcup$, and the two terms "$\sqcup.\{0, 1\}$" and "$\sqcup.\{0, 1, 2\}$". One would expect the set of possible results of the former to be included in the set of possible results of the latter. But the validity of the clause "$\sqcup.\{0, 1\} \prec \sqcup.\{0, 1, 2\}$" will depend on the specification of $\sqcup$. Unless this inclusion is an axiom (or a consequence of the axioms) it is *not* (and should not be) valid, since the evaluation of $\sqcup.\{0, 1\}$ does not require an evaluation of $\sqcup.\{0, 1, 2\}$ – an acceptable implementation would have the former always returning the value 0, while the latter could always return 2. Thus it is not *necessarily* so that for every evaluation of the term $\sqcup.\{0, 1\}$ there exists an evaluation of $\sqcup.\{0, 1, 2\}$ returning the same value. If we insisted on a fairness requirement – that for any computation, if a particular term is evaluated sufficiently often, all its possible results would be realized – then $s \prec t$ would coincide with the subset relation between the result sets of s and t in the set approach. We do not explore the issue of fair nondeterminism further in this paper.

The above interpretation does not contradict facts like "$\sqcup.\{0, 1\} \prec \sqcup.\{0, 1\}$". This inclusion will always be valid – for any application of $\sqcup.\{0, 1\}$ (on the left) there is an application of $\sqcup.\{0, 1\}$

(on the right, namely the same one) such that the two return the same result.

Besides necessity, the aspect of *possibility* is present in $\mathcal{L}_{neq}$ in that we use clauses (disjunctive formulae) in specifications.

In the following section we introduce the multialgebraic semantics. In section 4 it is refined to the function oriented semantics which expresses directly the notion of particular applications of operations by means of oracle arguments.

3. The multialgebraic semantics

We define the semantics of $\mathcal{L}_{neq}$ specifications using multistructures. Our definitions are very similar to those used by other authors [Pickett 1967, Hesselink 1988, Hußmann 1990, Hußmann 1993, Kapur 1980, Nipkow 1986, Bauer 1985] except for the notion of equality.

Definition 3.1 (Multistructures). Let SP be an $\mathcal{L}_{neq}$ specification. M is an SP-*multistructure* if

1. its carrier $|M|$ is an **S**-sorted set and
2. for every $f: S_1 \times \ldots \times S_n \to S$ in **F** there is a corresponding function $f^M: M_{S_1} \times \ldots \times M_{S_n} \to \mathcal{P}^+(M_S)$.

□

Operations applied to sets refer to their unique pointwise extensions. We let *MStr*(*SP*) denote the class of SP-multistructures. It has the distinguished term structure:

Definition 3.2 (Term structure). The *term multistructure* W_Σ for a specification $SP=(\Sigma, \Pi)$ is defined as:

1. for each $S \in \mathbf{S}$, $(W_\Sigma)_S$ is the set of ground terms of sort S,
2. for each $f: S_1 \times \ldots \times S_n \to S$ in **F**, $t_i \in (W_\Sigma)_{S_i}$: $f^{W_\Sigma}(t_i, \ldots, t_n) = \{f(t_i, \ldots, t_n)\}$

□

Definition 3.3. Let M be an SP-multistructure, $\beta: X \to |M|$ an assignment to a set of variables X. M $\mathcal{P}$*–satisfies* a literal *a* under β iff

- *a* is $s \prec t$ and $\beta[s] \subseteq \beta[t]$
- *a* is $s \#t$ and $\beta[s] \cap \beta[t] = \emptyset$
- *a* is $s \doteq t$ and $\beta[s] \equiv \beta[t]$, where $A \equiv B$ iff A and B are the same 1-element set.

M $\mathcal{P}$*-satisfies* a clause *C*, $M \models_{\mathcal{P}} C$, iff for each β, M $\mathcal{P}$-satisfies at least one literal of *C*.

An SP-*multimodel* is an SP-multistructure which satisfies all the axioms of SP. *MMod*(*SP*) denotes the class of multimodels of SP.

□

We let $\models$ denote the standard satisfiability relation and annotate it, as in $\models_{\mathcal{P}}$, to indicate that a particular definition of satisfiability is being used. Notice that although different occurrences of a given (nondeterministic) term *t* will denote one and the same set, the interpretation of $\doteq$ will make $t \doteq t$ false, if this set contains more than one element. The reason for using empty intersection (and not set inequality) as the interpretation of the negative literals $s \# t$ is the same as using "elementwise" equality as the interpretation of $\doteq$. Since we avoid set equality in the positive sense, the most natural negative form seems to be the one we have chosen. Under this interpretation $\doteq$ denotes *necessary equality* and $\#$ denotes *necessary inequality*, i.e. $t \# s$ is valid only if t and s *never* return the same value. For deterministic terms t and s, $s \# t$ is the same as $s \neq t$, i.e., deterministic negative literals correspond exactly to equational (deterministic) conditions.

Initiality of the term multistructure in MStr(SP) depends on the definition of multi-homomorphism. It is a known fact [Hußmann 1993, Mosses 1989a] that, in the general case, one should not expect the existence of initial multi*models*. In [Hußmann 1993] Hußmann has shown that such multimodels may not exist even if the specification language is restricted to simple formulae. Therefore we allow general clauses in the specifications and will concentrate on the whole class of multimodels of a specification. For the discussion of the structure of the class of multistructures and (the lack of) initiality results, the reader is referred to [Hußmann 1993, Walicki 1993a]. In the present context, the significance of the term structure is expressed in

Lemma 3.4. If M is an SP-multistructure then, for every set of variables X and assignment β: $X \to |M|$, there exists a unique function $\beta[]$: $W_{\Sigma,X} \to \mathcal{P}^{+}(M)$ such that:

$\beta[x] = \{\beta(x)\}$ $\qquad$ $\beta[c] = c^{M}$ $\qquad$ $\beta[f(t_i, ..., t_n)] = \bigcup\{f^{M}(t1, ...tn) \mid ti \in \beta[t_i]\}$

□

Observe that, as a consequence of the definition of multistructures, operations in M are $\subseteq$-monotonic, i.e., $\beta[s] \subseteq \beta[t] \Rightarrow \beta[f(s)] \subseteq \beta[f(t)]$.[6]

According to definition 3.3 variables are assigned only single elements from the carrier $|M|$ and so always denote individual values. This means that, for instance, the clause $s \# t$ is equivalent to $\{x \# s, x \# t\}$. Considering x to be an individual and t a set, $x \# t$ may be understood as the more familiar $x \notin t$. Thus we may treat our notation as an abbreviation for more elaborate formulae with two negated membership relations and one new variable (not occurring in the rest of the clause). Because of the special role the literals of the form $x \# t$ with $x \in \mathcal{V}$ play we will call them *bindings*.

As a simple example of the formalism and the important role of bindings, consider the following specification:

Example 3.5

S:	Nat	
F:	0: → Nat	(zero)
	s: Nat → Nat	(successor)
	_⊔_: Nat×Nat → Nat	(binary nondeterministic choice)
Π:	1. $0 \doteq 0$	
	2. $s(x) \doteq s(x)$	
	3. $1 \# 0$	(As usual, we abbreviate $s^{n}(0)$ as n.)
	4. $0 \sqcup 1 \doteq 0 \vee 0 \sqcup 1 \doteq 1$	

□

[6] This is the reason for calling them "multistructures," and distinguishing them from power structures (where operations go from power sets to power sets rather than from sets to power sets, and where operations need not be $\subseteq$-monotonic).

The first two axioms make zero and successor deterministic, and the third makes 0 distinct from 1. (Thus $\#$ serves as a limited form of negation.) In writing clause 4 one could have intended to require that any application of $0\sqcup 1$ returns either 0 or 1. But according to the interpretation of terms as arbitrary applications of the operation, it actually states for arbitrary two applications of the term $0\sqcup 1$ returning the values n_1 and n_2 respectively, n_1 will be 0 or n_2 will be 1. Since these were arbitrary pairs of applications, the effect would be that either every application of $0\sqcup 1$ returns 0 or every such application returns 1, i.e., that choice is not really nondeterministic but merely underspecified.

To get the desired meaning we need to state that axiom 4 holds when both occurrences of $0\sqcup 1$ refer to *the same application*. The means of binding a particular application of an operation could be obtained by introducing additional constructs such as "**let** c=$0\sqcup 1$ **in** ..." but we prefer to keep the language as compact as possible. The desired effect is obtained by (1) observing that the semantics of $\mathcal{L}_{\text{neq}}$ requires to assign only single values to variables, i.e., within a particular clause, a variable will denote one specific value. And (2) by making the identification of a particular term application with a variable in a negative literal. The variable may be then used more than once in the clause referring to the same value returned by this particular application.

Example 3.5b

Replace axiom 4 of example 3.5 with

$4'.\ x \# (0\sqcup 1) \vee x \doteq 0 \vee x \doteq 1$

□

Axiom 4′ states that for a given value x, x is either impossible as a result of $0\sqcup 1$, or x is 0 or 1. Viewing $0\sqcup 1$ as a set (of possible results), this represents the conditional formula: $x \in 0\sqcup 1 \Rightarrow x \doteq 0 \vee x \doteq 1$. In terms of application of operations, the axiom indicates that whenever x is the result returned by some application of $0\sqcup 1$ then it equals 0 or 1.

General choice is a very natural nondeterministic operator. Unless indicated otherwise, we will assume that it takes a non-empty, finite set as argument and satisfies the following axioms (where "{_}" returns a 1-element set, and "add(S, y)" adds an element y to the set S)[7]:

Choice:

F: $\sqcup.\_ : \mathrm{Set(T)} \rightarrow \mathrm{T}$

Π: $\sqcup.\{x\} \doteq x$

$x \# \sqcup.add(S, y) \vee x \doteq y \vee x \prec \sqcup.S$

□

Notice the "$\prec$" in the last formula. The operational interpretation of this clause is that the evaluation of a choice from the set $add(S, y)$ amounts to either taking y or evaluating a choice from the set S.

[7] In order not to elaborate unnecessary details we assume here, and in the following examples, that some specifications of the needed sorts (such as finite sets: Set, finite sequences: Seq) and operations are given.

4. Deriving deterministic specifications

In this section we define an alternative semantics of our specifications. A model M of a specification will now be a "particular computation" or, more precisely, a standard deterministic algebra where a nondeterministic operation is interpreted as a set of deterministic functions.

We describe a transformation of a nondeterministic $\mathcal{L}_{neq}$ specification into an *underspecified* deterministic specification over the equational language $\mathcal{L}_{deq}$. The transformation preserves validity of $\mathcal{L}_{neq}$ clauses in the sense that the model class of the derived specification satisfies a clause C iff the class of multimodels satisfies C. The transformation goes via an intermediary language $\mathcal{L}_{\exists}$. $\mathcal{L}_{\exists}$ is $\mathcal{L}_{neq}$ without inclusion, but with existentially quantified equations, and $\mathcal{L}_{deq}$ is simply $\mathcal{L}_{neq}$ with equality and inequality as the only primitives.

The semantics of an $\mathcal{L}_{neq}$ specification SP can then be defined alternatively in terms of the semantics of its $\mathcal{L}_{deq}$ transform SP′. The advantage of that is that the semantics of nondeterministic specifications can be given in terms of standard algebraic constructions. We also consider the validity-preserving refinement of the specification language in the transition from $\mathcal{L}_{neq}$ to $\mathcal{L}_{deq}$ (together with the semantics of the latter) a plausible model for observable nondeterminism which results from the presence of invisible parameters.

4.1. Transforming $\mathcal{L}_{neq}$ specifications into deterministic specifications.

In the following, for an $\mathcal{L}_{neq}$ syntactic entity χ (be it a signature, term, equation, inclusion or clause), we let $\chi_{\exists}$ denote the $\mathcal{L}_{\exists}$ transform and χ' the $\mathcal{L}_{deq}$ transform of χ.

The main step in transforming $\mathcal{L}_{neq}$ specifications is the transformation of the nondeterministic operators into underspecified, but deterministic, functions. This is done by equipping every operator symbol with an additional argument called an *index*.[8] The index argument(s) of an $\mathcal{L}_{\exists}$ or $\mathcal{L}_{deq}$ term identifies which *application* of the $\mathcal{L}_{neq}$ operators the term refers to. Thus the first step is to extend the $\mathcal{L}_{neq}$ signature Σ with a new index sort I and to modify the profiles of the operations:

> **Definition 4.1 (Index sort)**. Given an $\mathcal{L}_{neq}$ signature $\Sigma=(\mathbf{S}, \mathbf{F})$, the derived $\mathcal{L}_{\exists}$ signature $\Sigma_{\exists}$ is the pair $(\mathbf{S}_{\exists}, \mathbf{F}_{\exists})$ where:
>
> $\mathbf{S}_{\exists} = \mathbf{S} \cup \{I\}$ $\qquad$ $\mathbf{F}_{\exists} = \mathbf{Fi} \cup \{\text{if\_then\_else\_}: \text{Bool}\times I\times I\to I\}$
>
> where I is a new sort, and where every f: $S_1\times\ldots\times S_i\to S_{i+1}$ in $\mathbf{F}$ is converted into the corresponding f: $I\times S_1\times\ldots\times S_i\to S_{i+1}$ in $\mathbf{Fi}$.[9]
>
> □

Note that we index *all* operations, also the ones which are supposed to be deterministic. What

---

[8] In terms of the intended interpretation indices correspond to *oracles*. At the level of the derived deterministic specification, however, they are merely additional arguments to the operations. We therefore prefer the semantically more neutral word "index".

[9] In [Walicki 1993a], where the above construction was introduced, one also added a new constant λ of sort I to ensure that the sort is not empty. This extension has also important consequences for the existence of initial/quasi-initial semantics which was studied there. Here we merely assume that all sorts are not empty. The if_then_else_ function on index sorts is necessary only in one special case in the proof of soundness.

distinguishes deterministic operations from the nondeterministic ones in an $\mathcal{L}_{\text{neq}}$ specification is the validity of simple equations of the form $f(x) \doteq f(x)$, saying that every application of the term $f(x)$ evaluates to the same value as every (other) application of the term $f(x)$. In $\mathcal{L}_{\exists}$ terms this translates into the assertion that the index argument of f does not affect the result of its application. Thus the corresponding $\mathcal{L}_{\exists}$ axiom will be $f(i, x) = f(j, x)$ where i and j are distinct variables. That is, since the index value represents a particular application of the operation, arbitrary application is represented by an arbitrary index value – different occurrences of terms are indexed by distinct index variables. (As a notational convenience we shall often omit the indices of deterministic terms.) Transformation of equalities and inequalities, in general, will conform to this schema.

Inclusion $l \prec r$ is to mean that for every evaluation of l there exists an evaluation of r returning the same result. In terms of indices this amounts to the existence, for every index i at l, of an index j at r such that $l(i)$ and $r(j)$ evaluate to the same result. We transform $\mathcal{L}_{\text{neq}}$ axioms according to:

Definition 4.2. The $\mathcal{L}_{\exists}$ transform $C_{\exists}$ of a given $\mathcal{L}_{\text{neq}}$ axiom C: $\{a_1, \ldots, a_m\}$ is obtained as follows:

1. introduce *distinct* index variables for *all non-variable* subterms in C obtaining $\{a_{1\exists}, \ldots, a_{m\exists}\}$,
2. for every inclusion $l \prec r$ in C and every index variable i introduced in $r_{\exists}$, we add an existential quantifier $\exists i$ in front of the clause obtained in 1.
3. Finally, all symbols "$\prec$" and "$\doteq$" are replaced with "$=$", and all "$\not\doteq$" with "$\neq$".

□

For instance, let ξ be the axiom (from the Choice example): $x \not\doteq \sqcup.add(S, y) \vee x \doteq y \vee x \prec \sqcup.S$. The $\mathcal{L}_{\exists}$ transform of ξ will be:

$$\xi_{\exists}: \quad \exists i_3 \bullet \; x \neq \sqcup(i_1, add(i_2, S, y)) \;\vee\; x = y \vee \; x = \sqcup(i_3, S)$$

Note that, as a consequence of the second point of this definition, all existential quantifiers (if any) will refer only to index variables in the positive literals of $C_{\exists}$.

Definition 4.3 ($SP_{\exists}$). A deterministic specification $SP_{\exists} = (\Sigma_{\exists}, \Pi_{\exists})$ is obtained from $SP = (\Sigma, \Pi)$ by extending Σ as in definition 4.1 and taking $\Pi_{\exists} = \{C_{\exists} \mid C \in \Pi\}$.

□

$SP_{\exists}$ is a deterministic specification, but clauses in $\Pi_{\exists}$ may contain existential quantifiers. To obtain an equational specification in $\mathcal{L}_{\text{deq}}$ we Skolemize all $\mathcal{L}_{\exists}$ clauses in which existential quantifiers occur – we introduce a new indexing (Skolem) function for every existential quantifier. The $\mathcal{L}_{\text{deq}}$ clause corresponding to the clause $\xi_{\exists}$ is then:

$$\xi': \quad x \neq \sqcup(i_1, add(i_2, S, y)) \;\vee\; x = y \vee \; x = \sqcup(\sigma(x, y, S, i_1, i_2), S)$$

where σ is a new Skolem function. (The Skolem functions can be simplified by removing redundant arguments, e.g., σ will not depend on i_2 (if *add* is deterministic) nor on x (which is determined by i_1). Similarly, variables occurring in unique bindings can be eliminated by substitution when constructing $C_{\exists}$. Such simplifications have only bearing on possible implementations and we are not concerned with them here.)

Definition 4.4 (Derived specification). Given an $\mathcal{L}_{\text{neq}}$ specification SP=$((\mathbf{S}, \mathbf{F}), \Pi)$, the derived $\mathcal{L}_{\text{deq}}$ specification SP′ is $((\mathbf{S}', \mathbf{F}'), \Pi')$ where:
$\mathbf{S}' = \mathbf{S}_\exists$.
$\Pi' = \{C' \mid C_\exists \in \Pi_\exists\}$ where C' is the Skolemized version of $C_\exists$.
$\mathbf{F}' = \mathbf{F}_\exists \cup \{\boldsymbol{\sigma} \mid \boldsymbol{\sigma}$ an indexing function from $\Pi'\}$.
□

4.2. Computations

Both $\text{SP}_\exists$ and SP′ are ordinary first order theories, and so their classes of structures, $\text{Str}(\text{SP}_\exists)$ resp. Str(SP′), are defined in the usual way [e.g. Chang 1977]. Models are structures that satisfy the axioms and so the definition of the model classes, $\text{Mod}(\text{SP}_\exists)$ resp. Mod(SP′), is the usual one too. By the standard argument concerning the preservation of satisfiability by the process of Skolemization [Chang 1977], the two model classes are equal in the sense that every $M_\exists \in \text{Mod}(\text{SP}_\exists)$ can be turned into an $M' \in \text{Mod}(\text{SP}')$ by supplying an interpretation for the indexing functions, and every $M' \in \text{Mod}(\text{SP}')$ can be turned into an $M_\exists \in \text{Mod}(\text{SP}_\exists)$ by forgetting such an interpretation.

Of course, our main concern will be with the class Mod(SP′), since SP′ is as close to an equational specification as we can get.

Definition 4.5. For an $\mathcal{L}_{\text{neq}}$ specification SP *the class of SP* (*computational*) *structures*, *CStr*(*SP*), is defined to be the class Str(SP′). The (loose) semantics of SP, *Comp*(*SP*), is defined to be the class Mod(SP′).[10]
□

Since $M \in \text{Mod}(\text{SP}')$ corresponds to a particular choice of functions satisfying SP, we will call the members of Comp(SP) *computations* of SP.

Just like MMod(SP) did not, in general, possess an initial object, so at the level of $\mathcal{L}_{\text{deq}}$ specifications one cannot hope to find an initial algebra appropriate for the interpretation of general clauses. The reason is well illustrated by Meseguer [Meseguer 1990, Meseguer 1992] and Wirsing [Wirsing 1989], and is a consequence of the fact that disjunctive equations where none of the alternatives are maximal (valid in all models) do not admit initial models. One can, however, obtain initial algebras by banning disjunctions – i.e., by allowing only Horn formulae (as it is done in [Hußmann 1993] and [Mosses 1989a]).

Instead of looking for initial algebras exclusively, we therefore consider the class of all structures satisfying the axioms of the specification. The proponents of initial semantics may find partial satisfaction in the proposals of generalizing this notion to *quasi-initiality* [Goguen 1988, Kaplan 1987, Volger 1989a, Volger 1989b, Wolter 1992]. Quasi-initiality yields multiple non-

[10] The term "computational structure" is not to be confused with the "computation structure" as used, for instance, in [Wirsing 1989]. The latter is a synonym for "reachable structure" while computational structures as defined here are new kinds of models of nondeterminism.

isomorphic structures, but nevertheless conforms to the dictum of "no junk – no confusion" cited as the main justification for the initial semantics. The results from [Wolter 1992] are directly applicable at the level of computations – any equational specification with Gentzen formulae (such as SP′) admits quasi-initial semantics. A mild restriction on the specifications allows us to lift quasi-initial semantics to the multialgebra level [Walicki 1993a].

In order to relate the classes Comp(SP) and MMod(SP) we define what it means for a computational structure to satisfy an $\mathcal{L}_{\text{neq}}$(SP) formula. There are at least two alternatives here and we record both of them:

Definition 4.6. Let C be an $\mathcal{L}_{\text{neq}}$(SP) clause. M∈CStr(SP) ∃-*satisfies* C, $M \models_{\exists} C$, iff $M \models C_{\exists}$.
□

The other possibility is to tighten the requirements for satisfiability of inclusions. (Recall that existential quantifiers in $C_{\exists}$ bound, at most, index variables introduced into inclusions.) Instead of considering an inclusion $s \prec t$ satisfied if for any index value at s there exists an index value at t making the two equal, we may require that the index at t must be a function of the index at s denotable by the indexing functions from the translated specification.

Definition 4.7. Let C be an $\mathcal{L}_{\text{neq}}$(SP) clause, and $i_1, \ldots, i_z$ be all the existentially quantified variables in $C_{\exists}$. M∈CStr(SP) $\mathcal{C}$*–satisfies* C, $M \models_{\mathcal{C}} C$, iff there exist $\mathcal{L}_{\text{deq}}$ index terms $d_1, \ldots, d_z$ such that $M \models (C^{-}_{\exists})^{i_1,\ldots,i_z}_{d_1,\ldots,d_z}$, where $C^{-}_{\exists}$ is $C_{\exists}$ with all existential quantifiers removed.
□

$M \models (C^{-}_{\exists})^{i_1,\ldots,i_z}_{d_1,\ldots,d_z}$ denotes the standard satisfaction for all assignments of elements from M to the variables in the clause. Notice that $d_1, \ldots, d_z$ can be arbitrary terms – also single variables possibly occurring in $C^{-}_{\exists}$.

To illustrate the difference between the two definitions, let s, t be two constants of sort S, and let σ be the only indexing function introduced in the translated specification SP′. Let M∈CStr(SP) be as follows: $S^M = \{1, 2\}$, $I^M = \{\underline{a}, \underline{b}\}$, $\sigma^M(i)=i$, $s^M(\underline{a})=1$, $s^M(\underline{b})=2$, $t^M(\underline{a})=2$, $t^M(\underline{b})=1$, and consider the inclusion $s \prec t$. Obviously, $M \models \exists\, j \bullet s(i)=t(j)$ but $M \not\models s(i)=t(\sigma^n(i))$ for any n.

The requirement of the existence of indexing terms $d_1, \ldots, d_z$ rather than just existence of elements in the carrier, will make a clause $\mathcal{C}$-valid only if there exists a corresponding $\mathcal{L}_{\text{deq}}$ clause C' (with the index terms $d_1, \ldots, d_z$) which is valid. This explains the relevance of $\mathcal{C}$–satisfiability – validity of an $\mathcal{L}_{\text{deq}}$ clause C' will imply validity of the corresponding $\mathcal{L}_{\text{neq}}$ clause C, and the former can be examined using standard techniques. (Of course, one is still left with the problem of finding appropriate index terms.) Obviously, for a given M, the two definitions are not equivalent. $M \models_{\mathcal{C}} C$ implies $M \models_{\exists} C$, but not vice versa. However, it is an interesting consequence of the completeness theorem that both notions of validity coincide on the class of all computations (corollary 5.3).

4.3. Comp(SP) and MMod(SP)

We have introduced computational models because, in our opinion, they offer a more operationally intuitive interpretation of $\mathcal{L}_{\text{neq}}$ specifications than multialgebras do. The latter can be seen as a convenient abstraction of computations and an intermediate level between the external specification and its internal semantics.

Lemma 4.8. Define the operator $\mathcal{P}$: CStr(SP)→ MStr(SP):

1. $(\mathcal{P}M)_S = M_S$, for every $S \in \mathbf{S}$.
2. $f^{\mathcal{P}M}(x) = \{f^M(i, x) \mid i \in M_I\}$, for every $f \in \mathbf{F}$.

If M∈Comp(SP) then $\mathcal{P}$M∈MMod(SP).

□

The $\mathcal{P}$ operator[11] abstracts away the information about indices and maps operations to their respective result sets. By the lemma, every computation determines a multimodel. There will be many computations with the same multimodel

Definition 4.9. A relation ≈ is defined on Comp(SP) by A≈B iff $\mathcal{P}$A=$\mathcal{P}$B.

□

≈ is an equivalence relation, and the ≈-equivalence class [A] corresponds to the classification of its elements as possible computations of the program (specification) which is modeled by the multialgebra $\mathcal{P}$A. It is easy to see that, in general, an ≈-equivalence class does not possess initial or terminal elements. Nonexistence of such computations is hardly surprising. We may even say that the opposite would not be desirable. One should not expect to find "the best" (generic, initial) computation of a nondeterministic program since such computations realize incommensurable possibilities of the program.

Thus the opposite transition – from MMod(SP) to Comp(SP) – cannot be defined uniquely unless we choose some arbitrary member from each equivalence class. What can be shown is that MMod(SP) contains no "redundant" models, i.e., there are no models which do not have a computational interpretation – every multimodel can be obtained by applying $\mathcal{P}$ to some computation.

Lemma 4.10. $\forall M \in$ MMod(SP) $\exists A \in$ Comp(SP) • $\mathcal{P}$A=M.

□

The following lemma is an easy consequence of definition 4.8 of the $\mathcal{P}$-operator:

Lemma 4.11. For any A∈CStr(SP) and clause C • A$\models_\exists C$ iff $\mathcal{P}$A$\models_{\mathcal{P}} C$.

□

This, together with the previous lemmas, leads to

Lemma 4.12. For any clause C • Comp(SP)$\models_\exists C$ iff MMod(SP)$\models_{\mathcal{P}} C$.

□

As a consequence of the completeness theorems reported in the next section we will be able to further strengthen the equivalence of Comp and MMod (corollary 5.3).

---

[11] It is easily extended to a functor if we look at Comp(SP) and MMod(SP) as categories.

5. The $\mathcal{L}_{\mathbf{neq}}$ calculus NEQ for specifications of nondeterministic operators

The rules of NEQ are:

R1: a) $\vdash x\#y, x\doteq y$ b) $\vdash x\#t, x\prec t$ $x, y\in\mathcal{V}$

R2: $$\frac{\vdash C^x_t \quad ; \quad \vdash D, s\doteq t}{\vdash C^x_s, D}$$

R3: $$\frac{\vdash C^x_t \quad ; \quad \vdash D, s\prec t}{\vdash C^x_s, D}$$ x not in a right-hand side of $\prec$ in C

R4: $$\frac{\vdash C, s\preceq t \quad ; \quad \vdash D, s\#t}{\vdash C, D}$$ (CUT) ($\preceq$ being either $\doteq$ or $\prec$)

R5: $$\frac{\vdash C}{\vdash C, e}$$ (WEAK)

R6: $$\frac{\vdash C, x\#t}{\vdash C^x_t}$$ (ELIM) $x\in\mathcal{V}-\mathcal{V}[t]$, at most one x in C

C^x_t denotes C with t substituted for x. A few comments regarding the rules may be in order.

R1 expresses the relation between $\#$ and equality and inclusion. Since variables x and y are individuals, the two rules correspond to, respectively, $x\neq y \vee x=y$, and $x\notin t \vee x\in t$. They also capture the fact that '$\doteq$' is a partial equivalence relation and is reflexive only for variables (see PER below).

R2 is a paramodulation rule allowing replacement of deterministic terms (in the case when $s\doteq t$ holds in the second assumption). In particular, it allows derivation of the standard substitution rule when the substituted terms are deterministic, and prevents substitution of nondeterministic terms for variables.

R3 allows "specialization" of a clause by substituting for a term t another term s which is included in t. The restriction that the occurrences of t which are substituted for don't occur in the right-hand side of $\prec$ in C is needed to prevent, for instance, the unsound conclusion $\vdash p\prec s$ from the premises $\vdash p\prec t$ and $\vdash s\prec t$.

$s\#t$ implies both negation of $s\doteq t$ and of $s\prec t$. R4 allows us to resolve these complementary literals.

R5 allows one to weaken the premise clause by extending it with additional disjuncts.

R6 eliminates redundant bindings, namely those that bind an application of a term occurring at most once in the rest of the clause.

We will write $\Pi\vdash C$ to indicate that C is provable from Π with the calculus NEQ.

The following derived rules allow us to rename variables in a clause, substitute deterministic terms for variables and introduce bindings, i.e., exchange several occurrences (applications) of a term with one unique application of it. ($x, y, z\in\mathcal{V}$, $t\in W_{\Sigma,X}$)

PER: a) $\vdash x\doteq x$ b) $\vdash t\prec t$ proof: R1a, resp. R1b, and R6

REN: $\dfrac{\vdash C}{\vdash C^x_y}$ proof: $\dfrac{\dfrac{\vdash C \quad ; \quad \vdash x\#y,\ x\doteq y}{\vdash C^x_y,\ x\#y}\text{R2}}{\vdash C^x_y}\text{R6}(x)$

SUB: $\dfrac{\vdash C \quad ; \quad \vdash D, t\doteq t}{\vdash C^x_t, D}$ proof: $\dfrac{\dfrac{\dfrac{\vdash C}{\vdash C^x_z}\text{REN }(z\notin\mathcal{V}[t,C,D]) \qquad \dfrac{\vdash D, t\doteq t \quad ; \quad \vdash z\#t, z\prec t}{\vdash D, t\doteq z,\ z\#t}\text{R3}}{\vdash \left(C^x_z\right)^z_t, D,\ z\#t}\text{R2}}{\vdash C^x_t, D}\text{R6}(z)$

INTR: $\dfrac{\vdash C^y_t}{\vdash C^y_x,\ x\#t}$ y not in a right-hand side of $\prec$ proof: $\dfrac{\vdash C^y_t \quad ; \quad \vdash x\#t,\ x\prec t}{\vdash C^y_x,\ x\#t}\text{R3}$

The INTR rule allows binding of several applications (occurrences) of a term and, for instance, to conclude that if $\vdash h(g, g)\prec f(g)$ then $\vdash h(x, x)\prec f(g),\ x\#g$. Bindings reflect the computational interpretation of operations as returning unique results and are crucial for completeness of NEQ as the following example illustrates [Hußmann 1993, Walicki 1993a].

Example 5.1

Let SP be $(\mathbf{S}, \mathbf{F}, \Pi)$ where

$\mathbf{S}$: S

$\mathbf{F}$: $a, b, g\colon \to \mathrm{S}$ $\quad f\colon \mathrm{S}\to\mathrm{S}$ $\quad h\colon \mathrm{S}\times\mathrm{S}\to\mathrm{S}$

Π: 1. $a\doteq a$ $\quad$ 2. $b\doteq b$ $\quad$ 3. $a\prec g$ $\quad$ 4. $x\prec h(x,x)$ $\quad$ 5. $h(x,x)\prec f(x)$

We have that $\mathrm{MMod(SP)}\models_{\mathcal{P}} g\prec f(g)$ and the semantic reasoning to this effect goes as follows: for any $\mathrm{M}\in\mathrm{MMod(SP)}$, $\mathrm{m}\in\mathrm{M_S}$

$$\mathrm{m}\in g^{\mathrm{M}} \overset{4.}{\Rightarrow} \mathrm{m}\in h^{\mathrm{M}}(\mathrm{m},\mathrm{m}) \overset{5.}{\Rightarrow} \mathrm{m}\in f^{\mathrm{M}}(\mathrm{m}) \overset{\mathrm{m}\in g\,;\,f\,\subseteq\text{-monoton.}}{\Rightarrow} \mathrm{m}\in f(g)^{\mathrm{M}}$$

If we did not have bindings and the ELIM rule, this inclusion would not be derivable. With R3 we can only obtain $x\prec f(x)$, from which R5 gives $\{x\#g,\ x\prec f(x)\}$. But since $g\doteq g$ is not derivable we cannot substitute g for x.

The reasoning can proceed only if it can consider "elementwise" transitions. The derivation in NEQ proceeds as follows:

$$\dfrac{\dfrac{\text{R3}\ \dfrac{\vdash h(x,x)\prec f(x) \quad ; \quad \vdash x\prec h(x,x)}{\vdash x\prec f(x)} \qquad \dfrac{\vdash f(g)\prec f(g)}{\vdash f(x)\prec f(g),\ x\#g}\text{INTR}}{\vdash x\prec f(g),\ x\#g}\text{R3}}{\vdash g\prec f(g)}\text{ELIM}$$

□

The example illustrates the significance of making explicit the distinction between operations and their applications, since this distinction plays a crucial role at the semantic level.

The main new result concerning the two semantics and the calculus is:

Theorem 5.2. For every $\mathcal{L}_{\text{neq}}$ specification SP=(Σ, Π) and clause $C \in \mathcal{L}_{\text{neq}}$(SP),

1) MMod(SP)$\models_{\mathcal{P}} C$ iff $\Pi \vdash C$.
2) Comp(SP)$\models_{\mathcal{C}} C$ iff $\Pi \vdash C$.

□

Soundness is proven by induction on the length of the proof $\Pi \vdash C$. Completeness in 1) is not proven directly but via the derived deterministic semantics and completeness in 2).

Corollary 5.3. For every $\mathcal{L}_{\text{neq}}$ specification SP=(Σ, Π) and clause $C \in \mathcal{L}_{\text{neq}}$(SP)

Comp(SP)$\models_{\mathcal{C}} C$ iff Comp(SP)$\models_{\exists} C$ iff MMod(SP)$\models_{\mathcal{P}} C$.

□

The implications from left to right are easy consequences of the definitions of satisfiability, and the completeness theorems yield the equivalence MMod(SP)$\models_{\mathcal{P}} C$ iff Comp(SP)$\models_{\mathcal{C}} C$.

That $\models_{\exists}$ implies $\models_{\mathcal{C}}$ is, perhaps, not surprising in view of the fact that Comp(SP) is closed under subalgebras. Thus if all computations $\exists$-satisfy a clause C (i.e., its existentially quantified version $C_{\exists}$) then all reachable computations will $\exists$-satisfy C. In particular, the initial covering of the class Comp(SP), i.e., all computations initial in their respective component of the class, will satisfy C. This means that the index terms whose interpretation in a quasi initial object I_0 gives $I_0 \models_{\mathcal{C}} C$ can be used to validate C in other computations to which there exists a unique homomorphism from I_0.

Corollary 5.3 significantly strengthens such an argument. There is an obvious difficulty with applying the above argument to negative formulae as they are not necessarily preserved under homomorphisms [Lyndon 1959]. More importantly, the above reasoning does not guarantee that for an existential quantifier in $C_{\exists}$ there will exist *one* index term – the terms may be different in different components of Comp(SP). But the corollary implies that if in all computations (multimodels) the result set of an operation s is included in the result set of an operation t then computing s *really depends* on computing t – there exists an index term $d(i)$ such that $s(i)=t(d(i))$.

The corollary shows also that Comp(SP) and MMod(SP) satisfy the same $\mathcal{L}_{\text{neq}}$(SP) formulae. Hence the former constitutes a refinement of MMod(SP) only in so far as the language is concerned. $\mathcal{L}_{\text{deq}}$ contains more detailed description of the operations using new indexing functions which are not present in $\mathcal{L}_{\text{neq}}$.

6. An Example

As an example of our formalism we give a specification of an access data type. Nondeterminism provides here an invaluable abstraction mechanism which releases the specifier from the burden of describing exactly the mechanism of allocating new pointers. This leads not only to a more abstract specification but also avoids the problems of distinguishing the structures which are supposed to have the same observational behavior. Motivation for this example comes from [Meldal 1987, Meldal 1989]. First, we illustrate the problems arising in the case of the usual deterministic underspecification and then show how they can be avoided by introducing nondeterministic operations.

We have the following sorts and operations: sort A of the accessed values, sort P_A of access values (pointers), and sort C_A which is a collection of access values P_A. Collections were introduced by Hoare and Wirth in [Hoare 1973]. An access value serves here as an index to a set of the referenced type. Changes to a referenced object as well as allocation of new objects are considered changes to the collection in a way quite similar to the treatment of arrays.

The operation *Init* produces a new, empty collection, *St*(*ore*) takes a collection, an access value and an object and allocates the object with the given access value returning a new collection, *Next* gives an access value of the next object to be allocated in the current collection, and *Acc*(*ess*) accesses the value through a given access value. Finally, the operation *New* allocates a new object with the given access value in a collection and returns the extended collection.

To simplify the example we will ignore things like NIL pointers with no accessed value, deallocation and the problems of limited storage space. First we give a deterministic underspecified version to illustrate the need for a more sophisticated abstraction mechanism. To mimic as far as possible the expressiveness of $\mathcal{L}_{\text{neq}}$ we allow here use of equalities, inequalities and disjunctions.

D-POINTER **is**

S: $\{ A, P_A, C_A \}$

F: $\{ \perp : \; \to A$ (an error value)

$\quad Init: \; \to C_A$

$\quad St\_: \; C_A \times P_A \times A \to C_A$

$\quad Next\_: \; C_A \to P_A$

$\quad New\_: \; C_A \times A \to C_A$

$\quad Acc\_: \; C_A \times P_A \to A \}$

• $x, y: P_A,\ C: C_A,\ a, b: A \in \mathcal{V}$

Π: { 1. $St(St(C, x, a), y, b) \doteq St(St(C, y, b), x, a),\ x \neq y$

2. $St(St(C, x, a), x, b) \doteq St(C, x, b)$

Acc: 3. $Acc(St(C, y, a), x) \doteq Acc(C, x),\ x \neq y$

4. $Acc(St(C, x, a), x) \doteq a$

5. $Acc(Init, x) \doteq \perp$

Next: 6. $Acc(C, Next(C)) \doteq \perp$

New: 7. $New(C, a) \doteq St(C, Next(C), a)$ }

end

The first axiom tells that the order in which objects are allocated is inessential as long as they have different access values. 2. makes the last *St* operation at access value x override all previous ones. (These two axioms are equivalent to the conditional $St(St(C, x, a), y, b) \doteq$ if $x{=}y$ then $St(C, y, b)$ else $St(St(C, y, b), x, a)$.) Accessing a value x goes along the nesting of *St* until the access value equal to x is found or until the *Init* collection is reached. In the latter case $\perp$ is returned indicating that x has not been allocated. Axiom 6. ensures that *Next* does not return an allocated value.

Axiom 7. defines the *New* operation in the expected way. Since this operation is fully defined in terms of *St* and *Next*, it might be considered merely a convenient abbreviation. However, it is natural to assume that this is the operation we are actually interested in – the operation *Next* is needed to define it. Observe that *New* does not return a pointer but merely modifies the collection.

It is the general assumption of algebraic specifications that all operations return result of one sort. Hence, if one wanted to define one operation which does both – modifies the collection and returns a new pointer – one would define an additional sort corresponding to the cartesian product $[C_A \times P_A]$ with appropriate projection functions $\_{}_1$ and $\_{}_2$. Although this might be closer to a programmer's intuition, it would unnecessarily complicate the example. Introducing an operation *Alloc*: $C_A \times A \rightarrow [C_A \times P_A]$ would require, for instance, reformulation of the axiom 6. as $Acc(C, Alloc(C,a)_2) \doteq \perp$ and, in general, cluttering the example with projection functions. We refrain from making this complication because the operations *New* and *Next* play the role of such projections, and one may easily introduce the sort $[C_A \times P_A]$ and an operation *Alloc* at a later stage of development of an implementation from the specification D-POINTER.

It seems that the underspecification D-POINTER puts reasonable restrictions on the possible implementations. In particular, axiom 1. makes the collections abstract – the sequence in which new objects are allocated is not essential. However, consider the following two expressions, where $a \neq b$ are constant accessed values:

1. $New(New(C, a), b) \doteq St(St(C, Next(C), a), Next(St(C, Next(C), a)), b)$
2. $New(New(C, b), a) \doteq St(St(C, Next(C), b), Next(St(C, Next(C), b)), a)$

The former stores first an a (at the *Next* access value) and then a b (again at *Next*), while the latter stores first b and then a. Simple equational reasoning yields

1a. $Acc(\ New(New(C, a), b), Next(C)\ ) \doteq a$

2a. $Acc(\ New(New(C, b), a), Next(C)\ ) \doteq b$

and then, $a \neq b$ implies the antecedent of the following implication (which follows from $Next(C) \doteq Next(C)$ by the congruence axiom for equality) :

$$Acc(New(New(C, a), b), Next(C)) \neq Acc(New(New(C, b), a), Next(C)) \Rightarrow New(New(C, a), b) \neq New(New(C, b), a)$$

To illustrate the problem, we may rephrase this in terms of an imperative program. Assume some Pascal-like language with a command **allocate**(**var** coll: C_A; v: A) **return** P_A. A call to **allocate**(C, v) does three things: (1) returns an access value to a new object in the collection C, (2) stores the value v in that object, and (3) changes C appropriately. (It may be thought of as an operation which in virtue of (2) and (3) implements *New*.)

1. $Z := C$;	2. $Z := C$;
$x :=$ **allocate**(C, a);	$y :=$ **allocate**(C, b);
$y :=$ **allocate**(C, b);	$x :=$ **allocate**(C, a);

(The program variable Z is used to capture the state of the collection before the allocations occur.) In terms of the operations from the specification, we might formulate these as:

1. $Z := C$;	2. $Z := C$;
$x := Next(C)$; $C := St(C, x, a)$;	$y := Next(C)$; $C := St(C, y, b)$;
$y := Next(C)$; $C := St(C, y, b)$;	$x := Next(C)$; $C := St(C, x, a)$;

These two sequences of commands should be equivalent for all purposes. Yet, after executing sequence 1. we have that $x{=}Next(Z)$, but this is not the case after execution of 2. which yields $x{=}Next(St(Z, Next(Z), b))$. The problem is that the function *Next* essentially counts the number of objects that have been stored, identifying uniquely an access value by its place in the sequence of allocations. The language of deterministic operations is *too expressive*, forcing us to consider details beyond what is necessary for our purposes. The problem might be avoided by designing more specific "storage structures" but this would be more suitable for a definition of programming language semantics rather than specification. Alternatively, one might introduce a more abstract notion of "observational equivalence" [Hupbach 1983, Sannella 1985] (e.g., with respect to the operation *Acc*) in order to consider the two programs equivalent.

The example illustrates the general problem of abstraction from some low-level details which are irrelevant at the actual level of description. Underspecification does not allow, in general, for abstraction from conceptually invisible entities (such as hidden state, timing or interaction with a human being). In this case it forces one to distinguish two expressions because of the internal structure of their arguments (the order of allocating pointers) which may correspond to a hidden state. (We discuss this problem in more detail in [Walicki 1994a].)

A simple way out is to avoid identifying different occurrences of $Next(Z)$ as referring always to the same unique value, i.e., to consider it as a nondeterministic operation. The only change to the specification D-POINTER would then be in the last axiom:

POINTER **is**
as in D-POINTER
. . .
$\bot \doteq \bot$
$St(C, x, a) \doteq St(C, x, a)$
$Acc(C, x) \doteq Acc(C, x)$,
. . .
New: 7. $New(C, a) \prec St(C, Next(C), a)$ }
end

Only Acc, St and $\bot$ are deterministic. Since $Next$ has become nondeterministic, the New operation becomes nondeterministic, too. Also, we no longer have the identity $Next(C) \doteq Next(C)$, but only (set) equivalence (by PER): $Next(C) \asymp Next(C)$.[12]

Now, if a is a defined accessed value, it is reasonable to admit the axiom $a \# \bot$. Using that and axioms 4. and 6. we obtain:

$$\begin{array}{ll}
\text{SUB} & \dfrac{Acc(C, Next(C)) \doteq \bot}{} \\
\text{INTR} & \dfrac{Acc(St(C,x,a), Next(St(C,x,a))) \doteq \bot}{} \\
\text{R2} & \dfrac{x \# Next(St(C,x,a)),\ Acc(St(C,x,a),x) \doteq \bot \ ;\ a \# \bot}{} \\
\text{R4} & \dfrac{x \# Next(St(C,x,a)),\ Acc(St(C,x,a),x) \# a \quad ; \quad Acc(St(C,x,a),x) \doteq a}{x \# Next(St(C,x,a))}
\end{array}$$

Using this fact we get the two clauses:

1b. $x \# Next(C) \vee Acc(St(St(C, x, a), Next(St(C, x, a)), b), x) \doteq a$

2b. $x \# Next(C) \vee Acc(St(St(C, x, b), Next(St(C, x, b)), a), x) \doteq b$

which lead to the following conclusion (assuming $a \# b$):

---

[12] $s \asymp t$ is an abbreviation for two inclusions: $s \prec t$ and $t \prec s$.

$$
\text{R2}\;\frac{\text{R2}\;\dfrac{1b \qquad ; \qquad a \# b}{x \# Next(C),\ \ Acc(\,St(St(C,x,a),Next(St(C,x,a)),b),\,x) \# b} \qquad ; \qquad 2b}{\dfrac{x \# Next(C),\ \ Acc(St(St(C,x,a),Next(St(C,x,a)),b),\,x)\ \#\ Acc(St(St(C,x,b),Next(St(C,x,b)),a),\,x)}{x \# Next(C),\ \ Acc(New(St(C,x,a),b),\,x)\ \#\ Acc(New(St(C,x,b),a),\,x)}}
$$

where the last line is obtained by two applications of R3 from axiom 7. However, this does not imply $New(New(C,\ a),\ b)\ \#\ New(New(C,\ b),\ a)$. Observe also, that the conclusion follows only because we have identified the bindings $x \# Next(C)$ from 1b and 2b – x in the conclusion is supposed to be the same when returned by the $Next(C)$ called from $St(C,\ Next(C),\ a)$ and from $St(C,\ Next(C),\ b)$.

Thus we cannot establish that the two collections are disjoint (distinct), but we cannot prove that they are identical either, i.e., POINTER $\nvdash\ New(New(C,\ a),\ b) \doteq New(New(C,\ b),\ a)$. Even the simpler equality $St(C,\ Next(C),\ a) \doteq St(C,\ Next(C),\ a)$ would be too strong (and does not follow from the POINTER axioms) since the two applications of $Next(C)$ may allocate a at different access values.

Adding the axiom $New(New(C,\ a),\ b) \prec New(New(C,\ b),\ a)$ would make the two collections equivalent, in the sense that whatever is the collection resulting from $New(New(C,\ a),\ b)$ could be also obtained from $New(New(C,\ b),\ a)$ and vice versa. We would then have $New(New(C,\ a),\ b) \succ\!\!\!\prec New(New(C,\ b),\ a)$ but still could not prove this formula with $\doteq$ instead of $\succ\!\!\!\prec$.

One might wish to specify an "ideal" access type which, in analogy with the typical specifications of natural numbers, has an unlimited resource of access values.

U-POINTER **is**

as in POINTER

. . .

Next:

8. $Next(St(C,\ x,\ a)) \prec Next(C)$
9. $x \# Next(C),\ x \prec Next(St(C,\ y,\ b)),\ x \doteq y$

end

According to axiom 8. any access value which is not allocated in $St(C,\ x,\ a)$, i.e., which can be returned by $Next(St(C,\ x,\ a))$, is not allocated in C either (and so can be returned by $Next(C)$ as well). 9 is the reverse of that saying that any access value x which is available as a possible result of $Next(C)$ is also available from the extended collection $St(C,\ y,\ b)$ provided that x is different from y.

Interestingly, with these axioms we can establish, for instance, the equivalence $Next(St(C,\ x,\ a)) \succ\!\!\!\prec Next(St(C,\ x,\ b))$ which says that the choice of the $Next$ access value to be allocated does not depend on the accessed values stored in the collection:

$$
\text{R4}\;\frac{\text{R3}\;\dfrac{8. \qquad ; \qquad 9.}{\vdash z \# Next(St(C,x,a)),\ \ z \prec Next(St(C,x,b)),\ z \doteq x} \qquad \dfrac{\vdash x \# Next(St(C,x,a))}{\vdash x \# z,\ z \# Next(St(C,x,a))}\;\text{INTR}}{\dfrac{\vdash z \# Next(St(C,x,a)),\ \ z \prec Next(St(C,x,b))}{\vdash\ \ Next(St(C,x,a)) \prec Next(St(C,x,b))}\;\text{R6}}
$$

The other direction can be now proved by REN.

7. Conclusions and related work

We have defined a formalism suitable for specifying nondeterministic operators. In distinguishing between terms and their applications, we have focused on the observational aspect of programs, on their *computations* (which is what one observes when evaluating a program). We have introduced the calculus NEQ and given the first proof of completeness for multialgebraic semantics.

We have given a procedure for interpreting a specification of a nondeterministic system in terms of an underspecified equational system within a classical conditional equational logic extended with disjunction. The translation yields a specification over a refined language and enables us to define a function oriented, computational semantics. We feel that the latter has a more plausible computational interpretation and reflects our intuition about evaluations of nondeterministic programs. The computational semantics is non-intrusive with respect to the algebraic semantics of deterministic constructs, in the sense that it does not require a change of existing models. This is in contrast to a move to a relation- or set valued model which *does* have an impact on the model of deterministic functions, whereas our understanding of nondeterminism retains a standard function oriented model.

A deterministic interpretation of nondeterministic specifications can be found in some earlier works [Kapur 1980, Subrahmanyam 1981]. They used only simple equations (interpreted as set equalities) and specified a nondeterministic operation f by means of the associated *characteristic predicate* $P_f(x)$ expressing that x is a possible result of f. Translation of an equation $s=t$ was semantically equivalent to our intermediary translation step (def 4.3) for two inclusions, and the characteristic predicates were used for binding results of applications of nondeterministic terms: $\forall x{:}[P_s(x) \Rightarrow \exists y{:}\ (P_t(y) \wedge x{=}y)] \wedge \forall y{:}[P_t(y) \Rightarrow \exists x{:}\ (P_s(x) \wedge x{=}y)]$.

Our derived specifications are based on further Skolemization of such formulae and yield a new semantics where nondeterminism is made explicit in form of the index arguments. Furthermore, [Kapur 1980, Subrahmanyam 1981] did not address the problem of referential opacity of nondeterministic terms at the level of the specification language. Reasoning had to be carried out at the level of the derived deterministic formulae in full first-order logic.

We have also shown that the refinement of the specification language to the level of indexed functions is compatible with the multialgebra semantics. On the one hand, any multimodel has a computational interpretation and, on the other, multisemantics can be used as a correct abstraction tool in cases when such an abstraction from the computational semantics is needed. Both multimodels and functional models of a specification SP satisfy the same formula of $\mathcal{L}_{\text{neq}}$(SP).

Some of the more recent work on algebraic specifications of nondeterminism bears some similarities to the approach presented in this paper.

An elegant construction – unified algebras – has been proposed by Mosses in [Mosses 1989a, Mosses 1989b]. The unified algebras provide quite general and flexible semantic tools. Nondeterministic choice is modeled as a separate sort comprising exactly the possible results. (This is very similar to our nondeterministic constants.) The language allows variables to refer to arbitrary applications of the operations (as in our bindings) in which case they must be interpreted as individuals (unique results) in the semantic domain. In addition, unified algebras represent one of a few algebraic approaches supporting *both* singular and plural (“call-time-choice” and “run-time-choice") semantics of parameter passing. (We have given an account of this distinction in a multi- and poweralgebraic framework in [Walicki 1994a,Walicki 1994b].) In spite of the similarities at the language level, the semantics leading to such flexibility is very different from ours. In particular, it completely changes the traditional notion of a model – also in the cases when specifications involve only deterministic functions.

In [Hußmann 1988, Hußmann 1993] Hußmann uses the standard generalization of algebras to multialgebras as the semantics of nondeterminism. The primitives of his specification language are analogous to the primitives of the language $\mathcal{L}_{neq}$. They comprise inclusions and a determinacy predicate (where the assertion DET(t) corresponds to our $t \doteq t$). The rewriting systems designed for such specifications provide an excellent example of a very constructive approach to reasoning about nondeterminism. Unfortunately, several restrictions must be put on the specifications in order to prove even a limited form of completeness, and the full completeness result is still lacking. (NEQ can be used to show which extensions are needed to Hußmann's calculus and theorem 5.2 yields then the desired completeness theorem [Walicki 1992].) The language and the calculus given by Hußmann can be constructed as special cases of ours. In fact, general clauses in our language and calculus can be restricted, for instance to those having only a single positive formula, without impairing the results obtained for the general case. It is an open problem whether the system implemented by Hußmann [Hußmann 1985] can be accommodated to this more general context. In the recent work [Kriauciukas 1994], an alternative and extended version of NEQ is given which also includes rewriting.

Unlike our approach, both Mosses and Hußmann succeed in construction of initial models. However, it turns out that initial semantics for nondeterministic specifications tend to introduce redundant elements, thus violating the "no junk" dictum of the original initial approach [Walicki 1993b]. In our formalism, disjunction provides sufficient means for eliminating "junk" and leads to more intuitive semantics.

Acknowledgment

The paper was significantly improved thanks to the suggestions of the anonymous referees. We are particularly grateful to one referee whose proposal to recast the original sequent based presentation in terms of clauses made both the logic and proofs simpler and more elegant.

The first author thanks Norwegian Research Council for the financial support during the work on this paper.

REFERENCES

[Aceto 1986] Aceto, L., DeNicola, R., Fantechi, A., "Testing Equivalencies for Event Structures", in *Mathematical models for the semantics of parallelism*, LNCS, vol. 280, Springer, 1986.

[Bauer 1985] Bauer, F., *The Munich Project CIP: The Wide Spectrum Language CIP*, LNCS vol. 183, Springer, 1985.

[Bergstra 1986] Bergstra, J.A., Klop, J.W., "Algebra of communicating processes", *Proceedings of CWI Symposium on Mathematics and CS*, 89 - 138, Oct. 6-7 1986.

[Birkhoff 1935] Birkhoff, G., "On the Structure of Abstract Algebras.", *Proc. Cambridge Philos. Soc*, vol. 31, 433-454, 1935.

[Chang 1977] Chang, C.C., Keisler, H.J., *Model Theory*, Amsterdam, North-Holand, 1977.

[Dershowitz 1990] Dershowitz, N., Jouannaud, J.P., "Rewriting Systems", in *Handbook of Theoretical Computer Science*, vol. B, The MIT Press, 1990.

[Dijkstra 1976] Dijkstra, E.W., *A Discipline of Programming*, Prentice-Hall, 1976.

[Ehrig 1985] Ehrig, H., Mahr, B., *Fundamentals of Algebraic Specification*, vol. 1, Springer, 1985.

[Goguen 1981] Goguen, J.A., Meseguer, J., "Completeness of Many-Sorted Equational Logic", *SIGPLAN Notices*, vol. 16, no. 7, 1981.

[Goguen 1982] Goguen, J.A., Meseguer, J., "Universal realization, persistent interconnection and implementation of abstract modules", in *Proc., 9th Int. Coll. on Automata, Languages and Programming*, LNCS, vol. 140, Springer, 1982.

[Goguen 1987] Goguen, J.A., Meseguer, J., "Remarks on Remarks on Many-Sorted Equational Logic", *SIGPLAN Notices*, vol. 22, no. 4, 41-48, April 1987.

[Goguen 1988] Goguen, J.A., *What is unification? A categorical view of substitution, equation, and solution*, Tech. Rep. CSLI-88-124, Center for Study of Languages and Information, 1988.

[Hennessy 1980] Hennessy, M., Milner, R., "On Observing Nondeterminism and Concurrency", in *Automata, Languages and Programming*, LNCS, vol. 85, Springer, 1980.

[Hennessy 1988] Hennessy, M., "Observing Processes", in *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, LNCS, vol. 354, Springer, 1988.

[Hesselink 1988] Hesselink, W.H., "A Mathematical Approach to Nondeterminism in Data Types", *ACM: Transactions on Programming Languages and Systems*, vol. 10, 1988.

[Hoare 1973] Hoare, C.A.R., Wirth, N., "An axiomatic definition of the programming language PASCAL", *Acta Informatica*, vol. 2, 335-355, 1973.

[Hoare 1985] Hoare, C.A.R., *Communicating Sequential Processes*, Prentice-Hall International Ltd., 1985.

[Hußmann 1985] Hußmann, H., *Rapid prototyping for algebraic specifications – RAP system user's manual*, Tech. Rep. MIP-8504, Universität Passau, 1985.

[Hußmann 1988] Hußmann, H., "Nondeterministic algebraic specifications and nonconfluent term rewriting", in *Algebraic and Logic Programming*, LNCS, vol. 343, Springer, 1988.

[Hußmann 1990] Hußmann, H., *Nondeterministic Algebraic Specifications*, Ph.D. thesis, Fakultät für Mathematik und Informatik, Universität Passau, 1990.

[Hußmann 1993] Hußmann, H., *Nondeterminism in Algebraic Specifications and Algebraic Programs*, Birkhäuser, 1993.

[Huet 1980] Huet, G., Oppen, D., "Equations and Rewrite Rules: A Survey", in *Formal Language Theory: Perspectives and Open Problems*, Academic Press, 1980.

[Hupbach 1983] Hupbach, U.L., Reichel, H., "On Behavioural Equivalence of Data Types", *EIK, Journal of Information Processing and Cybernetics*, vol. 19, no. 6, 297-305, 1983.

[Kahn 1974] Kahn, G., "The semantics of a simple language for parallel programming", in Proceedings of *Proceedings of IFIP 1974,* North-Holland, 1974.

[Kaplan 1987] Kaplan, S., "Conditional Rewriting", in *Conditional Term Rewriting Systems*, LNCS, vol. 308, Springer, 1987.

[Kaplan 1988] Kaplan, S., "Rewriting with a Nondeterministic Choice Operator", *Theoretical Computer Science*, vol. 56, 37-57, 1988.

[Kapur 1980] Kapur, D., *Towards a theory of abstract data types*, Ph.D. thesis, Laboratory for CS, MIT, 1980.

[Kriauciukas 1994] Kriauciukas, V., Walicki, M., *Reasoning and Rewriting with Set-Relations I: Ground Case*, CSL'94; also Tech. Rep. 92, University of Bergen, Dept. of Informatics, 1994.

[Lyndon 1959] Lyndon, R.C., "Properties Preserved under Homomorphism", *Pacific Journal of Mathematics*, vol. 9, 1959.

[Maibaum 1977] Maibaum, T.S.E., *The Semantics of Nondeterminism*, Tech. Rep. CS-77-30, University of Waterloo, Ontario, Canada, December 1977.

[Meldal 1987] Meldal, S., *Allocations of Objects Considered as Nondeterministic Expressions*, Tech. Rep. CSL-TR-87-335, CSL Stanford University, 1987.

[Meldal 1989] Meldal, S., "An Abstract Axiomatization of Pointer Types", in Proceedings of *The 22nd Annual Hawaii International Conference on System Sciences,* IEEE Computer Society Press, 1989.

[Meseguer 1990] Meseguer, J., "Conditional Rewriting Logic: Deduction, Models and Concurrency", in Proceedings of *CTRS'90,* LNCS vol. 516, 1990.

[Meseguer 1992] Meseguer, J., "Conditional rewriting logic as a unified model of concurrency", *TCS*, no. 96, 73-155, 1992.

[Milner 1980] Milner, R., *Calculi for Communicating Systems*, LNCS vol. 92, Springer, 1980.

[Mosses 1989a] Mosses, P.D., "Unified Algebras and Institutions", in Proceedings of *LICS'89, Fourth Annual Symposium on Logic in Computer Science,* 1989.

[Mosses 1989b] Mosses, P.D., "Unified Algebras and Action Semantics", in *STACS'89*, LNCS, vol. 349, Springer, 1989.

[Nipkow 1986] Nipkow, T., "Non-deterministic Data Types: Models and Implementations", *Acta Informatica*, vol. 22, 629 - 661, 1986.

[Nipkow 1987] Nipkow, T., “Observing nondeterministic data types”, in *Recent Trends in Data Type Specification*, LNCS, vol. 332, Springer, 1987.

[O’Donell 1977] O'Donnell, M., “Computing in Systems Described by Equations”, LNCS, vol. 58, Springer, 1977.

[Pickett 1967] Pickett, H.E., “Homomorphisms and subalgebras of multialgebras”, *Pacific Journal of Mathematics*, vol. 21, 327-342, 1967.

[Qian 1993] Qian, X., Goldberg, A., “Referential Opacity In Nondeterministic Data Refinement”, *ACM LoPLaS*, vol. 2, no. 1-4, 233-241, 1993.

[Sannella 1985] Sannella, D., Tarlecki, A., “On Observational Equivalence and Algebraic Specification”, in Proceedings of *Joint Conference on Theory and Practice of Software Development, TAPSOFT,* LNCS vol. 185, Springer, 1985.

[Subrahmanyam 1981] Subrahmanyam, P.A., “Nondeterminism in Abstract Data Types”, in *Automata, Languages and Programming*, LNCS, vol. 115, Springer, 1981.

[Volger 1989a] Volger, H., “The semantics of disjunctive deductive databases”, in *CSL’89*, LNCS, vol. 440, Springer, 1989.

[Volger 1989b] Volger, H., *The Semantics of Disjunctive Deductive Databases*, Tech. Rep. MIP-8931, Fakultät für Mathematik und Informatik, Universität Passau, October 1989.

[Walicki 1992] Walicki, M., *Calculii for nondeterministic specifications: three completeness results*, Tech. Rep. 75, Institutt for Informatikk, Universitetet i Bergen, December 1992.

[Walicki 1993a] Walicki, M., *Algebraic Specifications of Nondeterminism*, Ph.D. thesis, University of Bergen, Department of Informatics, 1993.

[Walicki 1993b] Walicki, M., Meldal, S., “Initiality+Nondeterminism => Junk”, *Proc. of NIK'93*, 1993.

[Walicki 1994a] Walicki, M., Meldal, S., “Multialgebras, Power Algebras and Complete Calculi of Identities and Inclusions”, to appear in *Recent Trends in Data Type Specifications,* LNCS 1994.

[Walicki 1994b] Walicki, M., *Singular and Plural Nondeterministic Parameters: Multialgebras, Power Algebras and Complete Reasoning Systems*, Tech. Rep. 96, Institutt for Informatikk, Universitetet i Bergen, 1994.

[Winskel 1988] Winskel, G., “An introduction to event structures”, LNCS, vol. 354, Springer, 1988.

[Wirsing 1989] Wirsing, M., *Algebraic Specification*, Tech. Rep. MIP-8914, Universität Passau, 1989.

[Wirsing 1990] Wirsing, M., “Algebraic Specification”, in *Handbook of Theoretical Computer Science*, vol. B, The MIT Press, 1990.

[Wolter 1992] Wolter, U., Löwe, M., “Beyond Conditional Equations”, in *CAAP’92*, LNCS, vol. 581, Springer, 1992.

APPENDIX A. The completeness proof (theorem 5.2).

This is the proof of completeness wrt. computational semantics (as defined in section 4.2), both wrt. $\exists$- and $\mathcal{C}$-validity. (The parts concerning the latter are put into square brackets [$_{\mathcal{C}}$ ...].)

A Henkin-style argument applied in this proof requires that we show several counterparts of the standard results needed in such proofs. The general strategy requires showing that any consistent specification has a model (computation) and that this, in turn, implies that clauses valid in all computations are provable. To establish the first implication we complete a given theory by enumerating all formulae and adding a formula or its negation depending on which one is consistent with the theory. Since $\#$ provides only a limited form of negation, we introduce a general one by defining, for a given specification SP and clause C, a specification whose computations cannot satisfy C. Suppose that we want to ensure validity of the *negation* of

$$t_1 \# s_1, \ldots, t_l \# s_l, p_1 \doteq r_1, \ldots, p_m \doteq r_m, u_1 \prec v_1, \ldots, u_n \prec v_n \qquad (C)$$

Adding C to the specification would amount to making valid the formula (all the a, b, c, d, e, f's are index variables, $\ddot{x}$ represents the variables of (C))

$$\forall \ddot{x}, a_1, b_1, c_1, d_1, e_1, \ldots, \exists f_1, \ldots, \exists f_m \bullet \left(\bigvee_{i=1}^{l} t_i(a_i) \neq s_i(b_i) \vee \bigvee_{j=1}^{n} p_j(c_j) = r_j(d_j) \vee \bigvee_{k=1}^{m} u_k(e_k) = v_k(f_k) \right) \qquad (C1)$$

Validity of its first order negation

$$\exists \ddot{x}, a_1, b_1, c_1, d_1, e_1, \ldots, \forall f_1, \ldots, \forall f_m \bullet \left(\bigwedge_{i=1}^{l} t_i(a_i) = s_i(b_i) \wedge \bigwedge_{j=1}^{n} p_j(c_j) \neq r_j(d_j) \wedge \bigwedge_{k=1}^{m} u_k(e_k) \neq v_k(f_k) \right) \qquad (\neg C1)$$

can be obtained by ensuring the existence of appropriate elements (terms). So, for every variable x we introduce a new deterministic constant c_x and substitute c's for the corresponding variables in C obtaining a ground clause $C(\Gamma)$. Furthermore, we introduce a new (deterministic) constant for every existential quantifier and add equations making them applications of corresponding terms. Finally, we add literals to identify and distinguish appropriate terms.

Definition 8.1 (Negation). The negation $\sim C$, of a clause C is the following set of literals (all t, s, p, r are after substitution of new constants for the variables in C):

1. $c_x \doteq c_x$ for every variable $x \in \mathcal{V}[C]$

For every $t_i \# s_i$ (except bindings) and $p_m \prec r_m$ add one, and for every $p_j \doteq r_j$ two new constants:

2. $c_i \doteq c_i$ $\quad c_m \doteq c_m$ $\quad c_j \doteq c_j$ $\quad c_{j1} \doteq c_{j1}$

For every binding $x \# t$:	3. $c_x \prec t$			
For every other $t_i \# s_i$:	4. $c_i \prec t_i$	$c_i \prec s_i$		
For every $p_j \doteq r_j$:	5. $c_j \prec p_j$	$c_{j1} \prec r_j$	5i.	$c_{j1} \# c_j$
For every $p_m \prec r_m$:	6. $c_m \prec p_m$		6i.	$c_m \# r_m$

□

1 and 2 make c's independent from the indices, i.e. $c(i)$=$c(j)$. 3, 4, 5 and 6 correspond then to Skolemization of the corresponding existential quantifiers identifying c's with the terms on the right-hand side of $\prec$ (each with a new indexing term). 4 and 5 make each of $\exists\, a_i, b_i.\ t_i(a_i)$=$s_i(b_i)$ and

$\exists e_j, d_j.\ p_j(e_j)\neq r_j(d_j)$ true and 6i, turning $\exists\, d_m$ of C into $\forall\, d_m$, makes $\exists e_m \forall\, d_m.p_m(e_m)\neq r_m(d_m)$ true.

Skipping indices at the deterministic constants, we can write the set of formulae made valid by $\sim C$:

3. $\exists\, k.c_x{=}t(k)\ \wedge$
4. $\exists\, a_i.c_i{=}t_i(a_i)\ \wedge\ \exists b_i.c_i{=}s_i(b_i)\wedge$
5. $\exists\, e_j.c_j{=}p_j(e_j)\ \wedge\ \exists d_j.c_{j1}{=}r_j(d_j)\wedge c_{j1}\neq c_j\ \wedge$
6. $\exists\, e_m.c_m{=}p_m(e_m)\wedge \forall\, d_m.c_m\neq r_m(d_m)$

The signature of SP; $\sim C$ is the signature of SP extended with the new constants. Satisfiability of clauses over the extended signature in the CStr(SP; $\sim C$) is defined in the same way (definition 4.6 and 4.7). To be able to speak about satisfiability of $\sim C$ in SP-structures we need to extend the definition of satisfiability as follows:

Definition 8.2. Let $C\in\mathcal{L}_{\text{neq}}(\text{SP})$, and M$\in$CStr(SP). M$\models_\exists \sim C$ iff *there exists an interpretation* of the constants introduced in $\sim C$ making the formulae 3.-6. above true, i.e., iff M satisfies the formula:

$$\exists x, a_i, b_i, e_i, d_1..e_m \forall\, d_m \bullet \bigwedge_{i=1}^{n} t_i(a_i){=}s_i(b_i)\ \wedge\ \bigwedge_{j=1}^{m} p_j(e_j)\neq r_j(d_j)$$

□

This means that M $\exists$-satisfies $\sim C$ iff it satisfies the first order negation of C and so we have:

Lemma 8.3. For every $M\in$CStr(SP), $C\in\mathcal{L}_{\text{neq}}$(SP) $\bullet$ $M\not\models_\exists C$ iff $M\models_\exists \sim C$.

□

[$_{\mathcal{C}}$ For $\mathcal{C}$–satisfiability of $\sim C$, we will require that there exist an interpretation of the new constants which makes all inclusions of $\sim C$ $\mathcal{C}$–satisfied, i.e., that $\exists a_i, b_i, e_i, d_1..e_m$ from the definition 8.2 are replaced by the requirement of existence of the respective index terms. This will imply first order negation of C and thus we have

Lemma 8.4. For every $M\in$CStr(SP), $C\in\mathcal{L}_{\text{neq}}$(SP) $\bullet$ $M\not\models_{\mathcal{C}} C$ or $M\not\models_{\mathcal{C}} \sim C$.

□

This is a weaker property than the one from lemma 8.3, but it suffices for the constructions needed in the proof of completeness with respect to $\models_{\mathcal{C}}$.]

We note a couple of useful facts about negation.

Lemma 8.5.

1. For any C $\bullet$ C; $\sim C \vdash \emptyset$.
2. If $\Pi \vdash t\doteq t$ then Π; C; $\sim(C^x_t) \vdash \emptyset$.
3. C; $\sim$(C, D) $\vdash \emptyset$

Proof:

1. Let C be $x \# t$, $t_1 \# s_1$, ..., $t_n \# s_n$, $p_1 \doteq r_1$, ...$p_m \prec r_m$ and $\sim C$ be as in the definition 8.1. $C(\Gamma)$ is C with new constants substituted for all variables (1. of def. 8.1). C; $\sim C \vdash C(\Gamma)$ by SUB. Clauses 3 from def. 8.1 allow us to eliminate (CUT) the parts of $C(\Gamma)$ corresponding to bindings from C, i.e.

$$C;\ \sim C \vdash\ t_1 \# s_1, \ldots, t_n \# s_n, p_1 \doteq r_1, \ldots p_m \prec r_m$$

where terms denote the terms from $C(\Gamma)$ (after substitution of constants). Using INTR for each $t_i \# s_i$, and then clauses 2 and 4 from definition 8.1, we can eliminate (CUT) $t_i \# s_i$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\ldots, t_i \# s_i \ldots}{\ldots, x_i \# s_i,\ x_i \# t_i \ldots\quad ;\quad c_i \doteq c_i}\text{INTR}}{\ldots, c_i \# s_i,\ c_i \# t_i \ldots\quad ;\qquad c_i \prec s_i}\text{SUB}}{\ldots, c_i \# t_i \ldots\quad ;\qquad\qquad c_i \prec t_i}\text{R4}}{\ldots\ldots}\text{R4}$$

An analogous procedure leads to the elimination of all $\prec$ and $\doteq$.

2. The assumption $\Pi \vdash t \doteq t$ gives $\Pi\ ; C \vdash C^{x}_{t}$. The conclusion follows from 1.
3. This follows from 1. and the fact that negation of the extended clause C, D contains all elements of the negation of the sub-clause C.

□

We now proceed to collect the pieces needed for the proof of the main theorem. The first one will be the deduction theorem (8.9) stating that inconsistency of $\Pi; \sim C$ implies provability of C from Π. To show that, we need some technical preliminaries.

Definition 8.6. Given a set of clauses Π and a ground literal a, the *simple negation* of a wrt. Π, $\underline{a}$, is as follows:

$$\begin{array}{cc} a & \underline{a} \\ s \prec t & s \# t \\ s \# t & s \prec t \end{array} \Bigg\} \text{ if } \Pi \vdash s \doteq s \qquad\qquad \begin{array}{cc} a & \underline{a} \\ s \doteq t & s \# t \\ s \# t & s \doteq t \end{array} \Bigg\} \text{ if } \Pi \vdash s \doteq s \text{ and } \Pi \vdash t \doteq t$$

If a clause has a simple negation, we say that it is *invertible* (wrt. Π).

□

Thus, a clause is invertible only if at least one of its terms is deterministic. The fact which will be of importance in the following is that all clauses introduced in the definition 8.1 of negation are invertible. It is easy to see that for an invertible a, the clause $\{a, \underline{a}\}$ is derivable from R1a or R1b by SUB.

Theorem 8.7. Let Π be a set of clauses, G a clause, and $\Gamma = \{a_1; \ldots; a_n\}$ be a set of invertible clauses (wrt. Π). Then

$$\Pi; \Gamma \vdash G \text{ iff } \Pi \vdash G,\ \underline{\Gamma}$$

where $\underline{\Gamma}$ is the clause $\{\underline{a}_1, \ldots, \underline{a}_n\}$.

Proof:

(⇐) The implication is trivial by n applications of the cut rule.

(⇒) By induction on the length l of the first proof (where applications of the "axioms" R1 are counted as proofs of length 0).

<u>l=0:</u>

If G is an axiom of Π then the conclusion is obtained by R5. If G is one of a_i then, since all a_i are invertible, $\{a_i, \underline{a}_i\}$ is derivable and conclusion follows by R5.

R1. If Π; Γ proves G by R1 then Π proves the same clause by R1 and the conclusion follows by R5.

<u>l+1:</u>

In all cases the same rule as used in the proof from Π; Γ gives the proof from Π.

R2. By induction hypothesis, there exist proofs from Π of clauses corresponding to the assumptions of the rule: $\Pi \vdash C^x_t, \underline{\Gamma}$ and $\Pi \vdash D, s \doteq t, \underline{\Gamma}$. Applying R2 to these two clauses yields the proof of $C^x_s, D, \underline{\Gamma}$ from Π.

R3–R5. Analogous application of induction hypothesis as for R2.

R6. By induction hypothesis, there is a proof $\Pi \vdash C, x \# t, \underline{\Gamma}$. Since $\underline{\Gamma}$ is ground x does not occur in it, so we can apply R6 in the same fashion as in the proof from Π; Γ.

□

Lemma 8.8. Let Π be a set of clauses over Σ and Γ be a set of constants not belonging to Σ. Let Π^Γ denote the theory (over $\Sigma \cup \Gamma$) obtained by adding, for each $c \in \Gamma$, the axiom $c \doteq c$. Let C be a clause over Σ, and $C(\Gamma)$ denote the clause obtained by substituting for every variable in C a new $c \in \Gamma$. Then

$$\Pi \vdash C \text{ iff } \Pi^\Gamma \vdash C(\Gamma).$$

Proof:

($\Rightarrow$) The implication is trivial by repeated application of SUB.

($\Leftarrow$) The converse is proved by induction on the proof of $C(\Gamma)$.

<u>l=0:</u>

If $C(\Gamma)$ is an axiom of Π^Γ then it is either an axiom of Π or it is $c \doteq c$. PER gives the conclusion in the latter case.

R1b. If $t \in W_{\Sigma, X}$ then Π proves the same clause by R1b. Otherwise, some Γ-constants occur in t, but then Π proves (by R1b) the corresponding clause with new variables instead of all Γ-constants from t.

<u>l+1:</u>

R2. By induction hypothesis, Π proves the clauses corresponding to the assumptions of this rule with variables instead of Γ-constants. Then Π will prove C by R2.

The same argument holds for all the other rules. In R6 we can eliminate x in the proof from Π in the same way as it is done in the proof from Π^Γ because all the variables introduced for the constants in the assumptions are different from x.

□

Theorem 8.9 (Deduction theorem). For Π a set of clauses and C a clause:

$$\Pi; {\sim}C \vdash \emptyset \text{ implies } \Pi \vdash C.$$

Proof:

Let C and ${\sim}C$ be as in definition 8.1, i.e., C is of the general form $x \# t$, $s_i \# t_i$, $p_j \doteq r_j$, $p_m \prec r_m$. Then Π; ${\sim}C$ is Π^Γ; Δ, where Δ is the set of simple clauses 3-6i from definition 8.1 and Γ is the set of all constants introduced in the negation. Every $a \in \Delta$ is invertible so, by theorem 8.7, there is a proof

$$\Pi^{\Gamma} \vdash\ c_x \# t,\ c_i \# s_i,\ c_i \# t_i,\ c_j \# p_j,\ c_{j1} \# r_j,\ c_m \# p_m\ ,\ c_j \doteq c_{j1},\ c_m \prec r_m$$

We can now eliminate all c_i's. By lemma 8.8, we obtain

$$\Pi^{\Theta} \vdash\ c_x \# t,\ x_i \# s_i,\ x_i \# t_i,\ x_j \# p_j,\ x_{j1} \# r_j,\ x_m \# p_m\ ,\ x_j \doteq x_{j1},\ x_m \prec r_m$$

where all x_k are fresh variables, and $\Theta \subset \Gamma$ is the set of constants corresponding to the original variables in C. Observe that, by definition 8.1 of negation, each $c_k \in \Gamma\text{-}\Theta$ (and hence each x_k) occurs only twice. So, we can use R6 for all variables x_k (we give an example for x_j, x_{j1}) :

$$\cfrac{\cfrac{\ldots, x_j \# p_j\ ,\ x_{j1} \# r_j\ ,\ x_j \doteq x_{j1} \ldots}{\ldots, x_{j1} \# r_j\ ,\ p_j \doteq x_{j1} \ldots}\ \text{R6}}{\ldots,\ p_j \doteq r_j \ldots}\ \text{R6}$$

In this way we obtain a proof $\Pi^{\Theta} \vdash\ c_x \# t,\ s_i \# t_i,\ p_j \doteq r_j,\ p_m \prec r_m$

Now all s, t, p, r are ground and, possibly, contain occurrences of c_x's. The final application of lemma 8.8 leads us to the desired conclusion that there is a proof of the clause $x \# t,\ s_i \# t_i,\ p_j \doteq r_j,\ p_m \prec r_m$ from Π.

□

We can now prove

Lemma 8.10. The implication 1 $\Rightarrow$ 2 holds:

1. For any Π : $\Pi \nvdash \emptyset$ implies $\mathrm{Comp}(\Pi) \neq \emptyset$
2. For any Π : $\Pi \models_{\exists} C$ $[_{\mathcal{C}}\ \Pi \models_{c} C\ ]$ implies $\Pi \vdash C$

Proof :

Suppose that $\Pi \models_{\exists} C$ $[_{\mathcal{C}}\ \Pi \models_{c} C\ ]$ (i.e., every computation of Π satisfies C). Then, by lemma 8.3, $\Pi \not\models_{\exists} {\sim}C$ $[_{\mathcal{C}}$ 8.4, $\Pi \not\models_{c} {\sim}C$] i.e. no computation of Π satisfies ${\sim}C$. So Π; ${\sim}C$ has no computation and, by the assumption 1, is inconsistent, i.e., $\Pi ; {\sim}C \vdash \emptyset$. Then $\Pi \vdash C$ by theorem 8.9.

□

What remains is to show that a consistent theory has a computation. By theorem 4.10 it suffices to show the existence of a multimodel. So let Π be a consistent set of clauses over signature Σ. Construct Π_{ω} as follows:

1. Let Γ be a (countable) set of new constants not occurring in Σ. Let $\Pi^{\Gamma} = \Pi \cup \{c \doteq c \mid c \in \Gamma\}$. By lemma 8.8, for any clause C : $\Pi^{\Gamma} \vdash C(\Gamma)$ iff $\Pi \vdash C$. In particular, Π^{Γ} is consistent.
2. Fix an enumeration C_1, C_2, ... of all clauses over Σ, Γ, X (modulo bijective renaming of variables from the countable set X) and let

   $\Pi_0 = \Pi^{\Gamma}$

   $$\Pi_{i+1} = \begin{cases} \Pi_i\ ;\ C_i & \text{if it is consistent} \\ \Pi_i\ ;\ {\sim}C_i & \text{otherwise} \end{cases}$$

   When adding ${\sim}C$ introduce always the first constants from Γ not used so far.
3. $\Pi_{\omega} = \bigcup_{i<\omega} \Pi_i$

Lemma 8.11. If Π is consistent then Π_{ω} is consistent.

Proof:

If Π_{i+1} is inconsistent then $\Pi_{i+1}=\Pi_i; \sim C_i$ and $\Pi_i; \sim C_i \vdash \emptyset$. So, by theorem 8.9, $\Pi_i \vdash C_i$. But Π_i; C_i is inconsistent, so Π_i must be inconsistent. Thus, for every i, Π_i is consistent. Obviously, the compactness theorem holds, i.e. a theory Π is inconsistent iff there is a finite subset $\Pi' \subset \Pi$ which is inconsistent. Since every finite subset of Π_ω is consistent Π_ω is consistent.

□

We have also the following:

Lemma 8.12. Let C be an arbitrary clause over Σ, Γ, X:

1. $C \notin \Pi_\omega$ iff $\sim C \subset \Pi_\omega$.
2. a) $C \in \Pi_\omega$ $(x \in \mathcal{V}[C])$ iff b) $\forall t$: $t \doteq t \in \Pi_\omega \Rightarrow C^x_t \in \Pi_\omega$
   iff c) $\forall$ground t: $t \doteq t \in \Pi_\omega \Rightarrow C^x_t \in \Pi_\omega$
3. For every ground term t : $t \doteq t \in \Pi_\omega$ or $\exists d$ ($d \doteq d \in \Pi_\omega$ and $d \prec t \in \Pi_\omega$).

Proof:

1. is obvious by the construction of Π_ω.
2. follows from lemma 8.5. (a⇒ b) If $C \in \Pi_\omega$ then, for all deterministic t, $\Pi_\omega \vdash C^x_t$ by SUB. If there exists t such that $C^x_t \notin \Pi_\omega$ then, by 1, $\sim (C^x_t) \subset \Pi_\omega$, and by lemma 8.5 $\Pi_\omega \vdash \emptyset$. But Π_ω is consistent by lemma 8.11. (b⇒ c) Obvious. (a⇐c) If $C \notin \Pi_\omega$ then $\sim C \subset \Pi_\omega$. Let c be the new constant introduced for variable x in the negation of C. $C^x_c \notin \Pi_\omega$ because otherwise $\sim C$ and C^x_c would both be in Π_ω and, by 8.5, we could prove $\emptyset$ from Π_ω.
3. Assume that $t \doteq t \notin \Pi_\omega$ and that for all d (not necessarily ground) such that $d \doteq d \in \Pi_\omega$: $d \prec$ t $\notin \Pi_\omega$. Then, for all d: $d \# t \in \Pi_\omega$, for if not, then $\sim (d \# t) \in \Pi_\omega$ for some d and this negation would give as a d' such that $d' \prec t \in \Pi_\omega$. But then 2 implies that $x \# t \in \Pi_\omega$. This however would mean that Π_ω is inconsistent by a single application of R6.

□

Lemma 8.13. If Π is consistent then Π_ω has a multimodel which is a multimodel of Π.

Proof:

We construct a multimodel M with the carrier containing the equivalence classes $[t] = \{s \mid t \doteq s \in \Pi_\omega\}$ for every ground term t such that $t \doteq t \in \Pi_\omega$. For every other ground term there is a (deterministic) term s such that $s \prec t \in \Pi_\omega$ (8.12.3), so we let the set interpretation of such terms be the set of such s. I.e., we define the set interpretation of ground terms $\langle \_ \rangle : W_\Sigma \rightarrow \mathcal{P}^+(|M|)$

$$\langle t \rangle = \begin{cases} \{[t]\} & \text{if } t \doteq t \in \Pi_\varpi \\ \bigcup \langle s \rangle \bullet s \prec t \in \Pi_\varpi & \text{otherwise} \end{cases} \qquad (*)$$

The operations are defined on the deterministic terms as $f^M([t]) = \langle f(t) \rangle$, and on other terms by pointwise extension $f(t)^M = f^M(\langle t \rangle) = \bigcup_{[s] \in \langle t \rangle} f^M([s])$. This is a correct definition, i.e., we have that $f(t)^M$ coincides with $\langle f(t) \rangle$ as defined by (*):

1) $f(t)^{\mathrm{M}} = \bigcup_{[s]\in\langle t\rangle} f^{\mathrm{M}}([s]) \subseteq \langle f(t)\rangle$

$$[s]\in\langle t\rangle \Rightarrow s\prec t\in\Pi_\omega \Rightarrow f(s)\prec f(t)\in\Pi_\omega \Rightarrow \langle f(s)\rangle \subseteq \langle f(t)\rangle \Rightarrow f^{\mathrm{M}}([s]) \subseteq \langle f(t)\rangle$$

2) $\langle f(t)\rangle \subseteq f(t)^{\mathrm{M}} = \bigcup_{[s]\in\langle t\rangle} f^{\mathrm{M}}([s])$

Suppose the opposite, i.e., that there is an $[m]$ such that a) $[m]\in\langle f(t)\rangle$ but b) for any $[s]\in\langle t\rangle$, $[m]\notin f^{\mathrm{M}}([s])$. From a) we have that $m\doteq m$ and $m\prec f(t)\in\Pi_\omega$, and from b) that for any s such that $s\doteq s\in\Pi_\omega$: $s\prec t\notin\Pi_\omega$ or $m\prec f(s)\notin\Pi_\omega$. But this implies that the clause $G=\{x\#t, m\#f(x)\}\in\Pi_\omega$ – otherwise its negation, i.e., $c\prec t$, $m\prec f(c)$ and $c\doteq c$ would be in Π_ω, contradicting b). But a single application of R6 to G gives $m\#f(t)\in\Pi_\omega$ contradicting the assumption that $m\prec f(t)\in\Pi_\omega$.

Thus M is a multialgebra and for every assignment β: $X\to|\mathrm{M}|$, there is a function $\beta[\_]$: $W_{\Sigma,X}\to\mathcal{P}^+(|\mathrm{M}|)$ defined as: $\beta[t(x)] = \langle t(\beta(x))\rangle = t^{\mathrm{M}}(\beta(x))$.

To see that M $\mathcal{P}$-satisfies the axioms of Π_ω, first observe that for all ground terms t, s:

i) $\mathrm{M}\models_{\mathcal{P}} t\doteq s$ iff $\{[t]\}\equiv\{[s]\}$ iff $[t]=[s]$ iff $t\doteq s\in\Pi_\omega$

ii) $\mathrm{M}\models_{\mathcal{P}} t\prec s$ iff $\langle t\rangle\subseteq\langle s\rangle$ iff $\forall p \bullet [p]\in\langle t\rangle \Rightarrow [p]\in\langle s\rangle$ iff

$$\forall p \bullet p\doteq p\in\Pi_\omega \Rightarrow (p\prec t\in\Pi_\omega \Rightarrow p\prec s\in\Pi_\omega) \text{ iff } t\prec s\in\Pi_\omega$$

iii) $\mathrm{M}\models_{\mathcal{P}} t\# s$ iff $\langle t\rangle\cap\langle s\rangle=\emptyset$ iff $\forall p \bullet [p]\in\langle t\rangle \Rightarrow [p]\notin\langle s\rangle$ iff

$$\forall p \bullet p\prec t\in\Pi_\omega \Rightarrow p\prec s\notin\Pi_\omega \text{ iff } t\# s\in\Pi_\omega$$

The last iff in ii) holds because: ($\Leftarrow$) If $t\prec s\in\Pi_\omega$ then for every p, $p\prec t\in\Pi_\omega \Rightarrow p\prec s\in\Pi_\omega$. ($\Rightarrow$) If $t\prec s\notin\Pi_\omega$ then its negation is in Π_ω and gives us a p contradicting $\forall p$. The last iff in iii) holds by a similar argument. ($\Rightarrow$) If $t\# s\notin\Pi_\omega$ then its negation is in Π_ω and gives us a p. ($\Leftarrow$) If $t\# s\in\Pi_\omega$ then no such p can exists unless Π_ω is inconsistent.

By lemma 8.12.2 the above observation can be generalized to non-ground literals.

Now, suppose that M does not satisfy an axiom C: $a_1, \ldots, a_n$ in Π_ω, i.e., there exists an assignment β:$X\to|\mathrm{M}|$ such that all a's are false in M. Let β' be a syntactic assignment reflecting β, i.e., $\beta'(x)=f$ if $\beta(x)=[f]$, and let $\beta'(a_i)=a1_i$. By lemma 8.12.2 all deterministic instances of C are in Π_ω, so $\beta'(C)\in\Pi_\omega$.

If $\mathrm{M}\not\models_{\mathcal{P}} C$ under β then $\mathrm{M}\not\models_{\mathcal{P}} a1_i$, for every a_i. Then i-iii) give the first, and the construction of Π_ω the second implication:

$$\mathrm{M}\not\models_{\mathcal{P}} a1_i \Rightarrow a1_i\notin\Pi_\omega \Rightarrow \sim a1_i\subset\Pi_\omega$$

So Π_ω contains the ground clause $\beta'(C)$ together with the negation of its every component. This means that we can eliminate all components from $\beta'(C)$ and obtain $\emptyset$ from Π_ω. E.g., if $a1$ is $s\doteq t$, then applying INTR twice we derive

$$\cfrac{\cfrac{p\doteq r, \ldots, a1_n}{x_i\# p, x_i\doteq r, \ldots, a1_n}}{x_i\# p,\ x_{i1}\# r, x_i\doteq x_{i1}, \ldots, a1_n}$$

The negation of $a1$ contains $c_i\prec p$, $c_{i1}\prec r$, and $c_i\# c_{i1}$. Substituting c_i's for x_i's enables us to remove $a1$. Similarly, we can eliminate all other literals from $\beta'(C)$ deriving the empty clause, which contradicts the assumption that Π_ω is consistent. Hence M is a multimodel of Π_ω. Since $\Pi\subset\Pi_\omega$ M is a multimodel of Π.

□

Lemmas 8.11 and 8.13 yield immediately

Theorem 8.14. Any consistent Π has a multimodel.

□

and we obtain

Theorem 8.15 (5.2). For every specification SP=(Σ, Π), clause $C \in \mathcal{L}_{neq}(\mathrm{SP})$

$$\mathrm{Comp(SP)} \models_{\exists} C \text{ implies } \Pi \vdash C$$

$$[_C \quad \mathrm{Comp(SP)} \models_C C \text{ implies } \Pi \vdash C \quad]$$

Proof:

By theorem 8.14 any consistent specification SP has a multimodel. By theorem 4.10 we can conclude that SP has a computation M∈Comp(SP). Lemma 8.10 gives then the conclusion.

□

APPENDIX B. Soundness of NEQ (theorem 5.2).

In this appendix we prove soundness of NEQ wrt. multialgebraic and functional semantics. The proof shows soundness wrt. $\exists$-satisfaction and $\mathcal{C}$–satisfaction after which lemma 4.12 gives soundness wrt. $\mathcal{P}$-satisfaction. The proof for $\mathcal{C}$–satisfaction requires only sometimes slight extensions of the proof for $\exists$ -satisfaction. We therefore carry out the latter and add appropriate extensions for the former in square braces [$_{\mathcal{C}}$ ...]. The symbols $\mathbb{T}$, $\mathbb{F}$ are used for the boolean values 'true' and 'false', respectively.

Let $M \in$Comp(SP), β: $X \to |M|$ be an assignment. For a literal $e \in \mathcal{L}_{\text{neq}}$(SP), let $\beta[e]$ denote the truth value of the formula in M under assignment β (i.e., $\beta[e]=\mathbb{T}$ iff $M \models_{\exists} e$ [$_{\mathcal{C}}$ $M \models_{\mathcal{C}} e$]. Similarly, for a clause C, $\beta[C]$ denotes the truth of C under assignment β.

R1. For every variable x and assignment β, $\beta(x)=\beta(x)$, i.e., $\beta[x] \equiv \beta[x]$. (a) So $\beta[x] \cap \beta[y] \neq \emptyset \Rightarrow \beta[x] \equiv \beta[y]$. (b) This clause is $\mathcal{C}$–valid iff there exist index terms d such that the clause $x \neq t(i) \vee x=t(d)$ is valid. Obviously choosing d to be the same index variables as i we obtain the trivial clause $x \neq t(i) \vee x=t(i)$ which is valid in any computation. $\mathcal{C}$–validity implies $\exists$-validity

R2. Assume $\beta[C^x_s, D]=\mathbb{F}$. By the second premise, $\beta[s \doteq t]$ must be true, i.e., $\beta[s] \equiv \beta[t]$. This means that for any clause C, $\beta[C^x_t]$ iff $\beta[C^x_s]$. Hence the first premise would be false. [$_{\mathcal{C}}$ This proves $\exists$-satisfiability and $\mathcal{C}$–satisfiability.]

R3. Assumption $\beta[C^x_s, D]=\mathbb{F}$ gives (as in R2) that $\beta[s \prec t]=\mathbb{T}$, i.e. $\beta[s] \subseteq \beta[t]$. By monotonicity of all operations, this means that for all f, (*): $\beta[f^x_s] \subseteq \beta[f^x_t]$. Let $e^x_t \in C^x_t$ be such that $\beta[e^x_t]=\mathbb{T}$. In each of the three possible cases for e this together with (*) contradicts the assumption. If e is $f \# g$ then, by (*), $\beta[f^x_t \# g^x_t] \Rightarrow \beta[f^x_s \# g^x_s]$. Similarly if e is $f \doteq g$ (since $\beta[f] \neq \emptyset$ for all f). If e is an inclusion, $f \prec g$ then, by the restriction on the rule, x does not occur in g. From $\beta[f^x_t] \subseteq \beta[g]$ and (*) we obtain $\beta[f^x_s] \subseteq \beta[g]$.

[$_{\mathcal{C}}$ The first two cases are the same for $\models_{\exists}$ and $\models_{\mathcal{C}}$. That M $\mathcal{C}$–satisfies $f^x_t \prec g$ under β means that there exist index terms d' making $f^x_t(j)=g(d')$ true, where j comprises the index variables k of t. Truth of $s \prec t$ implies that there exist index terms d such that $s(i)=t(d)$ holds under all assignments making D false – hence under β. In particular, substituting d for k gives us that $(f^x_t(j)=g(d'))^k_d$ holds. But this means that $f^x_s(l)=g(d')^k_d$ holds (where $l=j \cup i\text{-}k$), i.e., that there exist index terms $d''=d'^k_d$ such that M satisfies $f^x_s(l)=g(d'')$ under β. This makes the conclusion of R3 $\mathcal{C}$–satisfied in M.]

R4. If $\beta[C, D]=\mathbb{F}$ then, by the first premise, $\beta[s \preceq t]$ is true. But then $\beta[s] \cap \beta[t] \neq \emptyset$ and $\beta[s \# t]$ in the second premise will be false, making the whole second premise false.

[$_{\mathcal{C}}$ For $\mathcal{C}$–satisfiability we have two cases. The first, for $C \cap D=\emptyset$, is the same as for $\exists$-satisfiability. If the same formula occurs in C and D, in particular if both are just one and the same inclusion e, it may happen that indexing functions validating e in the first premise, say σ, and in the second one, δ, are different. The indexing function validating e in the conclusion will then be "if $(s \preceq t)'$ then δ else σ" where $(s \preceq t)'$ is (the indexed version of) the cut formula from the first premise.]

R5. The rule is trivially sound.

R6. $\exists/\mathcal{C}$–interpretation of the assumption will give a unique index (variable) i to t in $x\#t$, and its validity in M means that the clause holds for all values of i. If x does not occur in a right-hand side of $\prec$ the interpretation of the conclusion will also give a unique index to the only t (since there was at most one x) which has been substituted for x. The two clauses obtained in this way are equivalent. If x occurs in a right-hand side of $\prec$ then validity of the conclusion requires existence of an index d $[_{\mathcal{C}}$ index term d $]$ (at t substituted for x) making the interpreted clause true. But since, by the assumption, the clause is true for all values of i, we can pick any index $[_{\mathcal{C}}$ term $]$ and obtain a valid instance of the assumption.

The restriction that x does not occur in t is necessary. Otherwise ELIM would not be sound. For instance, we could use it to deduce

$$\frac{x \# t(x) \vee x \doteq 1}{t(x) \doteq 1}$$

while the structure A with $A_S=\{1, 2, 3\}$, $t^A(i, 1)=1$, $t^A(i, 2)=3$, $t^A(i, 3)=2$, satisfies the assumption but not the conclusion of this deduction.

□

Lemma 4.12 yields immediately

Corollary 9.1. For every multistructure M and rule R, if M $\mathcal{P}$–satisfies the premises of R then it $\mathcal{P}$–satisfies its conclusion.

□