CLASSIFIED MODELS FOR SOFTWARE ENGINEERING

By

Gordon Stuart

B. Sc. University of Victoria, 1970

M. Sc. University of Western Ontario, 1973

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Computer Science

We accept this dissertation as
conforming to the required standard

Dr. William W. Wadge, Supervisor (Department of Computer Science)

Dr. R. Nigel Horspool, Departmental Member (Department of Computer Science)

Dr. Hans Muller, Departmental Member (Department of Computer Science)

Dr. David Leeming, Outside Member (Department of Mathematics)

Dr. L. Robertson, Outside Member (Department of Physics)

Dr. José Meseguer, External Examiner (SRI International)

Abstract

Supervisor: W.W. Wadge

In this dissertation it is shown that abstract data types (ADTs) can be specified by the Classified Model (CM) specification language – a first-order Horn language with equality and sort "classification" assertions. It is shown how these sort assertions generalize the traditional syntactic signatures of ADT specifications, resulting in all of the specification capability of traditional equational specifications, but with the improved expressibility of the Horn-with-equality language and additional theorem proving applications such as program synthesis.

This work extends corresponding results from Many Sorted Algebra (MSA), Order Sorted Algebra (OSA) and Order Sorted Model (OSM) specification techniques by promoting their syntactic signatures to assertions in the Classified Model specification language, yet retaining sorted quantification. It is shown how this solves MSA problems such as error values, polymorphism and subtypes in a way different from the OSA and OSM solutions. However, the CM technique retains the MSA and order sorted approach to parameterization. The CM generalization also suggests the use of CM specifications to axiomatize modules as a generalization of variables within Hoare Logic, with application to a restricted, but safe, use of procedures as state changing operations and functions as value returning operations of a module. CM proof theory and semantics are developed, including theorems for soundness, completeness and the existence of a free model.

Examiners:

Dr. William W. Wadge, Supervisor (Department of Computer Science)

Dr. R. Nigel Horspool, Departmental Member (Department of Computer Science)

Dr. Hans Muller, Departmental Member (Department of Computer Science)

Dr. David Leeming, Outside Member (Department of Mathematics)

Dr. L. Robertson, Outside Member (Department of Physics)

Dr. José Meseguer, External Examiner (SRI International)

Table of Contents

List of Figures

Acknowledgement

I would like to thank my original supervisor, Dr. David Parnas, for introducing me to Software Engineering and its formalization. I would also like to thank my current supervisor Dr. Bill Wadge for introducing me to semantics and the classified approach to specification. To both of these Computer Scientists, and to Dr. Nigel Horspool who supervised me for a short while between Dr. Parnas and Dr. Wadge, I am grateful for their guidance and encouragement.

I am also grateful to my wife Nancy and to my four children for their patience and encouragement during many long years of study.

This work was supported in part by grants from the National Science and Engineering Council (Canada) to Prof. David L. Parnas and William W. Wadge.

Chapter 1

Introduction

A classified model is the intended meaning of an abstract data type (ADT) specification written in a specification language that defines sorts by assertions rather than by the syntactic signatures of conventional ADT specification techniques. A specification in the proposed language consists of a set of first-order Horn formulas (including the equality predicate) that are universally quantified over the defined sorts.

For example, the universally quantified atomic assertions and equations in figure 1.1 specify the natural numbers with the operations successor, addition and multiplication. The first two assertions (declaration assertions) inductively define the elements of sort `nat` by asserting in the base case that 0 is an element of `nat` and in the inductive step that the successor of a natural number is also a natural number. Universally quantified variables are subscripted with a sort identifier to indicate that the variable may range over all objects that are asserted by the specification to be of that sort. The remaining assertions (equations) define the binary operations using the same base and inductive cases by identifying (equating) elements of `nat` according to the usual meaning of these operations.

The software structure `STACK-OF-NAT` is specified in figure 1.2 by extending the specification `NAT` with additional assertions. The extension introduces `stk` as another sort of fundamental object and defines it by base and inductive cases in the first two assertions. The remaining assertions define the operations `top` and

$$
\begin{array}{rl}
 & \mathtt{nat(0)} \\
(\mathtt{X_{nat}}) & \mathtt{nat(s(X_{nat}))} \\
(\mathtt{X_{nat}}) & \mathtt{X_{nat} + 0 == X_{nat}} \\
(\mathtt{X_{nat}}, \mathtt{Y_{nat}}) & \mathtt{X_{nat} + s(Y_{nat}) == s(X_{nat} + Y_{nat})} \\
(\mathtt{X_{nat}}) & \mathtt{X_{nat} * 0 == 0} \\
(\mathtt{X_{nat}}, \mathtt{Y_{nat}}) & \mathtt{X_{nat} * s(Y_{nat}) == (X_{nat} * Y_{nat}) + X_{nat}}
\end{array}
$$

Figure 1.1: Classified Model Specification (NAT) For Naturals (nat)

$$
\begin{array}{rl}
 & \mathtt{stk(stknil)} \\
(\mathtt{S_{stk}}, \mathtt{X_{nat}}) & \mathtt{stk(push(S_{stk}, X_{nat}))} \\
(\mathtt{S_{stk}}, \mathtt{X_{nat}}) & \mathtt{top(push(S_{stk}, X_{nat})) == X_{nat}} \\
(\mathtt{S_{stk}}, \mathtt{X_{nat}}) & \mathtt{pop(push(S_{stk}, X_{nat})) == S_{stk}}
\end{array}
$$

Figure 1.2: Classified Model Specification STACK-OF-NAT

pop which, respectively, return and remove the most recently push-ed nat object.

Classified model (CM) specifications can be used like the familiar axiomatic theories of mathematics (e.g. groups and rings) to prove properties of the underlying objects specified or to compute with these objects. For example, using rules which include the usual algebraic rules of substitution of an expression for any variable and replacement of a term, by an equal term we can prove from the above STACK-OF-NAT specification the assertion

$$(\mathtt{T_{stk}})\ \mathtt{nat(if\ T_{stk} == stknil\ then\ 0\ else\ top(T_{stk}))}$$

that the if-then-else expression is of sort nat, given the usual definition of the if-then-else function. When $\mathtt{T_{stk}}$ is not the empty stack, stknil, then the sort of the if-then-else expression is the sort of $\mathtt{top(T_{stk})}$, i.e. sort nat according to the specification. The sort of top(stknil) is not defined by the

stack specification, but the `if-then-else` expression has the (arbitrary) value 0 when $\mathtt{T_{stk}}$ is `stknil`, so the `if-then-else` expression is also of sort `nat` in this case. Using an induction rule, to be described later, we can also prove from the specification `NAT` the assertion

$$(\mathtt{X_{nat}}, \mathtt{Y_{nat}})\ \mathtt{nat}(\mathtt{X_{nat}} + \mathtt{Y_{nat}})$$

by proving

$$(\mathtt{X_{nat}})\ \mathtt{nat}(\mathtt{X_{nat}} + \mathtt{0})$$

and

$$(\mathtt{X_{nat}})\ \mathtt{nat}(\mathtt{X_{nat}} + \mathtt{s(a)})$$

from the specification, the declaration `nat(a)` and the induction hypothesis

$$(\mathtt{X_{nat}})\ \mathtt{nat}(\mathtt{X_{nat}} + \mathtt{a}).$$

In a similar way the induction rule can be used to prove other important `NAT` assertions such as the associative, commutative, and distributive laws. The order of proofs is important: assertions proved by one application of induction are added to the specification and used in subsequent induction proofs.

1.1 The Motivation For Classified Models

The primary motivation for the introduction of classified model specifications is to encourage the use of specifications in the derivation of properties of the underlying objects specified. That is, to use specifications for more than just syntactic purposes as, for example, the way a Modula-2 compiler uses a definition module or an Ada compiler uses a package specification. Instead, a programming language type system can be implemented in the style of the ML[Wik87] language,

similar to the way the **if-then-else** expression above can be proved in the CM system to always be of sort **nat**. A conventional syntactically oriented type system of a language like Modula-2, however, ignores the nature of the Boolean test and abstracts only its sort. The best that can be concluded using such a type system is that the **if-then-else** expression is of a supersort of **nat** which includes error objects (e.g., top of an empty stack) as well. This is one of the approaches taken in the Order Sorted Algebra (OSA)[GJM85] type system where the supersort of a sort "S" is known as "S?". An OSA language such as OBJ2[FGMO87] also uses specifications for more than just type checking; OBJ2 executes specifications by interpreting them as oriented rewrite rules.

In this dissertation a different approach is taken by using specifications for derivation (theorem proving) rather than directly for computation. Type checking in the CM system is a special kind of theorem proving. A more general demonstration of theorem proving with specifications includes the synthesis of a program from a specification (a special case of planning) in a subsequent chapter. This application demonstrates the mechanics of derivations with CM specifications, but does not address the difficult aspects of choosing a proof strategy.

Classified model specifications were also motivated by the desire to write specifications that are easily understood. This, plus type checking and theorem proving in general, suggest that specifications:

- allow error values, so that for example the **stack** operation **top** when applied to a non-stack object, e.g. `top(0)`, is not classified as a natural;
- allow polymorphism, so that for example a single function **head** can be applied to a list of any sort and produce an object of the appropriate sort;
- allow one sort to be a subsort of another sort, so that for example natural

numbers may also be integers and integers may also be rationals;

- allow parameterized types, so that for example for each type `t` there is a type `stack(t)` with components of type `t`.

1.2 Many-Sorted Algebras (MSA) and Abstract Data Types (ADT)

The roots of the specification technique described in this dissertation are in algebra and specifically in many-sorted, or heterogeneous, algebra[BL70][GH78]. An algebra is a set along with operations over the set. Data types have traditionally been defined as a set of data domains, or carriers, which have named constants and operations over the carriers. Furthermore, the carriers are generated from the constants by use of the operators. Data types have often been modeled by algebras having several carriers, also known as many-sorted algebras. An abstract data type has been traditionally defined as a class of data types that are the same up to renaming of the carriers, constants and operations.

1.2.1 An Algebraic Specification

A single sorted algebraic specification, or definition of an algebra, is a description of the properties of the algebra, e.g., the axioms for a group are extended from the axioms of semigroup and monoid:

Definition 1 (Semigroup) *A semigroup is a system $(S, \cdot)$, where S is a set and $\cdot$ is a binary operation on S which satisfies the associative law:*

$$(a \cdot b) \cdot c = a \cdot (b \cdot c)$$

Definition 2 (Monoid) *A monoid is a system* $(S, 1, \cdot)$*, where* $(S, \cdot)$ *is a semigroup and* 1 *is a designated element of* S *satisfying the identity:*

$$a \cdot 1 = a = 1 \cdot a$$

Definition 3 (Group) *A group is a system* $(S, 1, {}^{-1}, \cdot)$*, where* $(S, 1, \cdot)$ *is a monoid and* $^{-1}$ *is a unary operation such that*

$$a \cdot a^{-1} = 1 = a^{-1} \cdot a.$$

An implicit closure property states that the application of any operation above to elements of S results in an element of S. There is also an implicit universal quantification of the variable symbols a, b and c over the elements of S.

Familiar examples of groups include the integer and real numbers. Another group, the "free word algebra" for these axioms, has word objects constructed from the constant and variable symbols or the operation symbols applied to word arguments, except that words which are identified by any equations represent the same object. For example, the word algebra for semigroup includes $\mathtt{a}$, $\mathtt{b}$, $\mathtt{c}$ and $\mathtt{a} \cdot \mathtt{b}$ as well as more complex word objects such as $(\mathtt{a} \cdot \mathtt{b}) \cdot \mathtt{c}$, which is identified with $\mathtt{a} \cdot (\mathtt{b} \cdot \mathtt{c})$ by the single semigroup equation.

The set of all groups is the usual meaning of the group specification. In contrast, the usual meaning of a data type specification is a free term model, and specifically an initial term model, or standard model, as described below.

Specifications for data types in this dissertation are constructed from other specifications just as the group specification is constructed from the monoid specification, which is in turn constructed from the semigroup specifications.

```
stknil :→ stk
push : stk, nat → stk
top : stk → nat
pop : stk → stk

top(stknil) == 0
top(push(S, I)) == I
pop(stknil) == stknil
pop(push(S, I)) == S
```

Figure 1.3: MSA specification STACK-OF-NAT

1.2.2 A Many-Sorted Algebraic (MSA) Specification

A data type such as a stack of natural numbers may involve several different sorts of objects and it is not possible in general to define a data type in terms of operations over any single sort. If we are concerned with stacks of naturals, for example, we must deal with operations like push and top some of whose arguments and/or results are naturals, not stacks. For this reason it is usual to use many-sorted algebras[BL70] (MSAs) rather than single sorted algebras.

In figure 1.3 a stack-of-natural is specified in the many-sorted algebra (MSA) specification style. The first part of the specification is a syntactic signature that specifies all allowed combinations of function symbols and arguments (a closure property). An operation name is followed by a colon and the operation arity (a list of the sort of each argument) and, following the arrow, the sort of the result of the operation. Together, the arity and result sort are called the rank. The symbol stknil is a constant of sort stack, much like the identity of a group is an element of the single sort represented by the set S in the group specification. Combinations of operation symbol and arguments not conforming to the syntactic signature are not allowed, for example pop(5). The fact that such terms are not

allowed is addressed in the next chapter by generalizing this syntax section of a specification to the status of assertions in the CM specification language.

The second part of the specification consists of implicitly universally quantified equations indicating combinations which represent the same value. Some of the allowed combinations of operation symbol and argument do not have to represent any specific value, for example `top(stknil)` is arbitrarily identified with zero in this example. The fact that the value of `top` must be specified for all objects of sort `stk` (specifically, for `stknil`) is a shortcoming of the MSA method that is addressed in the next chapter.

1.2.3 The Meaning Of An MSA Specification

The meaning (semantics) of a MSA specification is commonly selected as one of the following:

Classical semantics includes *all* possible many sorted algebras having the properties described by the specification. This is analogous to the meaning of single sorted specifications such as group.

Initial algebra semantics [GTW78] selects a particular many-sorted algebra that is the most general in a sense described below.

Final algebra semantics [Wan79] is a diametrically opposite approach to initial algebra semantics.

A word algebra of an MSA specification can be constructed in the same way as the word algebra described above for the group example. The objects in a free word algebra may contain variable symbols, but objects in an initial word algebra contain only ground terms having no variable symbols. The representative names

for objects of the initial word algebra are terms formed according to the signature, modulo the equations. That is, any two terms are considered to represent the same object of the initial word algebra if they are formed from constants and operations which can be to be proved equivalent as a consequence of the equations by using the usual inference rules of algebra (replacement and substitution). The objects of the initial word algebra of an MSA specification are therefore the equivalence classes of ground terms and it can easily be seen that, for example, `stknil` and `pop(push(stknil,2))` represent the same object of the above word algebra. This initial algebra has been characterized[MG85] as:

Generated: The specified objects have names. Each object in any generated algebra is represented by some ground term of the signature.

Generic: The algebra is the most general possible in the sense that any other algebra is a special case of a generic algebra, e.g., only those objects are identified (equated) which the specification *requires* to be identified. A generic algebra corresponds to a world where only those ground equations are true which are either in the specification or follow from these by application of the usual algebraic rules of inference, i.e., substitution and replacement.

Since each algebra in the class of all initial algebras defined by a specification is isomorphic to the initial word algebra [MG85], the latter is regarded as the representative of the class and the meaning of an MSA specification.

A final algebra, if it exists, for an MSA specification is an algebra in which objects are identified *unless* the specification requires them to be distinct.

1.2.4 What Can We Do With An MSA Specification?

The techniques of term rewriting [HO80] generally determine how we can reason with equational specifications. We might wish to know if another equation is a consequence of the axioms in a specification, e.g., if $0 + x = x$ is a consequence of the group axioms. We may also compute with equations by regarding them as oriented rewrite rules, e.g., functional programming languages. The state-of-the-art in equational reasoning, or the extensions of equational reasoning to Horn logic with equality, depends upon advances in the theory of term rewriting and theorem proving in general.

Besides reasoning with axioms we can use them just as specifications. As described in [BG77], we should not build monolithic specifications. We should instead construct large specifications from smaller ones using appropriate techniques of parameterization or by otherwise extending previous work as above for the group example.

1.2.5 Related Work

The literature on MSA and related techniques is so vast that even a dated bibliography such as [KL83] has over 1800 titles. Since a current bibliography is surely much larger, no attempt is made to summarize related work. A recent explanation of MSA is given in [MG85]. Some of the limitations of the MSA technique are cited in work such as [Maj77].

1.3 The Problem: Data Types Are Not The Same As MSAs

As noted earlier, data types have traditionally been defined as a set of data domains, or carriers, which have named constants and operations over the carriers.

Furthermore, the carriers are generated from the constants by use of the operators. A many-sorted algebra is a set of disjoint sets along with operations over the sets. Data types have often been studied formally as many-sorted algebras, but data types are not just many-sorted algebras. A number of problems encountered in the interpretation of data types as many-sorted algebras cannot be resolved within the many-sorted algebra formalism. First, the "constructor-selector"[GM87a] problem cannot be solved within the MSA formalism. Other problems involve:

- Error values,
- Polymorphism, and
- Subsorts.

Error values result from nonsensical combinations of operations. Recall that in the MSA formalism the sort of an operation is dictated by its signature so that, for example, in the `STACK-OF-NAT` MSA specification the `top` of any stack, including the empty stack, is a natural number. There are several ways of dealing with the problem. We can ignore the problem by assigning default values to the problem combinations, e.g., `top(nilstk)` $=$ `0`, but then it is very hard to even formulate the concept of a safe program. Alternatively, we can introduce special error objects explicitly, but then the specification is complicated by the requirement to qualify some equations with preconditions or to distinguish "ok" equations from "error" equations.

Polymorphism (overloading) is the use of the same operation symbol with various argument and result sorts. Because in the MSA formalism the sort of each operation symbol is dictated by the signature, we must use a different operation symbol for each different argument and result sort. For example, the addition

operator "+" is used in mathematics and in most programming languages as an overloaded operator for natural numbers, integers, rationals, etc., but in the MSA formalism we must use a different symbol for each different use. To make matters worse, each of these operations needs its own copy of the "generic" specifying equations that are given in the introduction to this chapter.

Subsorts are not allowed in the MSA formalism. The resolution of this problem is fundamental to the solution of the two previous problems. If, for example, we consider the natural numbers to be a subsort of all the possible values returned by the `STACK-OF-NAT` operation `top`, then it makes sense to consider `top(stknil)` as just a member of the supersort. Similarly, if we consider the natural numbers as a subset of the integers and the integers as a subset of the rationals, we can easily consider a polymorphic addition operator.

The main problem with the MSA formalism is that its sort structure is based on a simplistic approach to sorts in which each operator is sorted statically in a signature declaration. In the beginning of the ADT research effort many-sorted algebras may have appeared to be an appropriate formalism for data types, but over time enough troublesome cases have been presented to warrant the consideration of more general formalisms. The solution suggested in the classified model approach is to generalize the notion of sort so that different terms constructed from the same operation symbol can perhaps have different sorts and so that a single term, or set of terms, can have more than one sort. This also induces a subsort relation among sorts.

1.4 Order Sorted Algebra (OSA) And Order Sorted Model (OSM)

Order sorted algebra (OSA)[GJM85] techniques generalize MSA by providing a subsort partial ordering among the sorts. Operator overloading and nonsensical application of function symbols, "errors", are handled by restricting functions to subsorts. Order sorted model (OSM) techniques generalize OSA by allowing predicates and Horn formulas instead of just conditional equations.

For example, a stack-of-natural, with subsort **nstk** standing for non-empty stack, is specified in the order sorted specification language OBJ2 of figure 1.4. The first indented statement indicates that the natural numbers, **nat**, are neither extended or contracted, i.e., an OBJ2 specification **NAT** is enriched to **STACK-OF-NAT**. The next two statements indicate that **stk** (a stack) and **nstk** (non-empty stack) are sorts, where **nstk** is a subsort of **stk**. The lines starting with "op" form a syntactic signature that specifies, as in MSA, all allowed combinations of function (operation) symbols and arguments. The "var" statements declare variables for the following equations indicating combinations which represent the same value. Sound and complete rules of inference exist for OSM[GM87b] and include the MSA and OSA systems as special cases.

The Order Sorted methods are an improvement over the many sorted techniques since some of the problems mentioned above are solved. But, the subsorts of OSA induce a new problem which does not arise with the old "pigeon-holing" MSA approach in which each term has exactly one sort. It is not possible to reason about the sort of the value of an expression, i.e., about special values an expression may have because of special properties of the algebra, as we did in the opening section of this chapter for an **if-then-else** expression.

```
obj  STACK-OF-NAT is
     protecting nat  .
     sorts stk, nstk  .
     subsorts nstk < stk .
     op stknil :→ stk .
     op push : stk nat → nstk .
     op top : nstk → nat .
     op pop : nstk → stk .
     var I  :  nat
     var S  :  stk
     eq top(push(S,I)) = I .
     eq pop(push(S,I)) = S .
jbo
```

Figure 1.4: Order Sorted OBJ2 STACK-OF-NAT specification

1.5 A Solution: Signature Generalizations

The technique chosen in this dissertation to solve the problems of MSA is to generalize the notion of a signature and of the sort of an operation. Recall that the sorts of an MSA are not ordered, and that each operation symbol has a unique sort prescribed by the syntactic signature. Polymorphism can be achieved in the CM proposal by allowing an operation symbol to have more than one sort. Subsorts can be achieved by allowing a partial order on the sort symbols to be induced by sort declarations of terms in the assertion language. This is the Classified Model approach of this dissertation.

The classified model (CM) approach to the specification and verification of abstract data types and modules (abstract data types with hidden "state" sorts) allows a semantic rather than syntactic definition of subsorts. A classified model is a single sorted model which has a classification of its universe into a family of not necessarily disjoint subsets.

1.6 Institutions

Many basic results hold in more than one logical system (MSA, OSA, CM) and the notion of *institution*[GB84] was introduced as a generalization of "logical system". An institution consists of:

- a collection of signatures (or vocabularies) and signature morphisms for each signature Σ;
- a set of Σ-sentences which can be formed from the vocabulary;
- a set of Σ-models;
- a satisfaction relation for each signature: $\Sigma\text{-}model \models_\Sigma \Sigma\text{-}sentence$.

such that the satisfaction relation is invariant under signature morphisms. All of the logical systems described above are institutions and all (but not full first order logic) are "liberal" institutions. Goguen and Burstall state in the abstract of [GB84]:

> A first main result shows that if an institution is such that interface declarations expressed in it can be glued together, then theories (which are just sets of sentences) in that institution can also be glued together. A second main result gives conditions under which a theorem prover for one institution can be validly used on theories from another; this uses the notion of an institution morphism. A third main result shows that institutions admitting free models can be extended to institutions whose theories may include, in addition to the original sentences, various kinds of constraints upon interpretations; such constraints are

useful for defining abstract data types, and include so-called "data", "hierarchy" and "generating" constraints.

Several specification languages, e.g., Clear[BG77], OBJ[FGMO87] and ACT ONE[EM85] are defined for specific liberal institutions, but by the results cited above many of their concepts can be safely translated to other liberal institutions. That is an important thrust of this thesis: many of the parameterization and modularization techniques already defined in the literature can be transferred to the CM institution. The reason for this is that any CM assertion is easily relativized to an equivalent assertion in the Horn-with-equality institution by simply omitting the sort subscripts from variable symbols and including in the antecedent of the assertion a corresponding sort predicate of the altered variable. This translation can also be expressed as an "institution morphism" [GB84]. The important point is that the Horn-with-equality logical system is a liberal institution[GB84].

The results of [GB84] specifically show that for the Horn-with-equality liberal institution:

- The first main result of [GB84] states that existing parameterization techniques can be used with Horn-with-equality which has a "free construction" which allows interface specifications to be glued together.

- The second main result of [GB84] states that we can use a general first order deduction technique for Horn-with-equality.

- The third main result of [GB84] states that we can use all the constraint mechanisms of languages like Clear[BG77].

1.7 Why Use Horn-With-Equality For CM Specification?

The first order language of Horn formulas with equality is sufficiently restricted to allow the desirable free construction property, with all of its implications for parameterization.

It is also the richest first order language which *admits initial models.* That is, after an axiom is changed in, or added to, a specification there is still some initial model for the specification. Any set of first order axioms having this property are provably equivalent to a Horn theory[Mak87]. All of these results also apply to this thesis since the first order language of Horn formulas with equality is co-extensive with the classified model specification language having syntactically sorted variables.

1.8 Thesis Organization

Chapter 2, Classified Models, is a description of the main points of the classified model approach. All detailed definitions and proofs have been deferred to an Appendix A.

Chapter 3, Parameterized Module Specifications, is a description of existing parameterization and modularization techniques for the basic classified model specifications of Chapter 2. This chapter concludes with a parameterized key-word-in-context (kwic) module CM specification that depends upon parameterized CM ADT specifications for queue and sorting queue. This example illustrates how specifications can be constructed from smaller parts.

Chapter 4, Classified Model Situational Logic, establishes a notation for a situational logic which depends for conciseness upon the flexible sort structure of the classified model approach.

Chapter 5, Hoare Logic and Situational Classified Model Specifications, applies the situational logic to a generalization of the Hoare logic for program verification. This application has the potential to allow the safe use of procedures and functions in a Hoare verification when they are used in the context of operations upon a module specified by the CM technique.

Chapter 6, Program Synthesis, is another application of the situational logic in which CM specifications are used in the synthesis of programs. An advantage is noted for CM specifications over more general first order specifications cited in the literature.

Chapter 7, A Software Engineering Technique, describes a practical software engineering technique which supports the theory of this thesis that procedures and functions can be used safely in the context of module operations.

Chapter 8, Conclusions and Further Research, lists the major results of this work and its potential for further development.

Appendix A, Classified Horn Logic With Equality, contains a detailed proof for each of the results reported in Chapter 2.

Chapter 2

Classified Models

The signature section of a conventional MSA or of an order sorted specification embodies a fundamental assumption of these methods: they are based on a notion of sort which is primarily syntactic. That is, a type system based upon a signature is a classification of syntactic objects, i.e., expressions. The sort of an expression is dependent upon the sorts of its sub-expressions.

In the classified model (CM) approach the declarations of a MSA signature are promoted to the status of full fledged assertions. The information that is coded in a conventional signature, and more, can be expressed as declaration assertions. With this view, a type system is primarily a classification of semantic objects, i.e., of data objects. The sort of an expression can be deduced as a theorem and is not just a syntactic consequence of the sorts of its sub-expressions.

Although the classification permitted by the order sorted methods is more sophisticated than those allowed by the MSA technique, the implications of the signature remain the same. The CM approach can be viewed as carrying the enrichment from MSA to order sorted methods to its logical conclusion.

2.1 Syntax

The definition of the language of classified model specifications (the CM language) is divided into three parts. The vocabulary introduces disjoint sets of symbols which are used to construct the language. The terms and formulas give

formation rules for the construction of legal objects of the language.

The vocabulary consists of symbols that have a fixed meaning and symbols whose meaning is to be defined, also known as a signature. In the applied logic, or object language, used in specification examples the symbols of the signature are in the `teletype` font and the symbols are chosen to convey to the reader the defined meaning. In metatheoretical proofs, such as soundness and completeness, the symbols of the signature are metavariables that range over the symbols of the object language. These metavariables are depicted in **sans serif** type font.

Definition 4 (Vocabulary) *The vocabulary consists of all the defined symbols of the language.*

1. *The implication connective is* $\rightarrow$.
2. *A set of signatures is defined, where a specific signature* Σ *contains function and predicate symbols of defined arities, or number of argument places.*
   (a) *The function symbols include the constant (0-ary function) symbols.*
   (b) *The predicate symbols include:*
      i. *the propositional constant (0-ary predicate) symbols,*
      ii. *the sort (unary predicate) symbols, and*
      iii. *the infix binary identity predicate "$\mathbin{=\!\!=}$", which is different from the metalanguage equality symbol "$=$".*
3. *A set of unsorted variable symbols is defined.*
4. *A set of sorted variables symbols is defined, where each sorted variable symbol is subscripted by some sort predicate symbol.*
5. *The universal quantifier* $(\mathcal{X})$ *is defined, where* $\mathcal{X}$ *is any set of variables.*

Notation:

1. Script symbols are used to represent:

   (a) an arbitrary formula, e.g., $\mathcal{F}$ represents either an atomic formula or a Horn formula (defined below);

   (b) a collection of variables, e.g., $\mathcal{X}$;

2. Brackets "$[\![$" and "$]\!]$" are used to enclose CM syntax objects in mathematical expressions.

The set of $\Sigma(\mathcal{X})-$terms is constructed from a signature Σ and a set of variables $\mathcal{X}$ by combining function symbols with operand expressions.

Definition 5 ($\Sigma(\mathcal{X})-$Terms) *The set of $\Sigma(\mathcal{X})-$terms is defined inductively from a signature Σ and a set of variables $\mathcal{X}$ by:*

1. *every variable symbol in $\mathcal{X}$ is a term;*
2. *if $\mathsf{t}_0, \ldots, \mathsf{t}_{n-1}$ are terms and f is an n-ary function symbol, then $\mathsf{f}(\mathsf{t}_0, \ldots, \mathsf{t}_{n-1})$ is a term.*

The set of *ground* $\Sigma-$terms, $\Sigma(\emptyset)$, have no variable symbols.

The set of formulas is constructed from the set of terms by using the implication logical connective and universal quantification.

Definition 6 ($\Sigma-$Formulas) *The set of $\Sigma-$formulas is defined inductively from a signature Σ and a set of variables $\mathcal{X}$ by:*

1. *if $\mathsf{t}_0, \ldots, \mathsf{t}_{n-1}$ are $\Sigma(\mathcal{X})-$terms and P is an n-ary predicate symbol, then $\mathsf{P}(\mathsf{t}_0, \ldots, \mathsf{t}_{n-1})$ is an atomic formula;*

2. *if* A *is an atomic formula (the head) and the set* $\{ \mathsf{B}_0, \ldots, \mathsf{B}_{n-1} \}$ $(n \geq 0)$ *are atomic formulas (the body) then* $\mathsf{B}_0, \ldots, \mathsf{B}_{n-1} \rightarrow \mathsf{A}$ *is a Horn formula;*

3. *if* $\mathcal{X}$ *is a list of variable symbols and* $\mathcal{F}$ *is an atomic or Horn formula, then* $(\mathcal{X})\,\mathcal{F}$ *is a closed formula, where* $\mathcal{X}$ *includes (at least) all the variables occurring in all the terms in* $\mathcal{F}$.

A Horn formula with no body is identified with an atomic formula having just the head, but not the implication connective.

Whenever "formula" is mentioned below, it is assumed to be closed unless otherwise stated.

Definition 7 (Substitution) *Let* Σ *be a signature, let* $\mathcal{X}$ *and* $\mathcal{Y}$ *be sets of variables and let* θ *be a function* $\theta : \mathcal{X} \rightarrow \Sigma(\mathcal{Y})$. θ *is called a substitution function and can be extended to a function* $\theta^* : \Sigma(\mathcal{X}) \rightarrow \Sigma(\mathcal{Y})$*:*

1. *for each variable symbol* X *in* $\mathcal{X}$, $\theta^*\mathsf{X} = \theta\mathsf{X}$

2. *for each n-ary function symbol* f, $\theta^*\mathsf{f}(\mathsf{t}_0, \ldots, \mathsf{t}_{n-1}) = \mathsf{f}(\theta^*\mathsf{t}_0, \ldots, \theta^*\mathsf{t}_{n-1})$.

3. *for each n-ary predicate symbol* P, $\theta^*\mathsf{P}(\mathsf{t}_0, \ldots, \mathsf{t}_{n-1}) = \mathsf{P}(\theta^*\mathsf{t}_0, \ldots, \theta^*\mathsf{t}_{n-1})$,

θ^* *can also be applied to a Horn formula by applying it to each individual atomic formula.*

θ^* is commonly abbreviated as θ.

Definition 8 (Specification) *A* specification $S = (\Sigma, \Gamma)$ *consists of a signature* Σ *and a set of closed* Σ*–formulas* Γ.

The following terminology is used:

1. An *equation* is an atomic formula constructed from just the equality predicate.

2. A *conditional equation* is a Horn formula constructed from just the equality predicate.

3. A *base assertion* is an atomic formula constructed from a unary predicate symbol.

4. A *generation assertion* is a Horn formula having a unary predicate symbol in the head.

5. A *declaration assertion* is a base or generation assertion.

6. A *declaration* is an instance of a unary sort predicate.

2.2 Semantics

The semantics of the CM language provides a meaning for the syntax objects described above by defining:

1. Σ-model for the elements of the signature;

2. assignment function for variables;

3. extended assignment function for terms;

4. truth in a model for closed formulas.

Definition 9 (Σ-Model) *For any given signature Σ, a Σ-model M is a pair (D_M, α_M) where*

1. *D_M is a non-empty set called the universe, or domain, of M;*

2. α_M *is a* Σ−interpretation function *which assigns:*

   (a) *for each 0-ary function (i.e., constant) symbol* c *in* Σ: $\alpha_M[\![\mathsf{c}]\!] \in D_M$;

   (b) *for each other n-ary function symbol* f *in* Σ *a function* $\alpha_M[\![\mathsf{f}]\!]: (D_M)^n \rightarrow D_M$;

   (c) *for each 0-ary function (i.e., propositional) symbol* P *in* Σ *an element* $\alpha_M[\![\mathsf{P}]\!] \in \{T, F\}$;

   (d) *for each other n-ary predicate symbol* P *in* Σ *a subset* $\alpha_M[\![\mathsf{P}]\!]$ *of the Cartesian power* $(D_M)^n$. *The truth set of the equality predicate must be* $\{<d, d> : d \in D_M\}$.

Like a conventional single sorted model, there is no *a priori* notion of sort for operation symbols. The sets $\{\alpha(\mathsf{s}) : \mathsf{s}$ is a sort symbol$\}$ correspond to the carriers in a conventional MSA. In a classified model M there might be elements of the universe which are not in any of the "carriers". These are error objects which are the result of operations such as pop(stknil) that have no intended purpose. It is shown below that for "normal" specifications in which every term in the specification is of some classification these extraneous objects cause no difficulties in practice.

A particularly important $\Sigma(\mathcal{X})$−Term model is defined by generation over a set of variable symbols.

Definition 10 ($\Sigma(\mathcal{X})$**−Term Model** $\mathcal{T}_\Sigma(\mathcal{X})$) *Let* Σ *be a signature and* $\mathcal{X}$ *a set of variables, then* $\mathcal{T}_\Sigma(\mathcal{X})$ *is the* Σ*-model with universe the set of* $\Sigma(\mathcal{X})$*−terms and an interpretation mapping* α *for the symbols in* Σ *such that:*

1. *each* $\Sigma(\mathcal{X})$*−term is interpreted as itself, that is:*

(a) for each variable $\mathsf{X} \in \mathcal{X}$*:* $\alpha[\![\mathsf{X}]\!] = \mathsf{X}$

(b) If $\mathsf{t}_0, \ldots, \mathsf{t}_{n-1}$ *are* $\Sigma(\mathcal{X})$*−terms and* f *is a n-ary function symbol, then* $\alpha[\![\mathsf{f}]\!](\alpha[\![\mathsf{t}_0]\!], \ldots, \alpha[\![\mathsf{t}_{n-1}]\!]) = \mathsf{f}(\mathsf{t}_0, \ldots, \mathsf{t}_{n-1})$

2. *for the identity predicate symbol:* $\alpha[\![==]\!] = \{<\mathsf{t},\mathsf{t}> : \mathsf{t}$ *is a* $\Sigma(\mathcal{X})$*−term* $\}$*;*

3. *any sort predicate symbol* s *is interpreted as just the set of sorted variables that have that sort subscript, i.e.,* $\alpha[\![\mathsf{s}]\!] = \{\mathsf{X}_\mathsf{s} : \mathsf{X}_\mathsf{s}$ *is a variable with sort subscript* s $\}$*;*

4. *any other predicate symbol* P *is interpreted as the empty set, i.e.,* $\alpha[\![\mathsf{P}]\!] = \{\}$*;*

$\mathcal{T}_\Sigma(\emptyset)$ is abbreviated $\mathcal{T}_\Sigma$.

Any Σ−model is given a $\Sigma(\mathcal{X})$−model structure by extending a Σ−interpretation by an *assignment function* which interprets the variable symbols.

Definition 11 (Assignment Function) *Let* Σ *be a signature, let* $\mathcal{X}$ *be a set of variables, let* $M = (D_M, \alpha)$ *be a* Σ*-model with interpretation function* α *and let* θ *be a function* $\theta: \mathcal{X} \rightarrow D_M$ *(*θ *is called an* assignment function*). The* extended assignment function θ^* *interprets* $\Sigma(\mathcal{X})$*−terms and formulas.*

1. *for each variable symbol* $\mathsf{X} \in \mathcal{X}$*,* $\theta^*[\![\mathsf{X}]\!] = \theta[\![\mathsf{X}]\!]$.

2. *for each constant symbol* c*,* $\theta^*[\![\mathsf{c}]\!] = \alpha[\![\mathsf{c}]\!]$.

3. *for each function symbol* f*,* $\theta^*[\![\mathsf{f}(\mathsf{t}_0, \ldots, \mathsf{t}_{n-1})]\!] = \alpha[\![\mathsf{f}]\!](\theta^*[\![\mathsf{t}_0]\!], \ldots, \theta^*[\![\mathsf{t}_{n-1}]\!])$.

4. *for each propositional symbol* P*,* $\theta^*[\![\mathsf{P}]\!] = \alpha[\![\mathsf{P}]\!]$

5. $\theta^*[\![\mathsf{t}_1 == \mathsf{t}_2]\!] = T$ *if* $\theta^*[\![\mathsf{t}_1]\!] = \theta^*[\![\mathsf{t}_2]\!]$*, and* $\theta^*[\![\mathsf{t}_1 == \mathsf{t}_2]\!] = F$ *otherwise.*

6. *for each predicate symbol* P*,* $\theta^*[\![\mathsf{P}(\mathsf{t}_0, \ldots, \mathsf{t}_{n-1})]\!] = T$ *if* $<\theta^*[\![\mathsf{t}_0]\!], \ldots, \theta^*[\![\mathsf{t}_{n-1}]\!]> \in \alpha[\![P]\!]$*, and* $\theta^*[\![\mathsf{P}(\mathsf{t}_0, \ldots, \mathsf{t}_{n-1})]\!] = F$ *otherwise.*

7. *for each Horn formula,* $\theta^*[\![\mathsf{B}_0, \ldots, \mathsf{B}_{m-1} \rightarrow \mathsf{A}]\!] = T$ *if*
   whenever $\theta^*[\![\mathsf{B}_i]\!] = T$ *for* $i = 0, \ldots, m-1$ *then* $\theta^*[\![\mathsf{A}]\!] = T$,
   and $\theta^*[\![\mathsf{B}_0, \ldots, \mathsf{B}_{m-1} \rightarrow \mathsf{A}]\!] = F$ *otherwise.*

Definition 12 (Sorted Assignment) *An assignment function* $\theta: \mathcal{X} \rightarrow D_M$ *is a* sorted assignment *function if whenever* $\mathsf{X}_s \in \mathcal{X}$ *then* $\theta[\![\mathsf{X}_s]\!] \in \alpha[\![s]\!]$, *and similarly for an* extended sorted assignment.

Definition 13 (Truth In A Model) *The* truth value *of a closed formula* $(\mathcal{X})\,\mathcal{F}$ *in a model* M *with universe* D_M *and interpretation function* α *is defined by extending* α *to a function* α^**:*

$\alpha^*[\![(\mathcal{X})\ \mathcal{F}]\!] = T$ *if* $\theta^*[\![\mathcal{F}]\!] = T$ *for any sorted assignment* $\theta: \mathcal{X} \rightarrow D_M$,
and $\alpha^*[\![(\mathcal{X})\ \mathcal{F}]\!] = F$ *otherwise.*

Definition 14 (S-Model, $\models$) *Let* Σ *be a signature and* Γ *a set of* Σ*–formulas. For a specification* $S = (\Sigma, \Gamma)$, *a formula* $(\mathcal{X})\,\mathcal{F}$ *and any* Σ*-model* M *with interpretation mapping* α*:*

1. $M \models (\mathcal{X})\,\mathcal{F}$ *(*M satisfies $(\mathcal{X})\,\mathcal{F}$, *or* $(\mathcal{X})\,\mathcal{F}$ is valid *in* M*) iff* $\alpha^*[\![(\mathcal{X})\,\mathcal{F}]\!] = T$
2. *S-model: a Σ-model that satisfies all formulas of Γ;*
3. *$Mod(S)$: the class of all S-models;*
4. *$Th(M)$: (theory of M) the set of all Σ-formulas that are valid in M;*
5. $S \models (\mathsf{X})\,\mathcal{F}$ *(*$(\mathcal{X})\,\mathcal{F}$ *is a* logical consequence *of* S*) iff* $(\mathcal{X})\,\mathcal{F}$ *is satisfied in any S-model*[1].

Along with the definition of model, it is useful to define a homomorphism between models as a structure-preserving mapping.

[1] $\Gamma \models \mathcal{F}$ is an abbreviation for $S \models \mathcal{F}$ when Σ is obvious from the context of $S = (\Sigma, \Gamma)$.

Definition 15 (Σ-Homomorphism) *Let M and N be Σ-models with interpretation mappings α and β, then $h : D_M \rightarrow D_N$ is a* Σ-homomorphism *if:*

1. $h(\alpha[\![\mathsf{f}]\!](d_0, \ldots, d_{n-1})) = \beta[\![\mathsf{f}]\!](h(d_0), \ldots, h(d_{n-1}))$ *for every n-ary function symbol* f *and all* $d_0, \ldots, d_{n-1} \in D_M$.
2. *if* $<d_0, \ldots, d_{n-1}> \in \alpha[\![\mathsf{P}]\!]$ *then* $<h(d_0), \ldots, h(d_{n-1})> \in \beta[\![\mathsf{P}]\!]$ *for every n-ary predicate symbol* P *and all* $d_0, \ldots, d_{n-1} \in D_M$.

2.3 Type Properties

Declaration assertions allow explicit formulation of the type properties of expressions which in other systems are formulated implicitly by the signature. The availability of declarations relieves us from the necessity of encoding sort information in a syntactic form (as in an MSA or order sorted signature) and at the same time is much more general as well. The simplest classified approach would include in the signature just equality and sort predicate symbols, but the generalization to arbitrary predicate symbols does not change the type structure and also has some advantages in the expressibility of specifications. It also makes sense to include this more general signature because all of the same mathematical concepts that are needed to formalize the simplest approach are needed as well for the general approach, as shown in Appendix A. The simplest approach was first described by W.W. Wadge[Wad82], on which much of this chapter is based.

For example, the CM declaration assertion

$$(\mathtt{T_{stack}}, \mathtt{I_{int}})\ \mathtt{stack(push(T_{stack}, I_{int}))}$$

replaces the conventional MSA syntactic typing of push as

$$\mathtt{push}: \mathtt{stack}, \mathtt{int} \rightarrow \mathtt{stack}.$$

Polymorphism can be accommodated by using more than one declaration; for example,

$$\begin{aligned}(\mathtt{X_{nat}}, \mathtt{Y_{nat}}) \quad & \mathtt{nat}(\mathtt{X_{nat}} + \mathtt{Y_{nat}}) \\ (\mathtt{X_{int}}, \mathtt{Y_{int}}) \quad & \mathtt{int}(\mathtt{X_{int}} + \mathtt{Y_{int}}) \\ (\mathtt{X_{real}}, \mathtt{Y_{real}}) \quad & \mathtt{real}(\mathtt{X_{real}} + \mathtt{Y_{real}}).\end{aligned}$$

Declarations can also be used to specify an ordering between sorts: the declaration $(\mathtt{X_{int}})\ \mathtt{real}(\mathtt{X_{int}})$ asserts that sort `int` is a subsort of sort real. This declaration is true in a model M with interpretation function α iff $\alpha(\mathtt{int})$ is a subset of $\alpha(\mathtt{real})$.

Declaration assertions can be used to give "special case" sort information which cannot be deduced just from sort information of subterms, for example:

$$\begin{aligned}(\mathtt{X_{real}}) \quad & \mathtt{int}(\mathtt{X_{real}} - \mathtt{X_{real}}) \\ (\mathtt{X_{int}}, \mathtt{Y_{real}}) \quad & \mathtt{int}(\mathtt{if\ True\ then\ X_{int}\ else\ Y_{real}}) \\ (\mathtt{X_{real}}) \quad & \mathtt{real}(\mathtt{if\ X_{real} > 0\ then\ sqrt}(\mathtt{X_{real}})\ \mathtt{else\ sqrt}(-\mathtt{X_{real}}))\end{aligned}$$

2.4 Initial Models

The specification of abstract data types and modules[2] was the primary motivation for the development of the classified approach. A specification is a collection of assertions which is satisfied by the models of the intended data types and operations. In general, a specification has many different models. That is, there are many different models in which all the assertions in the specification are true. The set of initial term models of the specification is one particular collection of models associated with the specification which have a claim to be the intended standard models.

[2] A module is a generalization of abstract data type in which some sorts are designated as "hidden state" sorts.

An initial term model, defined below, can also be characterized as a model which is:

Generated: The objects we specify must have names. Each object in any generated model is represented by some term of the signature.

Generic: The model is the most general possible in the sense that any other model is a special case of a generic model, e.g., only those objects are identified which the specification requires to be identified and only those objects are classified as sort **s** which the specification requires to be so classified.

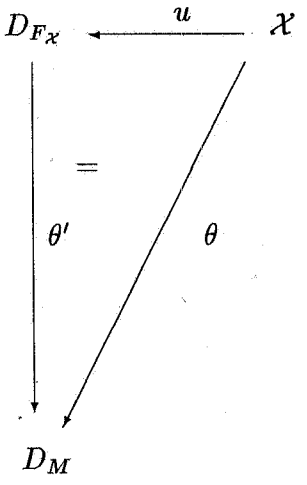
The initial model is therefore the minimal solution to the specification in the sense that the truth set of each predicate symbol (including $=\!=$) is minimal.

Definition 16 (Initial Model) *Let $\mathcal{C}$ be a class of Σ-models. A Σ-model $I \in \mathcal{C}$ is called* initial in $\mathcal{C}$ *iff for each Σ-model $M \in \mathcal{C}$ there is a unique Σ-homomorphism $h : I \to M$.*

If $\mathcal{C}$ is of the form $Mod(\Gamma)$, where Γ is some set of Σ-formulas, we also say that I is initial for Γ.

A generalization of initiality is given by freeness:

Definition 17 (Free Model) *Let $\mathcal{C}$ be a class of Σ-models, $\mathcal{X}$ a set of variables and $M \in \mathcal{C}$. A Σ-model $F_{\mathcal{X}} \in \mathcal{C}$ is called* free over $\mathcal{X}$ in $\mathcal{C}$ *iff there is a sorted assignment $u: \mathcal{X} \to D_{F_{\mathcal{X}}}$, called a universal mapping, such that for every sorted assignment $\theta: \mathcal{X} \to D_M$ there is a unique Σ-homomorphism $\theta' : D_{F_{\mathcal{X}}} \to D_M$ such that $\theta = \theta' \circ u$.*



Like the MSA and order sorted systems, there is always an initial model in the class of models satisfying a set of Horn formulas [Mak87]. Furthermore, it is also shown in [Mak87] that any (finite) first order specification that *admits an initial model* is equivalent to a (finite) Horn specification. For a specification S in a language $\mathcal{L}$ to admit an initial model means that S has an initial model and for any set of assertions $\mathtt{C}$ of $\mathcal{L}$ the specification $S \cup \mathtt{C}$ also has an initial model, but not necessarily the same initial model. This is an important characteristic for program specifications which evolve through enrichment by additional assertions or are parameterized by other specifications to be included later. Therefore, if it is important to ensure that a specification has an initial model then it is fruitless to search for first order specification languages more powerful than Horn specifications.

An important result of this dissertation, proved in the Appendix, is that any classified model specification has a free term model.

2.5 Classified Model Deduction

In the CM method sorts are declared by assertions, whereas in the Order Sorted method sorts are declared in a syntactic sort declaration section that is separate from the assertions. The semantic consequence of this is that classified models may have "error" objects that are not in the truth set of any sort predicate, e.g., `pop(stknil)` in the stack example. These objects do not even exist in order sorted models but they are rendered harmless in the classified model system since the rules of inference described below introduce only classified terms into a proof derived from a specification having only classified terms, i.e., a "normal specification".

Definition 18 (Classified Σ-Term) *Let* $S = (\Sigma, \Gamma)$ *be a specification and let* t *be a* $\Sigma(\mathcal{X})-$*term, then* t *is a* classified Σ-term *iff in the initial* Σ*-model with interpretation* α *and for all sorted assignment functions* θ *that extend* α *there is a classification* $\mathsf{s} \in \Sigma$ *(perhaps depending upon* θ*) such that:*

$$\theta^*(\mathsf{t}) \in \alpha(\mathsf{s}).$$

Definition 19 (Normal Specification) $S = (\Sigma, \Gamma)$ *is a* normal specification *iff each term* t *appearing in* Γ *is a classified* Σ*-term.*

It is shown in the Appendix that a variant (reproduced below) of the order sorted model rules of [GM87b] are sound and complete for atomic formulas of the CM system. Even incompletely specified operations are allowed because from a normal specification the rules allow the introduction of only classified terms. Incompletely specified operations are not the same as partial functions because in the initial model, and also in the free model, every term of the language has a

meaning. The classification, or lack of classification, does not affect the existence of a defined meaning for each term of the language.

The set of derivable formulas is defined by the following set of rules of inference. Each formula below is universally quantified, where all the variables appearing in such a formula are represented in the list of variables.

1. Reflexivity: $(\mathcal{X})\ \mathsf{t} = \mathsf{t}$ is derivable.

2. Symmetry: If $(\mathcal{X})\ \mathsf{t} = \mathsf{t}'$ is derivable then $(\mathcal{X})\ \mathsf{t}' = \mathsf{t}$ is derivable.

3. Transitivity: If $(\mathcal{X}')\ \mathsf{t} = \mathsf{t}'$ and $(\mathcal{X}'')\ \mathsf{t}' = \mathsf{t}''$ are derivable and if $\mathcal{X} = \mathcal{X}' \cup \mathcal{X}''$ then $(\mathcal{X})\ \mathsf{t} = \mathsf{t}''$ is derivable.

4. Substitutivity: if $(\mathcal{X}_\mathsf{i})\ \mathsf{t}_\mathsf{i} = \mathsf{t}'_\mathsf{i}$ is derivable and if $\mathcal{X} = \cup\mathcal{X}_\mathsf{i}$ for $\mathsf{i} = 0, \ldots, \mathsf{n}-1$ then:

   (a) for any n-ary function symbol f in Σ the equation
   $(\mathcal{X})\ \mathsf{f}(\mathsf{t}_0, \ldots, \mathsf{t}_{n-1}) = \mathsf{f}(\mathsf{t}'_0, \ldots, \mathsf{t}'_{n-1})$ is derivable;

   (b) for any n-ary predicate symbol P in Σ, if $(\mathcal{X})\ \mathsf{P}(\mathsf{t}_0, \ldots, \mathsf{t}_{n-1})$ is derivable then $(\mathcal{X})\ \mathsf{P}(\mathsf{t}'_0, \ldots, \mathsf{t}'_{n-1})$ is derivable.

5. Modus Ponens: If $(\mathcal{X})\ \mathsf{B}_0, \ldots, \mathsf{B}_{m-1} \rightarrow \mathsf{A}$ is derivable and if $\theta: \mathcal{X} \rightarrow \Sigma(\mathcal{Y})$ is any substitution such that:

   (a) each $(\mathcal{Y})\ \theta\mathsf{B}_\mathsf{i}$ is derivable for $\mathsf{i} = 0, \ldots \mathsf{m}-1$, and

   (b) $(\mathcal{Y})\ \mathsf{s}(\theta\mathsf{X}_\mathsf{s})$ is derivable for each $\mathsf{X}_\mathsf{s} \in \mathcal{X}$,

   then $(\mathcal{Y})\ \theta\mathsf{A}$ is derivable.

The substitution in rule 5 of a set of terms $\mathcal{T}_\Sigma(\mathcal{Y})$ for a set of variables $\mathcal{X}$ includes the notion of change of variables, as well as the elimination and introduction of variables. In such a substitution a term of $\mathcal{T}_\Sigma(\mathcal{Y})$ must be provably of sort s if it is substituted for a variable X_s of sort s. This substitution reflects the fact that it is important to include explicit quantifiers in any specification since any variable can only be eliminated by replacing it with some term which is provably of the same sort. It is this rule that in a derivation from a normal specification restricts the introduction of terms to those that are provably of the same classification.

For example, rule 4b is illustrated with the equation

$$(\mathtt{X}'_{\mathtt{int}})\ 2 * \mathtt{X}'_{\mathtt{int}} = \mathtt{X}'_{\mathtt{int}} * 2$$

and the assertion

$$(\mathtt{X}_{\mathtt{even}}, \mathtt{X}'_{\mathtt{int}})\ \mathtt{even}(\mathtt{X}_{\mathtt{even}} + (2 * \mathtt{X}'_{\mathtt{int}}))$$

to infer

$$(\mathtt{X}_{\mathtt{even}}, \mathtt{X}'_{\mathtt{int}})\ \mathtt{even}(\mathtt{X}_{\mathtt{even}} + (\mathtt{X}'_{\mathtt{int}} * 2)).$$

Also, rule 5 is illustrated by using the assertion

$$(\mathtt{X}_{\mathtt{pos}}, \mathtt{X}'_{\mathtt{pos}})\ \mathtt{pos}(\mathtt{X}_{\mathtt{pos}} + \mathtt{X}'_{\mathtt{pos}})$$

and the substitution $\theta : \{(\mathtt{X}_{\mathtt{pos}}, \mathtt{Y}_{\mathtt{pos}}), (\mathtt{X}'_{\mathtt{pos}}, \mathtt{Y}'_{\mathtt{int}} * \mathtt{Y}'_{\mathtt{int}})\}$, where the sort of $\theta\mathtt{X}'_{\mathtt{pos}}$ is assured by the assertion

$$(\mathtt{Y}'_{\mathtt{int}})\ \mathtt{pos}(\mathtt{Y}'_{\mathtt{int}} * \mathtt{Y}'_{\mathtt{int}}),$$

to derive

$$(\mathtt{Y}_{\mathtt{pos}}, \mathtt{Y}'_{\mathtt{int}})\ \mathtt{pos}(\mathtt{Y}_{\mathtt{pos}} + (\mathtt{Y}'_{\mathtt{int}} * \mathtt{Y}'_{\mathtt{int}})).$$

These rules can also be applied to more general formulas.

Let $S = (\Sigma, \Gamma)$ be a specification with assertions Γ and suppose that we are interested in proving certain properties of the data types specified. Any assertions which can be derived using the rules of inference just given are true in *all* models. There are non-ground assertions which are true in the initial model but which are not true in other models of the specification and so cannot be derived using the ordinary rules of inference. For example, the commutative law of addition is not a consequence (via the five rules) of the specification of the natural numbers given earlier.

An induction rule is a stronger rule of inference which takes into account the special features of the initial model. The universe of the initial model contains only those data objects required to exist by the specification. In the CM system we can state that the classified elements of the initial model are exactly those which are generated by the declaration assertions in the specification. Therefore we can state a rule allowing assertions to be proved by induction on the complexity of the structure of the data objects of a given type.

Suppose that Σ is a signature, that $S = (\Sigma, \Gamma)$ is a specification, that $\mathsf{s} \in \Sigma$ is a sort symbol that appears in the assertions S and that $\mathsf{P}(\mathsf{X_s})$ is an atomic assertion involving the variable $\mathsf{X_s}$ of sort s (and possibly others, possibly of other sorts, as well). Let $\mathcal{A}_\mathsf{i} \rightarrow \mathsf{s}(\mathsf{t_i})$ for $\mathsf{i} = 1 \ldots \mathsf{n}$ be all the declaration assertions for the sort s in S, where the body $\mathcal{A}_\mathsf{i}$ of each declaration assertion is either empty or a set of atomic assertions.

1. Form a new sequence $\mathsf{t}'_1, \ldots, \mathsf{t}'_\mathsf{n}$ of terms in which each t'_i is the result of replacing all variables of sort s in $\mathsf{t_i}$ by new constant symbols $\mathsf{c}_1, \ldots, \mathsf{c}_\mathsf{m}$ not already in the signature Σ.

2. Prove, using the ordinary rules of inference, the assertions $\mathsf{P}(\mathsf{t}'_\mathsf{i})$ for $\mathsf{i} \leq \mathsf{n}$. In doing so we use S, the declarations $\mathsf{s}(\mathsf{c}_\mathsf{j})$ $(\mathsf{j} \leq \mathsf{m})$ and the induction hypotheses $\mathsf{P}(\mathsf{c}_\mathsf{j})$ $(\mathsf{j} \leq \mathsf{m})$. Having done so, we can conclude that $\mathsf{P}(\mathsf{X}_\mathsf{s})$ is true in the initial model of S.

One of the advantages of this induction rule is that it allows proofs to be formulated entirely within the object language, i.e., essentially that used by programmers. It does not require knowledge of, or reference to, metamathematical notions such as that of a homomorphism. The soundness of the CM induction rule is proved in Appendix A.

2.6 Signatures And Type Checking

An obvious difference between a semantically based method such as CM and other more syntactically based specification methods is that with the CM technique there are no operation sorts and any operation can be applied to any operand. Does this mean that we must abandon syntactic type checking? The answer is "no", but with syntactic type checking we might not realize the full potential of the CM method.

For each sort symbol s of a specification $S = (\Sigma, \Gamma)$ there is a set of expressions t for which $\mathsf{s}(\mathsf{t})$ is true in the initial model of S. Since this classification may not in general be decidable we cannot expect to have an algorithm which checks the sorts of expressions. However, for any set of declaration assertions (without equations) in Γ the syntactic classification is decidable. From any specification S we can form another set T of assertions such that:

1. each element of T is true in the initial model of S (i.e., it is true of the sorts specified);

2. the syntactic classification induced by T is decidable.

We can say that the type checking with respect to T is "partially correct". The type checker can determine that an expression is of some sort, but there might be some expressions for which the type checker might not determine the least sort. For example, some value is always an integer even though the best that the type checker can do is classify it as of sort real. Fortunately, partial sort information is sufficient in many applications.

The CM technique allows the formation of expressions in an arbitrary fashion that is foreign to the more common strongly typed languages. For example, a queue can be added and integers can have a `front` operation applied. A language implementer does not have to provide a programmer with the full CM capability since the implementer could always select some set T (as described above) and require that expressions in a program be "classifiable" according to T. For example, given the OBJ2 (an OSA language) goal of executing specifications, it is probably appropriate to restrict the type checking to the syntactic form. Thus, OBJ2 can be considered a CM language in which an additional constraint has been imposed by the language designer. If the goals of a language include performing derivations about the properties of programs, it might be appropriate to adopt the general CM approach. The strength of a CM type discipline depends inversely on that of T since the formal CM system does not prescribe the strength but leaves the decision with the language designer (where it belongs). Milner's ML language has inspired much of this work.

There is a need for some restrictions on programs and specifications in order to avoid consideration of "error" terms. This is the purpose of "normal" specifications in which each term that appears in the specification can be classified in

the initial model. In combination with the rules of inference and the induction rule we are assured that unsorted terms cannot appear in a proof since none of them appear in the specification and they cannot be introduced by the rules.

2.7 Related Work: Order Sorted Techniques

A language implementer can, as described in the previous section, select a type mechanism that is more strict than the mechanism described for the Classified Model concept. For example, a syntactic signature and a corresponding strong typing mechanism can be chosen because of language implementation constraints such as parsing considerations. An OSA language such as OBJ could be considered an implementation of CM because it restricts the language to a manageable subset that can be implemented.

The classified approach was intended for more than just execution of specifications. This broader purpose includes execution as well as deduction that can be used, for example, in ML-style type checking or for proof of specification properties.

The following subsections compare OSA and CM by first focusing on their similarities and then upon the characteristics that distinguish them.

2.7.1 Similarities Between OSA And CM

Both OSA and CM are motivated by the desire to solve fundamental problems with MSA in the areas of overloaded operators and the treatment of errors. Both techniques find solutions in the introduction of subsorts, although they differ in the method of specifying subsorts. In this dissertation it is shown that OSA and CM can use similar rules of inference, although CM is formalized in a different

setting with proofs similar to the published OSA formalization. In [GM89] it is shown that a version of OSA having a universal sort, similar to the CM approach, encompasses the published versions of OSA.

Parameterization issues are identical for OSA and CM since both are liberal institutions. This could be called "specification-in-the-large" because it relates the ways that components of a larger specification are glued together. The construction of basic specifications by either OSA or CM techniques could be called "specification-in-the-small". The CM technique described in this chapter supplies just "specification-in-the-small" syntax, whereas both types are included in OSA languages such as OBJ. Like the parameterization issue, both OSA and CM have the same module (ADTs with hidden states) concepts.

2.7.2 Differences Between OSA And CM

OSA is an extension of MSA, whereas the classified method is not constrained to just a syntactic assignment of sorts to terms. That is, the classified type system, which includes the MSA and OSA syntactic typing as a special case, does not determine the sort of an expression strictly from the type of its subexpressions. Instead, sorts are determined according to assertions of the specification language.

Subsort declarations differ in OSA and CM according to the fundamental difference in their orientation. Subsort declarations of OSA are part of the syntactic signature, whereas they are included in the assertions of the classified approach. The counterpart of OSA sort constraints is the CM use of sort predicates in the antecedent of a formula to constrain the use of some sort. For example, an OSA sort constraint is applied in a bounded stack signature for push to limit the number of push terms. The CM counterpart uses an antecedent formula in a declaration

assertion. Subsorts in the CM technique are usually declared inductively independently of the supersort, because this often results in simpler derivations of properties of the subsort. This is in contrast to the OSA practice of defining each sort in terms of its sub-sorts, although this can be derived from the inductive form.

A fundamental difference between models of OSA and CM is that a CM initial model contains terms that do not exist in the corresponding OSA initial model. These are "error" terms that correspond to unintended uses of the operations. In an OSA initial model some of these terms occur in "error supersorts" of the declared sorts, whereas in the CM technique *any* operation can be applied to *any* term(s). In the CM technique we can also declare error supersorts for the anticipated errors. The arbitrary application of operations to term(s) appears at first glance to present a problem. These terms, however, are not permitted in derivations from normal specification using the rules of inference and induction rule presented in this chapter.

2.7.3 Unique Aspects Of Order Sorted Methods

The order sorted methods have the following[GM87b] characteristics:

Monotonicity condition: Suppose an operation "op" is "overloaded" by having both the declaration $\mathsf{op} : \mathsf{s}_1, \ldots, \mathsf{s}_n \rightarrow \mathsf{s}$ and the declaration $\mathsf{op} : \mathsf{s'}_1, \ldots, \mathsf{s'}_n \rightarrow \mathsf{s'}$. Suppose also the subsort partial order includes $\mathsf{s}_i \leq \mathsf{s'}_i$ for $i = 1, \ldots, n$. Then for the specification to satisfy the monotonicity condition it must be the case that $\mathsf{s} \leq \mathsf{s'}$.

Regular signature: Each term has a defined least sort.

The monotonicity condition ensures consistency between subsort declarations and operation declarations. That is, it ensures that the operation declarations do not imply a subsort relation contrary to the subsort declarations. In the CM system the monotonicity condition is unnecessary because subsort relations are inferred from the term sorts that are declared by the Horn assertions, so there can be no conflict in declarations. In the CM system the best we can do with regard to regularity is to prove some sort for each term, but it might not be the least sort.

2.8 Summary

The classified model specification technique has a defined syntax and semantics that has been related (in an appendix) by theorems showing soundness and completeness results and the existence of a free model for any classified model specification. Furthermore, there is an induction rule that can be used to prove results in the initial term model.

Chapter 3

Parameterized module specification

The main purpose of the Classified Model approach to the specification of Abstract Data Types is to allow explicit reasoning about the types of expressions. This is described in the previous chapter in the form of basic specifications that are meant to be read and understood independently of any other specifications. This basic form is adequate for short specifications, but Burstall and Goguen[BG77] argue that large specifications should be glued together from smaller specifications in order to break the larger specification into smaller understandable parts. The intention is that if each of the smaller parts can be understood and if the mechanism for gluing them together can be understood then the meaning of a larger specification can be constructed from the meaning of its parts. Their specification building process is independent of the basic specification technique, provided the basic technique has a "free construction" which makes it a liberal institution. Furthermore, the specifications resulting from such a construction have an initial model (and free model) semantics.

Goguen and Burstall state in [GB84] that first-order Horn logic with equality is a liberal institution. The Classified Model logic is easily identified with this liberal institution by simply relativizing the CM language. This is done by replacing the sorted variables of each assertion by unsorted variables and simultaneously placing a unary sort antecedent for each such variable in the assertion. This means that classified model specifications can be combined (glued together) to

form new specifications which have an initial model semantics. The techniques for combining theories used here are those pioneered by Burstall and Goguen[BG77] and developed for the specification languages Clear and OBJ. A concrete syntax for a language to combine specifications is avoided by applying directly to the basic specification style already demonstrated a simple syntax for the parameterization statements in a language like OBJ[FGMO87].

3.1 Parameterized MSA ADT Specifications

Parameterized specifications such as `STACK(DAT)` of figure 3.1 consist of an MSA specification with a formal parameter, or requirements, sub-specification such as `DAT`. The requirements specification is *actualized* with some actual specification such as `NAT` (the natural numbers) to form a specification such as `STACK(NAT)`. The actualization is specified with a binding which provides a correspondence from the signature of the requirements specification to the signature of the actual specification such that all the assertions of the requirements specification under the correspondence are valid in the initial model of the actual specification.

For example, the parameterized MSA specification `STACK(DAT)` of figure 3.1 contains the formal parameter sub-specification `DAT` consisting of just the signature `d` $:\rightarrow$ `dat` having just the sort `dat` and the constant `d` and no other operations or equations. The correspondence from the `DAT` specification to a specification `NAT` of the natural numbers is just the correspondence from the sort name `dat` to the sort name `nat` and the correspondence from the `dat` constant `d` to the `nat` constant 0.

The syntactic meaning of `STACK(NAT)` is the union of a `NAT` specification and the previous `STACK-OF-NAT` specification, assuming the actual specification and

```
stknil :→ stk
push : stk, dat → stk
top : stk → dat
pop : stk → stk

top(stknil) = d
top(push(S, I)) = I
pop(stknil) = stknil
pop(push(S, I)) = S
```

Figure 3.1: Parameterized MSA stack specification STACK(DAT)

the non-parameter part of the parameterized specification have no names in common. The semantics of STACK(NAT) is the same as the semantics of the union of NAT and STACK-OF-NAT, i.e., the initial algebra. It is derived from the semantics of STACK(DAT), which is more complicated.

The semantics of a parameterized specification such as STACK(DAT) cannot be based upon the initial algebra of the specification because it might happen that there are no constants in either the parameterized specification or the DAT requirements sub-specification and therefore the generated initial algebra might be empty. To remedy this, many authors (e.g., [BG77]) consider the meaning of a parameterized specification to be a transformation of *any* actual algebra (one that can be substituted for the requirements specification) to an algebra of the target specification. This target algebra can be constructed to be free and therefore also initial[EM85] if the substituted algebra is also free.

A special case of parameterization is given by the *enrichment* construction demonstrated in the semigroup, monoid and group example. In this case the

specification correspondence is an inclusion and the meaning of the monoid specification is based upon the meaning of the semigroup specification by the transformation described above. We can also have requirement specifications that are themselves parameterized, e.g., group is parameterized by monoid and monoid is parameterized by semigroup. This construction has a property that the order of substitution is not important.

Enrichment is often used (as in the semigroup, monoid and group example) to write a specification in terms of one or more other specifications. This technique is demonstrated in a classified model specification of a keyword-in-context module that is described in terms of a parameterized queue and a parameterized sorting queue.

3.2 Parameterized Types And Type Operations

Specifications can be combined either by means of parameterization, as described above, or by type operations such as union and intersection. We could allow type variables and user-specified operations on types to appear within specifications, but we do not, because many of the requirements of specification building can be met by separating the basic first order classified module specifications from the ways of combining them. Such a two level specification language is investigated by Wadge[Wad90]. The specification combination operations described in this chapter are a subset of operations that are already in use for other liberal institutions such as MSA. These operations are independent of the underlying first order specification technique (such as MSA or CM) and they are defined so that they transform some first order parameter specification into an appropriate first order target specification. Other operations not discussed here include placing

other types of constraints upon the formal parameter specification.

The operations on basic (and combined) specifications are:

Composition: Like function composition, in which two specifications can be hooked together because one specification supplies the requirements of a second specification. That is, the second specification is written in terms of the first specification.

Actualization: Like function application, in which an actual specification replaces a formal requirements specification within a larger specification.

Union: Like set union, in which the meaning is the union of the meanings of the individual specifications.

3.3 The Parts Of A Parameterized Module Specification

Each module specification (Goguen[Gog86], [BEPP87]) MOD consists of four partial or whole CM specifications called parameter (PAR), export interface (EXP), import interface (IMP) and body (BOD) which are related by the diagram of Figure 3.2 in which each arrow represents a correspondence between the signature of the specification part at the start of the arrow and some part of the signature of the specification part at the point of the arrow. When symbols are translated according to the correspondence, truth in the specification at the start of the arrow must be preserved in the specification at the point of the arrow.

The four parts are described briefly here and demonstrated in Figure 3.2. The export interface (EXP) reveals only the operations and sorts available to the module user and consequently defines the hidden state sorts of the module as those of BOD which do not have a counterpart in EXP via the specification

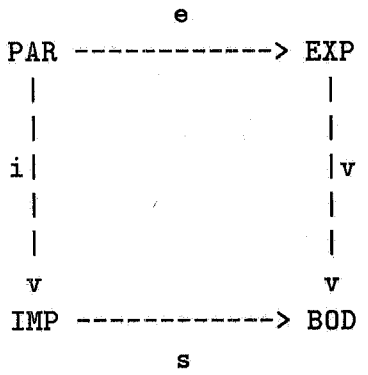


Figure 3.2: Module Specification Parts

correspondence v. This corresponds to the definition of a module as an abstract data type in which some of the sorts have been designated as hidden such that expressions of those sorts cannot be evaluated and are therefore hidden from observation. These are distinguished from the visible sorts which exhibit the observable behavior of a module as a result of evaluation of terms of visible sort.

The import interface (IMP) is also defined by a specification correspondence as the set of operations required to complete the specification BOD. That is, the specification BOD is written in terms of IMP in the sense that BOD refers to operations of IMP, but the operations are defined in IMP.

The body part (BOD) is the main part of the CM specification and is complete except perhaps for the IMP part.

The parameter part (PAR), defined by specification correspondences to IMP and EXP, represents the part common to the import and export sections. It is distinguished from the IMP part by the capability of actualizing this requirements part by any specification which does not violate its axioms, i.e., there is a specification correspondence from the PAR requirements to the actual specification.

For example:

1. A parameterized ADT specification corresponds to the case of the vertical arrows representing identity specification correspondences, resulting in just a single horizontal arrow. This case is described in detail in [EM85] and corresponds to the MSA specification for `STACK(DAT)` in the introductory part of this chapter.

2. A non-parameterized module specification corresponds to the case of the horizontal arrows representing identity specification correspondences. For example, any basic CM specification such as `STACK-OF-NAT` can be considered a non-parameterized module specification if some sort such as `stk` is considered hidden such that terms of that sort are not available for evaluation.

3. A parameterized module CM specification example is `STACK(DAT)`, below:

   PAR: A specification `DAT` consisting of just one sort name "`dat`". The inclusion correspondence goes to EXP and IMP. This requirement is easily met by any specification with at least one sort. Recall that a specification is a pair consisting of a signature of operation and predicate names and a set of assertions (possibly empty) of that signature.

   IMP: The specification depends only upon the import specification for sort name `dat` so IMP is the same as PAR and is defined by the identity correspondence from PAR.

   EXP: A specification with the sort names `dat` and `stk` and operation names `push`, `pop` and `top` in the export interface. The specification correspondence is the inclusion to BOD. Neither `stknil` nor any other term of sort `stk` is exported.

BOD: The stack classified model specification, given in a previous chapter, but in a relativized form here:

$$\texttt{stk(stknil)}$$

$$\texttt{(S,X) stk(S),dat(X)} \rightarrow \texttt{stk(push(S,X))}$$

$$\texttt{(S,X) stk(S),dat(X)} \rightarrow \texttt{pop(push(S,X))} = \texttt{S}$$

$$\texttt{(S,X) stk(S),dat(X)} \rightarrow \texttt{top(push(S,X))} = \texttt{X}$$

The specification correspondences above are simply the inclusion, but this can be generalized to include also renaming correspondences which are required to avoid unintentional name conflicts.

3.4 Specification Combining Operations

Following [BEPP87], three simple ways are given to combine specifications to form larger specifications. In this section some simple syntax corresponding to specification building operations is introduced to facilitate the expression of combinations of the operations.

Composition: `MOD1 · MOD2` designates a new module which results from matching the export interface of `MOD2` with the import interface of `MOD1`. This corresponds to the idea of defining `MOD1` in terms of `MOD2` and is expressed by a specification correspondence from `MOD1` to `MOD2` that expresses the way that `EXP2`, the export component of `MOD2`, meets the requirements of `IMP1`, the import component of `MOD1`. Additionally, there is a `PAR1` to `PAR2` specification correspondence. The new composite module has the same export interface and parameter part as `MOD1`, the same import interface as `MOD2` and a body consisting of the body part of `MOD1` with `IMP1` replaced by `BOD2`,

the body of MOD2.

Actualization: A new (parameterized) specification results from the substitution of an actual (parameterized) data type specification for the parameter part PAR of a module specification MOD via a specification correspondence from PAR (the requirements, or formal specification) to the body of the actual specification. If the actual specification is parameterized then the parameter part of the new specification is the parameter part of the actual specification, otherwise it is empty. The other parts of the new specification are the union of the corresponding parts of the actual specification and the specification MOD.

Union: MOD1 + MOD2 designates the union of modules, where any common submodules are represented only once in the union.

For example, the parameterized specification STACK(DAT) can be actualized with NAT and expanded according to the construction descriptions for union and actualization, i.e., the expanded specification is STACK-OF-NAT union NAT.

3.5 An Example: Key Word In Context

The specification of a "kwic" (keyword in context) index generation module has often been described in the literature as the task of accepting as input a set of, for example, book, titles and producing a sorted list of all the rotations of all input titles. A variation not considered here is to eliminate some rotations that start with common words such as "and", "of", "the" etc.

The following representation of the problem reflects fundamental assumptions regarding the sequential nature of words within lines and sorted lines in the kwic

index. These assumptions are reflected within the interface, first informally in the EXP list below and then formally in the BOD specification. Note that the difference between the EXP and BOD parts is that only expressions of the operations on the EXP interface can be evaluated by the user of the module, although he or she is allowed to read the BOD part to determine the meaning of the EXP operations.

Since titles consist of a sequence of atomic words and the words are rotated, each title is conveniently represented by a queue of words, which is easily rotated given the usual queue operations. Since titles are sequentially presented to the kwic module and sequentially extracted in sorted order from the module, the titles (queues of words) are arranged in order by a sorting queue. A sorting queue is really just a general purpose sorting module in which the results are returned (and removed) one-at-a-time instead of in the familiar sorted array. This provides flexibility for the implementer since not only is the sorting algorithm not specified, but neither is the sort-time, i.e., it is not specified that all input items must be maintained in sorted order at all times or partially or fully sorted just before output. Other representations have been used in the literature to describe kwic, but the above is based upon the fundamental assumptions.

3.5.1 Kwic Module Specification

The kwic module specification is:

PAR: The ADT NAT given in the introductory chapter and a specification for an ADT WORD having a partial order of its objects. These are related by the inclusion correspondence to EXP and IMP. The final specification is actualized with an actual specification for WORD such that the usual axioms for a partial order are satisfied.

IMP: The union of the specifications of sorting queue of line and a queue of word (to buffer words of an incomplete line), both of which also contain specification `NAT` for their "length" functions. These are given in the following sections.

EXP: The exported sorts are `nat` and `word` (and their operations). The exported kwic operations (with informal descriptions) are:

1. `StartLineIn` indicates that a new line is started for input.
2. `SetWord` supplies an additional word sequentially to the new line.
3. `EndLineIn` indicates that an input line is complete.
4. `GetLine` extracts the alphabetically first line from the kwic index.
5. `RemoveLine` removes the alphabetically first line from the kwic index.
6. `GetWord` returns the first word from the most recently extracted line.
7. `RemoveWord` removes the first word from the most recently extracted line.
8. `LineLen` returns the current length of the most recently extracted line.

The specification correspondence to BOD is the inclusion.

BOD: The body of the specification is the following classified model specification, where "<" and ">" are constructors for a tupling operation for objects of sort `st` (for state). Since the state sort is hidden, any convenient representation will suffice. The state sort consists of a pair having first part a sorting queue and second part a queue. The queue operations `add`, `remove`, `front` and `qlen` are defined in a parameterized queue specification of the

next section, as are the corresponding operation names **sadd**, **sremove** and **sfront** for a parameterized sorting queue.

K0: (S_{sq}, Q_q) $\mathtt{st}(<S_{sq}, Q_q>)$

K1: (S_{sq}, Q_q) $\mathtt{SetLineIn}(<S_{sq}, Q_q>) = <S_{sq}, \mathtt{qnil}>$

K2: (S_{sq}, Q_q, W_{wd}) $\mathtt{Setword}(<S_{sq}, Q_q>, W_{wd}) = <S_{sq}, \mathtt{add}(Q_q, W_{wd})>$

K3: (S_{sq}, Q_q) $\mathtt{EndLineIn}(<S_{sq}, Q_q>) = <\mathtt{kwicin}(S_{sq}, Q_q, \mathtt{qlen}(Q_q)), Q_q>$

K4: (S_{sq}, Q_q) $\mathtt{GetLine}(<S_{sq}, Q_q>) = <S_{sq}, \mathtt{sfront}(S_{sq})>$

K5: (S_{sq}, Q_q) $\mathtt{RemoveLine}(<S_{sq}, Q_q>) = <\mathtt{sremove}(S_{sq}), Q_q>$

K6: (S_{sq}, Q_q) $\mathtt{GetWord}(<S_{sq}, Q_q>) = \mathtt{front}(Q_q)$

K7: (S_{sq}, Q_q) $\mathtt{RemoveWord}(<S_{sq}, Q_q>) = <S_{sq}, \mathtt{remove}(Q_q)>$

K8: (S_{sq}, Q_q) $\mathtt{LineLen}(<S_{sq}, Q_q>) = \mathtt{qlen}(Q_q)$

K9: (S_{sq}, Q_q) $\mathtt{kwicin}(S_{sq}, Q_q, 0) = S_{sq}$

K10: (S_{sq}, Q_q, N_{nat}) $\mathtt{kwicin}(S_{sq}, Q_q, N_{nat} + 1) =$
$\mathtt{sadd}(\mathtt{kwicin}(S_{sq}, \mathtt{rotate1}(Q_q), N_{nat}), \mathtt{rotate1}(Q_q))$

K11: (Q_q) $\mathtt{rotate1}(Q_q) = \mathtt{sadd}(\mathtt{remove}(Q_q), \mathtt{front}(Q_q))$

A concrete specification is constructed by actualizing the parameter part of the above with STRING-OF-CHAR for WORD and the standard specification for NAT.

The BOD part of the specification contains operations **kwicin** and **rotate1** which do not appear in EXP. The operation **rotate1** rotates a queue of words by one word. The **kwicin** function is used to express the way that a queue of words is put into the kwic module. For example, to understand an application of the **kwicin** function, abbreviate the three element queue $\mathtt{add}(\mathtt{add}(\mathtt{add}(\mathtt{qnil}, \mathtt{b}), \mathtt{c}), \mathtt{a})$ by bca and compute:

$\mathtt{sfront}(\mathtt{kwicin}(\mathtt{sqnil}, \mathtt{bca}), 3)$

```
sfront(sadd(kwicin(sqnil,cab,2),cab))
sfront(sadd(sadd(kwicin(sqnil,abc,1),abc),cab))
sfront(sadd(sadd(sadd(kwicin(sqnil,bca,0),bca),abc),cab))
sfront(sadd(sadd(sadd(sqnil,bca),abc),cab)
abc
```

3.5.2 Parameterized Queue ADT Specification

A parameter specification defines a sort qat as the set of all the objects, in some theory with equality, that will be put in a queue.

The sorts defined below are: q is the set of all queues and nq is the set of all non-nil queues. Also, qnil is the distinguished *nil* queue.

Q1: $\texttt{q(qnil)}$	(q base)
Q2: $(\texttt{Q}_\texttt{q}, \texttt{X}_\texttt{qat})$ $\texttt{q(add}(\texttt{Q}_\texttt{q}, \texttt{X}_\texttt{qat}))$	(q generator)
Q3: $(\texttt{X}_\texttt{qat})$ $\texttt{nq(add(qnil,}\texttt{X}_\texttt{qat}))$	(nq base)
Q4: $(\texttt{Q}_\texttt{nq}, \texttt{X}_\texttt{qat})$ $\texttt{nq(add}(\texttt{Q}_\texttt{nq}, \texttt{X}_\texttt{qat}))$	(nq generator)
Q5: $(\texttt{X}_\texttt{qat})$ $\texttt{remove(add(qnil,}\texttt{X}_\texttt{qat})) = \texttt{qnil}$	(restore nil)
Q6: $(\texttt{Q}_\texttt{nq}, \texttt{X}_\texttt{qat})$ $\texttt{remove(add}(\texttt{Q}_\texttt{nq}, \texttt{X}_\texttt{qat})) = \texttt{add(remove}(\texttt{Q}_\texttt{nq}), \texttt{X}_\texttt{qat})$	(remove front)
Q7: $(\texttt{X}_\texttt{qat})$ $\texttt{front(add(qnil,}\texttt{X}_\texttt{qat})) = \texttt{X}_\texttt{qat}$	(single front)
Q8: $(\texttt{Q}_\texttt{nq}, \texttt{X}_\texttt{qat})$ $\texttt{front(add}(\texttt{Q}_\texttt{nq}, \texttt{X}_\texttt{qat})) = \texttt{front}(\texttt{Q}_\texttt{nq})$	(same front)
Q9: $\texttt{qlen(qnil)=0}$	(nil zero)
Q10: $(\texttt{Q}_\texttt{q}, \texttt{X}_\texttt{qat})$ $\texttt{qlen(add}(\texttt{Q}_\texttt{q}, \texttt{X}_\texttt{qat})) = \texttt{qlen}(\texttt{Q}_\texttt{q}) + 1$	(non-nil successor)

The usual axioms for a partial order could also be supplied so that a queue could be used as the parameter type for sorting queue.

3.5.3 Parameterized Sorting Queue ADT Specification

The set of objects to put in a sorting queue is defined in a parameter specification for a sort **sat** (the set of all atoms) in some theory with equality, $\leq$ and $<$.

The sorts defined below are: **sq** is the set of all sorting queues, **sq1** is the set of all sorting queues with at least one element, and **sq2** is the set of all sorting queues with at least two elements. Also, **sqnil** is the distinguished *nil* sorting queue.

SQ1: $\mathtt{sq}(\mathtt{sqnil})$ (**sq** base)

SQ2: $(\mathtt{Q_{sq}}, \mathtt{X_{sat}})\ \mathtt{sq}(\mathtt{sadd}(\mathtt{Q_{sq}}, \mathtt{X_{sat}}))$ (**sq** generator)

SQ3: $(\mathtt{X_{sat}})\ \mathtt{sq1}(\mathtt{sadd}(\mathtt{sqnil}, \mathtt{X_{sat}}))$ (**sq1** base)

SQ4: $(\mathtt{Q_{sq1}}, \mathtt{X_{sat}})\ \mathtt{sq1}(\mathtt{sadd}(\mathtt{Q_{sq1}}, \mathtt{X_{sat}}))$ (**sq1** generator)

SQ5: $(\mathtt{X_{sat}}, \mathtt{Y_{sat}})\ \mathtt{sq2}(\mathtt{sadd}(\mathtt{sadd}(\mathtt{sqnil}, \mathtt{X_{sat}}), \mathtt{Y_{sat}}))$ (**sq2** base)

SQ6: $(\mathtt{Q_{sq2}}, \mathtt{X_{sat}})\ \mathtt{sq2}(\mathtt{sadd}(\mathtt{Q_{sq2}}, \mathtt{X_{sat}}))$ (**sq2** generator)

SQ7: $(\mathtt{X_{sat}})\ \mathtt{sremove}(\mathtt{sadd}(\mathtt{sqnil}, \mathtt{X_{sat}})) = \mathtt{sqnil}$ (restore sqnil)

SQ8: $(\mathtt{Q_{sq1}}, \mathtt{X_{sat}})\ \mathtt{sfront}(\mathtt{Q_{sq1}}) \leq \mathtt{X_{sat}} \rightarrow$
$\mathtt{sremove}(\mathtt{sadd}(\mathtt{Q_{sq1}}, \mathtt{X_{sat}})) = \mathtt{Q_{sq1}}$ (remove sadd)

SQ9: $(\mathtt{Q_{sq1}}, \mathtt{X_{sat}})\ \mathtt{sfront}(\mathtt{Q_{sq1}}) > \mathtt{X_{sat}} \rightarrow$ (remove front)
$\mathtt{sremove}(\mathtt{sadd}(\mathtt{Q_{sq1}}, \mathtt{X_{sat}})) = \mathtt{sadd}(\mathtt{sremove}(\mathtt{Q_{sq1}}), \mathtt{X_{sat}})$

SQ10: $(\mathtt{X_{sat}})\ \mathtt{sfront}(\mathtt{sadd}(\mathtt{sqnil}, \mathtt{X_{sat}})) = \mathtt{X_{sat}}$ (single-front)

SQ11: $(\mathtt{Q_{sq1}}, \mathtt{X_{sat}})\ \mathtt{sfront}(\mathtt{Q_{sq1}}) \leq \mathtt{X_{sat}} \rightarrow$
$\mathtt{sfront}(\mathtt{sadd}(\mathtt{Q_{sq1}}, \mathtt{X_{sat}})) = \mathtt{X_{sat}}$ (sadd-front)

SQ12: $(\mathtt{Q_{sq1}}, \mathtt{X_{sat}})\ \mathtt{sfront}(\mathtt{Q_{sq1}}) > \mathtt{X_{sat}} \rightarrow$
$\mathtt{sfront}(\mathtt{sadd}(\mathtt{Q_{sq1}}, \mathtt{X_{sat}})) = \mathtt{sfront}(\mathtt{Q_{sq1}})$ (same-front)

SQ13: $\mathtt{sqlen}(\mathtt{sqnil}) = 0$ (nil zero)

SQ14: $(\mathtt{Q_{sq}}, \mathtt{X_{sat}})$ $\mathtt{sqlen(sadd(Q_{sq}, X_{sat}))} = \mathtt{sqlen(Q_{sq})} + \mathtt{1}$ (non-nil successor)

3.6 Other Issues

It could be argued that the specification style described here is merely a logic program (with equality) and that the kwic example is not really a specification because it is "coded" in terms of the queue and sorting queue specifications. Any description (i.e., specification) must be expressed within some language. Simple specifications like `STACK(DAT)` require no supporting specification for descriptive purposes because they are so simple. Such specifications are the exception and in general it is better to specify small parts that can be joined using a well-defined construction with appropriate semantics. The alternative is to write monolithic specifications such as those that can be obtained by textually expanding the kwic specification expression. It can be claimed that such a monolithic specification at least depends upon no other specification, but an inspection of the result would show that all the sorting queue and queue concepts would be buried within the work in some obscure way.

Finally, it should be emphasized that the possible choices for the underlying specifications are quite broad. Another choice, discussed in Chapter 8 (Software Engineering Techniques), is described by Parnas[Par72b]. Although Parnas does not give a formal specification of a kwic module, he does give specifications for modules that would form a suitable basis for implementing the kwic module: a line holder module, a module to form circular shifts of lines and an alphabetizer module. The kwic task could also be described in terms of these *implementation* modules which were selected because of their maintainability.

3.7 Related Work

Goguen[Gog86] describes a concrete syntax that implements all of the operations of this chapter (and more). Specifically, module parameterization is referred to as horizontal composition of specifications, dependence of a body specification upon some import specification is referred to as vertical module composition and information hiding commands correspond to the export interface. Goguen describes many ways that programs can be reused and suggests that specifications glued together in this way may be directly executed, provided the equational assertions of a specification conform to certain restrictions. He also suggests that specifications can be transformed into implementations, a subject that is also addressed in this dissertation.

3.8 Summary

In summary, basic specification languages that are also a liberal institution qualify for the module parameterization techniques that are described in this chapter. Since the combining operations have a well-defined semantics that has been developed for other liberal institutions, they are assumed for the remainder of this dissertation.

The kwic classified model specification demonstrates the techniques in a simple context that require many of the module parameterization techniques.

Chapter 4

Classified Model Situational Logic

The classified model technique allows the definition of theories in a situational logic, i.e., a logic in which states are explicit objects. For example, in the kwic specification of the previous chapter the export interface does not include the state sort "st", yet exported predicate and function symbols each have a state as one of their arguments and the truth of any assertion may vary from one state to another. Recall that any sort having a prohibition upon evaluation of its terms is a state sort. Therefore, the exported operations should not require the evaluation of state terms and they should not appear within the exported operations. In this chapter it is shown how the classified model approach can be used to separate the state terms from the exported terms so that state terms are not arguments of the exported operations. This also corresponds to the way imperative languages usually treat states and can be viewed as a restriction upon the use of a term of state sort that corresponds to the restriction imposed by the EXP interface of the previous chapter. This separation corresponds to the specification of immutable sorts as visible sorts and the specification of mutable sorts as the hidden sorts of a module specification.

The support of polymorphic and incompletely specified functions within the classified model technique is the key feature of the method that allows a situational logic. For example, an alternative classified model specification of the kwic problem of the previous chapter exploits the fact that operation symbols ";" and

":" can be defined to be polymorphic and that not all instances of their application need to be classified. The infix operation symbol ";" is known[MW87b] as the *production* function because its left argument is a state, its right argument is any term constructed from a "state changing" operation and the new ";"-term is the new state. The infix operation symbol ":" is known[MW87b] as the *return* function because its left argument is a state, its right argument is any term constructed from a "value returning" operation and the new ":"-term represents a value of a non-state sort. Both of these operations are assumed to be left associative.

While axiom K0 is unchanged from the previous chapter, axiom K1 is typical of the situational translation of the kwic specification. The production function allows the separation of the state term $<S_{sq}, Q_q>$ from the *fluent* [MW87b] term `StartLineIn`. The remaining axioms are translations of the same form, where the return function ":" is used instead when a term is some non-state sort.

K0: (S_{sq}, Q_q) $\mathtt{st}(<S_{sq}, Q_q>)$

K1: (S_{sq}, Q_q) $<S_{sq}, Q_q>;\mathtt{StartLineIn} = <S_{sq}, \mathtt{qnil}>$

K2: (S_{sq}, Q_q, W_{wd}) $<S_{sq}, Q_q>;\mathtt{SetWord}(W_{wd}) = <S_{sq}, \mathtt{add}(Q_q, W_{wd})>$

K3: (S_{sq}, Q_q) $<S_{sq}, Q_q>;\mathtt{EndLineIn} = <\mathtt{kwicin}(S_{sq}, Q_q, \mathtt{qlen}(Q_q)), Q_q>$

K4: (S_{sq}, Q_q) $<S_{sq}, Q_q>;\mathtt{GetLine} = <\mathtt{sremove}(S_{sq}), \mathtt{sfront}(S_{sq})>$

K6: (S_{sq}, Q_q) $<S_{sq}, Q_q>:\mathtt{GetWord} = \mathtt{front}(Q_q)$

K7: (S_{sq}, Q_q) $<S_{sq}, Q_q>;\mathtt{GetWord} = <S_{sq}, \mathtt{remove}(Q_q)>$

K8: (S_{sq}, Q_q) $<S_{sq}, Q_q>:\mathtt{LineLen} = \mathtt{qlen}(Q_q)$

K9: (S_{sq}, Q_q) $\mathtt{kwicin}(S_{sq}, Q_q, 0) = S_{sq}$

K10: $(S_{sq}, Q_{nq}, N_{nat})$ $\mathtt{kwicin}(S_{sq}, Q_{nq}, N_{nat} + 1) =$
$\mathtt{sadd}(\mathtt{kwicin}(S_{sq}, \mathtt{rotate1}(Q_q), N_{nat}), \mathtt{rotate1}(Q_q))$

K11: (Q_{nq}) $\mathtt{rotate1}(Q_{nq}) = \mathtt{add}(\mathtt{remove}(Q_{nq}), \mathtt{front}(Q_{nq}))$

`GetLine` and `RemoveLine` of the previous chapter can be combined as the new `GetLine` shown here. Also `GetWord` and `RemoveWord` can be combined in the same way as a stack operation `pop` that both returns a value and changes the stack. All of the return values in this example are singletons, but a tuple (not the state tuple) could be returned, corresponding to several return values of a procedure. In the CM language the tupling operation can be overloaded just by using it in more then one context. Note that the above specification mixes the infix and situational forms because they are both within the CM language. Specifically, note that the production and return functions are both polymorphic and incompletely defined (although in the initial model each unclassified term has a defined meaning – itself). Specifications can also be given in a relativized language that replaces sorted quantification by sorted antecedents, as in Figure 4.1.

4.1 Modules: ADT With States As Hidden Sorts

Two Σ-models have the same visible behavior when[MG85][GM82]:

1. They have the same visible, i.e., non-hidden, sorts;

2. Any Σ-term of visible sort evaluates to the same value in each model.

An ADT specification can be considered a module specification once we distinguish which sorts have hidden representations. We can then construct *initial* and *final* [Wan79] realizations (models) of a specification as well as others intermediate to these. For example, the module specification of figure 4.1 with hidden sort `cnt` specifies a module which counts the number of characters (sort `chr`) ever presented by operation `in`. The count is accessible via operation `out`. Assume both the theory of characters (`chr`) and the theory of natural numbers (`nat`) are also available.

$$
\begin{array}{ll}
 & \mathtt{cnt(nil)} \\
\mathtt{(S,X)} & \mathtt{cnt(S), chr(X) \rightarrow cnt(in(X,S))} \\
 & \mathtt{out(nil) = 0} \\
\mathtt{(S,X)} & \mathtt{cnt(S), chr(X) \rightarrow out(in(X,S)) = out(S) + 1}
\end{array}
$$

Figure 4.1: CM Specification Of A Character Counting Module

In general[MG85] :

1. Data types are Σ-models.

   Machines have internal states.

2. Abstract data types describe values (immutable) and their operations.

   Abstract machines describe software modules (mutable) and their operations.

3. Abstract machines are Σ-models, with sorts partitioned as:

   (a) *visible sorts* which exhibit the observable behavior by evaluation of expressions of visible sort;

   (b) *internal sorts* (states) which are hidden from observation by a prohibition on evaluation of expressions of internal sort.

4. Two abstract data types are equivalent iff they are isomorphic Σ-models.

   Two abstract machines are equivalent iff they have the same *behavior* (i.e., EXP operations return the same values), but they might not be isomorphic as data types because they have different state sorts.

This means that for modules the meaning (intended model) of the specification includes for the visible sorts the members of the initial model and for the state

sort(s) the members of any suitable model. This gives the implementer of a module the freedom to choose a representation that may be "initial", "final" (the most space efficient) or, more likely, an intermediate representation that is a tradeoff between storage space and computation time. This also makes meaningless discussions regarding which of initial, final semantics or something intermediate to these is more appropriate for module implementations since they all have the same *visible* behavior.

4.2 A situational/ADT Translation

The relationship between a module specification in the situational form and a (renamed) ADT form module specification is summarized in figure 4.2 (quantifiers are omitted for space reasons). The situational module specified is a counting module that returns via operation `Out` the total number of characters ever presented to the module by the operation `In`. The ADT form classifies as sort `cnt` the set of `in` terms and associates with `out` the depth of the `cnt`-classified terms. The first two columns are a module specification in two different situational forms that can be related by the linkage[MW87b] axioms:

1. `S;In(X) = in(S:X,S;X)`

2. `S:Out = out(S)`

The unary predicate `st` classifies the "state" sort representing the terms s_0, s_0;`In(a)`, s_0;`In(a)`;`In(a)` etc. In the ADT form specification `cnt` classifies the terms `nil`, `in(a,nil)`, `in(in(a,nil),a)`, etc.

The relationship between the single-instance module situational specification and the ADT form specification is:

Situational form	Intermediate form	ADT form
$\mathtt{st}(\mathtt{s}_0)$	$\mathtt{st}(\mathtt{s}_0)$	$\mathtt{cnt}(\mathtt{nil})$
$\mathtt{st}(\mathtt{S}), \mathtt{chr}(\mathtt{X}) \rightarrow$ $\mathtt{st}(\mathtt{S};\mathtt{In}(\mathtt{X}))$	$\mathtt{st}(\mathtt{S}), \mathtt{chr}(\mathtt{X}) \rightarrow$ $\mathtt{st}(\mathtt{in}(\mathtt{S}{:}\mathtt{X}, \mathtt{S}; \mathtt{X}))$	$\mathtt{cnt}(\mathtt{S}), \mathtt{chr}(\mathtt{X}) \rightarrow$ $\mathtt{cnt}(\mathtt{in}(\mathtt{X}, \mathtt{S}))$
$\mathtt{s}_0{:}\mathtt{Out} = 0$	$\mathtt{out}(\mathtt{s}_0) = 0$	$\mathtt{out}(\mathtt{nil}) = 0$
$\mathtt{st}(\mathtt{S}), \mathtt{chr}(\mathtt{X}) \rightarrow$ $\mathtt{S};\mathtt{In}(\mathtt{X}){:}\mathtt{Out} =$ $\mathtt{S}{:}\mathtt{Out} + 1$	$\mathtt{st}(\mathtt{S}), \mathtt{chr}(\mathtt{X}) \rightarrow$ $\mathtt{out}(\mathtt{in}(\mathtt{S}{:}\mathtt{X}, \mathtt{S}; \mathtt{X})) =$ $\mathtt{out}(\mathtt{S}) + 1$	$\mathtt{cnt}(\mathtt{S}), \mathtt{chr}(\mathtt{X}) \rightarrow$ $\mathtt{out}(\mathtt{in}(\mathtt{X}, \mathtt{S})) =$ $\mathtt{out}(\mathtt{S}) + 1$

Figure 4.2: Situational - ADT conversion

1. An ADT with hidden sorts can be considered an abstract machine in which the states are the cnt terms.

2. The middle column, which can be considered a generalization of the ADT column, has extra state designators because we cannot assume that all terms are *rigid* (i.e., S:X designates the same object X in any state) nor that the production function is free of side-effects for non-state objects (i.e., S;X is the same state as S)[MW87b].

3. The left column is an equivalent form of the middle column in which the single state term is separated from the *fluent*, other part of the term, by using the linkage axioms.

A conclusion that can be drawn is: *an imperative program is a fluent term and the state term is the initial state.* This is demonstrated in the chapter 6, Program Synthesis.

4.3 Situational Stack Theory

The following is a classified model specification (in the relativized language) for a situational stack single-instance theory.

$$\mathtt{st}(\mathtt{s}_0)$$

$$(\mathtt{S},\mathtt{X}) \quad \mathtt{st}(\mathtt{S}),\mathtt{nat}(\mathtt{X}) \rightarrow \mathtt{st}(\mathtt{S};\mathtt{Push}(\mathtt{X}))$$

$$(\mathtt{S},\mathtt{X}) \quad \mathtt{st}(\mathtt{S}),\mathtt{nat}(\mathtt{X}) \rightarrow \mathtt{S};\mathtt{Push}(\mathtt{X});\mathtt{Pop} = \mathtt{S}$$

$$(\mathtt{S},\mathtt{X}) \quad \mathtt{st}(\mathtt{S}),\mathtt{nat}(\mathtt{X}) \rightarrow \mathtt{S};\mathtt{Push}(\mathtt{X}):\mathtt{Top} = \mathtt{X}$$

An extension of the situational stack specification comprises the *linkage axioms* for converting between the situational and (relativized) ADT form.

$$(\mathtt{S},\mathtt{X}) \quad \mathtt{st}(\mathtt{S}),\mathtt{nat}(\mathtt{X}) \rightarrow \mathtt{S};\mathtt{Push}(\mathtt{X}) = \mathtt{push}(\mathtt{X},\mathtt{S})$$

$$(\mathtt{S},\mathtt{X}) \quad \mathtt{st}(\mathtt{S}),\mathtt{nat}(\mathtt{X}) \rightarrow \mathtt{S};\mathtt{Pop} = \mathtt{pop}(\mathtt{S})$$

$$(\mathtt{S},\mathtt{X}) \quad \mathtt{st}(\mathtt{S}),\mathtt{nat}(\mathtt{X}) \rightarrow \mathtt{S};\mathtt{Push}(\mathtt{X}):\mathtt{Top} = \mathtt{top}(\mathtt{push}(\mathtt{X},\mathtt{S}))$$

A multiple instance stack specification distinguishes among several instances of the type-of-interest object, which could be introduced by declaration assertions as shown in the first two assertions below. These assertions declare names for two different stacks called a and b, which are distinguished from the initial, joint state $\mathtt{s}_0$ of *both* stacks.

$$\mathtt{stk}(\mathtt{a})$$

$$\mathtt{stk}(\mathtt{b})$$

$$\mathtt{st}(\mathtt{s}_0)$$

$$(\mathtt{W},\mathtt{S},\mathtt{X}) \quad \mathtt{st}(\mathtt{W}),\mathtt{stk}(\mathtt{S}),\mathtt{nat}(\mathtt{X}) \rightarrow \mathtt{st}(\mathtt{W};\mathtt{Push}(\mathtt{S},\mathtt{X}))$$

$$(\mathtt{W},\mathtt{S},\mathtt{X}) \quad \mathtt{st}(\mathtt{W}),\mathtt{stk}(\mathtt{S}),\mathtt{nat}(\mathtt{X}) \rightarrow \mathtt{W};\mathtt{Push}(\mathtt{S},\mathtt{X});\mathtt{Pop}(\mathtt{S}) = \mathtt{W}$$

$$(\mathtt{W},\mathtt{S},\mathtt{X}) \quad \mathtt{st}(\mathtt{W}),\mathtt{stk}(\mathtt{S}),\mathtt{nat}(\mathtt{X}) \rightarrow \mathtt{W};\mathtt{Push}(\mathtt{S},\mathtt{X}):\mathtt{Top}(\mathtt{S}) = \mathtt{X}$$

Frame axioms such as the first two below are given explicitly for each of the finite number of stacks under consideration. These are given explicitly just this time and are assumed in subsequent examples. The Top function is usually assumed to be side-effect free, as shown in the third axiom. The Pop function can be specified to have a return value by the fourth axiom, although this isn't necessary when a separate Top function is given. By using the production and return functions we can specify a Pop that both returns a value and has an effect.

$$(\mathtt{W},\mathtt{S},\mathtt{X})\ \mathtt{st(W)},\mathtt{nat(X)} \rightarrow \mathtt{W};\mathtt{Push(a,X)};\mathtt{Pop(b)} = \mathtt{W};\mathtt{Pop(b)};\mathtt{Push(a,X)}$$

$$(\mathtt{W},\mathtt{S},\mathtt{X})\ \mathtt{st(W)},\mathtt{nat(X)} \rightarrow \mathtt{W};\mathtt{Push(b,X)};\mathtt{Pop(a)} = \mathtt{W};\mathtt{Pop(a)};\mathtt{Push(b,X)}$$

$$(\mathtt{W},\mathtt{S},\mathtt{X})\ \mathtt{st(W)},\mathtt{stk(S)},\mathtt{nat(X)} \rightarrow \mathtt{W};\mathtt{Top(S)} = \mathtt{W}$$

$$(\mathtt{W},\mathtt{S},\mathtt{X})\ \mathtt{st(W)},\mathtt{stk(S)},\mathtt{nat(X)} \rightarrow \mathtt{W};\mathtt{Push(S,X)}:\mathtt{Pop(S)} = \mathtt{X}$$

4.4 Related Work

Goguen and Meseguer describe FOOPLog[GM87c] which combines, and derives its name from, equational logic (a functional part), object-oriented modules (the OOP part) and Horn logic (the Log part). The semantics of the object-oriented part is call "reflective semantics".

Trace theory, originally described by Bartussak and Parnas[PB78], is a *full* first order axiomatic theory, about "traces", or sequences of symbols that represent imperative programming language function and procedure calls having value parameters. The relationship between trace theory and classified model situational logic is noted after a brief description of trace theory.

In trace theory, program calls from a defined set are concatenated to form a trace which represents an execution history of the corresponding functions and procedures. Also included among the traces are the calls (sequences of length one),

Signature:

$\texttt{push(char)}$
$\texttt{pop}$
$\texttt{top} \rightarrow \texttt{char}$

Axioms:

$(\texttt{T}, \texttt{X}_{\texttt{char}})$ $\texttt{L(T)} \rightarrow \texttt{L(T.push(X}_{\texttt{char}}\texttt{))}$
$(\texttt{T})$ $\texttt{L(T.top)} \leftrightarrow \texttt{L(T.pop)}$
$(\texttt{T}, \texttt{X}_{\texttt{char}})$ $\texttt{T.push(X}_{\texttt{char}}\texttt{).pop} \equiv \texttt{T}$
$(\texttt{T})$ $\texttt{L(T.top)} \rightarrow \texttt{(T.top} \equiv \texttt{T)}$
$(\texttt{T}, \texttt{X}_{\texttt{char}})$ $\texttt{L(T)} \rightarrow \texttt{V(T.push(X}_{\texttt{char}}\texttt{).top)} = \texttt{X}_{\texttt{char}}$

Figure 4.3: Trace Specification - Unbounded Stack

and the empty trace, which is not considered a call. Not necessarily all possible traces match the intended set of sequential function and procedure execution histories, so a subset of "legal" traces is characterized by a unary predicate "L". Similarly, only programming language functions return a value, so a unary partial function "V" is defined on traces having a function call suffix to designate the value returned by the function call after various traces.

For example[PB78], an unbounded stack of character module has procedure and function calls with the syntax section shown in figure 4.3. The syntax section is information for the programmer and has no semantic counterpart like the syntax section of the MSA and related techniques, i.e., it does not define the carriers of a data type.

The axioms are in a sorted full first order logic with the convention that the variable "T" is of sort trace, the symbol "." is a polymorphic infix trace (or call) concatenation function yielding a new trace. That is, traces, including calls and the empty trace, form a monoid with the concatenation operation and empty trace identity.

The correspondence between trace theory and classified model situational logic is:

- The "trace" sort corresponds to the classified sort "state".
- The "L" predicate assertions for trace legality corresponds to the CM declaration assertions defining the CM state sort.
- The trace "V" function corresponds to the the situational CM return function ":".
- The trace "." concatenation function corresponds to the situational CM production function ";".

As described in the introductory chapter, the CM restriction to Horn formulas is not a constraint on the power of a specification language if we are interested in theories which admit initial models. Recall that initial term models are generated (every element of the domain has a name) and generic (any other model is a special case of a generic model). Thus, if we are interested in specification of modules having initial term models then we do not need a full first order logic.

4.5 Summary

In summary, the situational form of a specification such as a stack is a legal CM specification which has an advantage over the ADT form of specification, namely separating *state* terms from *fluent* terms. This separation allows independent consideration of the state, or mutable, part of a specification from the immutable part, which in the case of multiple instance specifications includes instance identifiers. Subsequent chapters showing applications of CM specifications use the separation to justify a Hoare logic which generalizes from simple variables

to modules and a synthesis technique for generating an implementation of a program from its specification and the specification of modules which are to be used in the implementation.

Chapter 5

Situational Hoare Logic

The program verification method of Hoare is a well-known technique for proving the partial correctness of simple imperative programs. Unfortunately, the method does not scale-up well to include all combinations of features commonly found in practical imperative languages [Apt81]. A safe language subset includes the basic control operations of selection, repetition and sequential execution of program statements as well as the state changing assignment statement. Many of the problems with the basic method are rooted in the use of procedures with return values and with the unrestricted use of functions[ACH76].

The Hoare calculus is obtained from the earlier method of Floyd[Flo67] by restricting transfer of control to the above control structures and by eliminating the "goto" statement. In a similar way, some of the procedure and function problems of the Hoare method may be overcome by restricting the use of procedure and function calls to those defined in a classified model specification. Although this might seem too restrictive, just as the elimination of the goto statement was once viewed as radical, the software engineering technique described later supports just this style of programming.

A simple example of program verification by Hoare's method is used to demonstrate the standard technique for ordinary integer variables. Following this, the same example is used to demonstrate the extended Hoare method when the variables are specified by situational classified model assertions.

An example based on a CM situational stack theory is used to demonstrate the extension of Hoare logic for use with modules. This example is also used in the next chapter to illustrate the synthesis of programs from module specifications.

5.1 The Basic Hoare Method

The Hoare method of program verification consists of a logic and a calculus in which we can state propositions about the partial correctness of while programs, i.e., programs using the control structures described above in a safe language subset. A Hoare triple is an expression $\{P\}\ S\ \{Q\}$ where P and Q are open formulas of first-order logic and S is a while program. It is assumed that the reader has some understanding of the Hoare method of program verification, e.g., P represents a pre-condition and Q represents a post-condition. The Hoare calculus includes the following set of inference rules for deriving triples.

Assignment: $\{P^{t}/_{x}\}\ x := t\ \{P\}$.

If the property P is true after execution of the assignment, then before the assignment the property P, with x replaced by t, is true.

Composition: If the triples $\{P\}\ S_1\ \{R\}$ and $\{R\}\ S_2\ \{Q\}$ are derivable then the triple $\{P\}\ S_1; S_2\ \{Q\}$ is derivable.

If the first statement has a post-condition that matches the pre-condition of the second statement, then the composition of the statements by the ";" statement separator has the precondition of the first statement and the post-condition of the second.

Conditional: If the triples $\{P \wedge E\}\ S_1\ \{Q\}$ and $\{P \wedge \neg E\}\ S_2\ \{Q\}$ are derivable, then the triple $\{P\}$ if E then S_1 else S_2 fi $\{Q\}$ is derivable.

While: If the triple $\{P \wedge E\}\ S\ \{P\}$ is derivable, then the triple $\{P\}$ **while** E **do** S **od** $\{P \wedge \neg E\}$ is derivable.

If the loop invariant P is maintained by the loop body S, then the loop with precondition P has a post-condition that is the conjunction of the invariant and the negation of the loop condition E. Loop termination is proved separately.

Consequence: If the first-order formula $P \rightarrow Q$ is derivable, the triple $\{Q\}\ S\ \{R\}$ is derivable and the formula $R \rightarrow T$ is derivable, then the triple $\{P\}\ S\ \{T\}$ is derivable.

5.2 A Simple Situational Example

The example program fragment in figure 5.1 is intended to reduce the value of the variable `d` to the (lesser or equal) value stored in the variable `b`. The verification shows that the program accomplishes this and the commentary on the right side of each assertion gives a short justification for each proof step in terms of the basic Hoare rules. This is a very simple program which is included simply to demonstrate the technique.

Since the loop invariant is asserted before the loop and is maintained by the loop body we can conclude (according to the while rule) the loop post-condition and the subsequent final simplification stating the desired result.

The details for the derivation within the loop are given in figure 5.2. A variation on the usual assignment rule is that the post-condition is transformed to the pre-condition by replacing every occurrence of the variable changed by the state changing operation `dec(d)`. This is equivalent to the usual rule in which the variable modified is replaced in the post-condition by the right hand side of

{b ≤ d}	loop invariant
WHILE b≠d DO	
{b≤d ∧ b≠d}	see details below
dec(d)	equivalent to d := d-1
{b≤d}	desired loop invariant
END	
{b ≤ d ∧ b=d}	by while rule
{b=d}	simplification

Figure 5.1: While Program Hoare Verification – Integer Case

{b≤d ∧ b≠d}	
{b<d}	simplification of above
{b≤dec(d)}	property of decrement function
dec(d)	
{b≤d}	desired loop invariant

Figure 5.2: Loop Body Hoare Verification – Integer Case

an assignment. The steps from the first to the third assertion are by common simplifications of arithmetic, where the value of `dec(d)` is the same as `d-1`.

The derivation above uses an informal notion of variable and an implicit theory of natural numbers stored in the variables. These can be made more explicit by writing the situational classified model assertions of `nat` for the values stored and the classified model specification of figure 5.3 for the two identifiers `b` and `d` which represent separate variables of sort `card` (for cardinal). The `card` assertions are for a multiple-instance module which supports the storage of natural numbers. Assertion C5 states that zero is stored for each identifier in the initial state `s`. Assertion C6 states that after incrementing `C` the value of `C` is the successor of its value in the state `W`. Implicit frame axioms for each identifier ensure that separate variables do not interact. Axioms for the ordering relations $<$ and $\leq$ can be

C1: card(b)
C2: card(d)
C3: st(s)
C4: (W,C) st(W), card(C) → st(W; inc(C))
C5: (C) card(C) → s:C = 0
C6: (W,C) st(W), card(C) → W; inc(C):C = s((W:C))
C7: (W,C) st(W), card(C) → W; inc(C); dec(C) = W

Figure 5.3: Axioms For card Variables

{W:b ≤ W:d}	loop invariant
WHILE b≠d DO	
{W:b≤W:d ∧ W:b≠W:d}	see details below
dec(d)	equivalent to d := d-1
{W:b≤W:d}	desired loop invariant
END	
{W:b≤W:d ∧ W:b=W:d}	by while rule
{W:b=W:d}	simplification

Figure 5.4: While Program Hoare Verification – Situational Integer Case

obtained from the corresponding axioms for nat.

The extended verification of figure 5.4 is similar to the earlier example of figure 5.1 except that each occurrence of a variable name in an assertion is replaced with an expression denoting the value of that variable identifier in a specific state W. This makes explicit the state changing nature of some operations, e.g., assignment.

The main difference between the basic and extended Hoare techniques is in the treatment of state changing commands (e.g., assignment). *The effect of a state changing command such as* dec(d) *is recorded by forming a pre-condition from the post-condition by replacing every occurrence of the state variable* W *with the new state* W;dec(d). This is given by an assignment rule generalization that can be applied to any operation S that changes a state W:

$\{\texttt{W:b} \leq \texttt{W:d} \wedge \texttt{W:b} \neq \texttt{W:d}\}$	
$\{\texttt{W:b} < \texttt{W:d}\}$	simplification of above
$\{\texttt{W:b} \leq \texttt{W;dec(d):d}\}$	derived property of **dec** fcn
`dec(d)`	
$\{\texttt{W:b} \leq \texttt{W:d}\}$	desired loop invariant

Figure 5.5: Loop Body Hoare Verification – Situational Integer Case

S1 `stk(b)`
S2 `stk(d)`
S3 `st(s)`
S4 $(\texttt{W},\texttt{S})$ $\texttt{st(W)},\texttt{stk(S)} \rightarrow \texttt{st(W;push(S))}$
S5 $(\texttt{W},\texttt{S},\texttt{X})$ $\texttt{st(W)},\texttt{stk(S)},\texttt{nat(X)} \rightarrow \texttt{W;push(C,X):top(C)} = \texttt{X}$
S6 $(\texttt{W},\texttt{S},\texttt{X})$ $\texttt{st(W)},\texttt{stk(S)} \rightarrow \texttt{W;push(C,X);pop(C)} = \texttt{W}$

Figure 5.6: Axioms For **stk** Variables

State Change $\{\mathsf{P}^{\mathsf{W};\mathsf{S}}/\mathsf{W}\}\ \mathsf{S}\{\mathsf{P}\}$.

If the property P is true after execution of the state changing operation S then before the operation the property P, with W replaced by W; S, is true.

5.3 A Situational Stack Hoare Verification

The stack example of figure 5.6 is similar to the above **card** example, except the underlying classified model specification is a multiple-instance **stack-of-nat** module. The **card** module could have been named **store-of-nat** to emphasize the similarity. The frame axioms are implicit. There is an analogy between **inc** and **push**, between **dec** and **pop**, and between the **card** variable name and the **top** applied to each **stk** variable name.

The verification is for a program that reduces a given stack **d** to equal a substack **b**, where each value in the stack **d** is distinct. It is analogous to the

{W:b ≤ W:d}	loop invariant
WHILE top(b)≠top(d) DO	
{W:b≤W:d ∧ W:top(b)≠W:top(d)}	see details below
pop(d)	
{W:b≤W:d}	desired loop invariant
END	
{W:b≤W:d ∧ W:top(b)=W:top(d)}	by while rule
{W:b=W:d}	simplification

Figure 5.7: While Program Hoare Verification – Stack Case

{W:b≤W:d ∧ W:top(b) ≠ W:top(d)}	
{W:b<W:d}	by a stack theorem
{W:b≤W;pop(d):d}	by a stack theorem
pop(d)	
{W:b≤W:d}	desired loop invariant

Figure 5.8: Loop Body Hoare Verification – Stack Case

previous programs which reduced a given variable d to equal another (less than or equal) variable b. The verification is based upon several stack theorems, which can be derived from the above stack module specification, and assertions for substack ordering relations, which can be derived from the corresponding relations for nat. The stack theorems are dealt with explicitly in the similar example of the next chapter. The while loop derivation of figure 5.7 is similar to the previous cases. The details in figure 5.8 of the verification that the body of the loop maintains the loop invariant are similar to the corresponding versions for the card program.

An advantage of the situational approach is that various states of the computation can be given explicitly instead of by some arbitrary convention. For example, it is usual in Hoare logic to refer to the value of a variable x before some state changing operation by a "ghost" variable in the form of some textually

distinguished name such as `x'`. This is unnecessary when states are explicit since the value of `x` before a state changing operation `S` is `W:x` and after the operation the value is `W;S:x`.

5.4 Summary

It has been shown that Hoare's method can be extended to include modules as "big variables". The motivation rests in the software engineering technique, described in Chapter 7, that shows how to design programs that are limited to procedures and functions defined by classified model specifications.

Chapter 6

Program Synthesis

The synthesis of a program can occur as a side-effect of the proof of a specification theorem using a technique of Manna and Waldinger[MW87b]. They published full first-order axioms describing a "blocks world" and synthesized an imperative program to clear a given block of any covering blocks. However, to ensure that there is no movement of blocks beneath the target block to clear, their synthesis required one manual step to strengthen the theorem to prove. In this chapter, however, no strengthening is required since no movement of covered blocks can occur when the blocks are represented as a stack of blocks and the block to clear of covering blocks is modeled as the top of a substack. No strengthening is required because a stack module provides no operations to access any block below the current top block.

6.1 A Blocks-world Synthesis

A multiple-instance stack-of-block module, like the stack-of-nat module of the previous chapter, is used below with the sub-stack relation "less than or equal to" ($\leq$) and the well-founded strict sub-stack relation[1] "less than" ($<$).

The program derivation uses the Manna and Waldinger[MW89] tableau technique: a full first-order, non-clausal resolution method that generalizes refutation

[1] A well-founded relation has no infinite decreasing sequences.

techniques. It has also been automated and includes induction. For readers unfamiliar with the details of the technique, each step is paraphrased below. A Horn formula restriction in the assertions of a CM specification ensures the existence of an initial model, but because Horn with equality is a liberal institution, as noted in the introductory chapter, there is no such restriction in *derivation* with CM specifications in a full first-order language logical formalism such as that of Manna and Waldinger. Assertions in either the Assert or Goal columns in any tableau can be moved to the other column by negating the assertion. A refutation technique results if all goal assertions are moved to the Assert column in this way.

In the derivation, which starts in figure 6.1, all classification assertions of these relativized formulas are omitted for brevity, but the same program would be derived if they were included. The sub-stack `s:b` of blocks to be exposed is related to the initial stack `s:d` by the the sub-stack relation $\leq$ mentioned above. Each element of the stack `s:d` is distinct.

The goal assertion G1 in figure 6.1 states that we must modify the stack `d` by deriving some fluent Z_1 which causes some object `a` to be on top; we assume `a` is `top(b)`. The antecedent of the goal states that `b` is a non-empty sub-stack of `d`. The consequent of the goal states that in the initial state `s` the top of stack `b` equals the final state $s;Z_1$ value of the top of stack `d`. The derivation task is to find a fluent expression (a program) `clr` in the "Prog" column. During the derivation the program must be "suitable" for the goal, i.e., the term $s;Z_1$ must satisfy the goal assertion. This property is maintained by the rules as the fluent term in the "Prog" column is altered according to the rule of inference invoked at each step. Line A2 is the result of applying a well-founded induction rule to goal G1. The assertion says that the desired result is true for all stacks less than `d`, where capitalized symbols represent variables.

Label	Assert	Goal	Prog	Rule
G1		$(\texttt{s:nil} < \texttt{s:b} \wedge \texttt{s:b} \leq \texttt{s:d}) \rightarrow$ $\texttt{s:top(b)} = \texttt{s};\texttt{Z}_1\texttt{:top(d)}$	$\texttt{s};\texttt{Z}_1$	
A2	$\texttt{W:U} < \texttt{s:d} \rightarrow$ $(\texttt{s:nil} < \texttt{s:b} \wedge \texttt{s:b} \leq \texttt{W:U}) \rightarrow$ $\texttt{W:top(b)} = \texttt{W};\texttt{clr:top(U)}$			induct G1

Figure 6.1: Synthesis Step 1: Well-founded induction statement

Label	Assert	Goal	Prog	Rule
A3 A4 G5	$\texttt{s:nil} < \texttt{s:b}$ $\texttt{s:b} \leq \texttt{s:d}$	$\texttt{s:top(b)} = \texttt{s};\texttt{Z}_1\texttt{:top(d)}$	$\texttt{s};\texttt{Z}_1$	if-split G1
G6		$\texttt{s};\texttt{Z}_2\texttt{:d} < \texttt{s:d} \wedge$ $\texttt{s:nil} < \texttt{s:b} \wedge$ $\texttt{s:b} \leq \texttt{s};\texttt{Z}_2\texttt{:d} \wedge$ $\texttt{s:top(b)} = \texttt{s};\texttt{Z}_2\texttt{:top(b)}$	$\texttt{s};\texttt{Z}_2;\texttt{clr}$	res G5,A2 $\texttt{W} \leftarrow \texttt{s};\texttt{Z}_2$ $\texttt{U} \leftarrow \texttt{d}$ $\texttt{Z}_1 \leftarrow \texttt{Z}_2;\texttt{clr}$

Figure 6.2: Synthesis Step 2: Well-founded induction application

The verification (synthesis) continues in figure 6.2 with a split of goal G1 into assumptions A3,A4 and goal G5. Goal G5 and the induction assumption A2 resolve except for the terms `W:top(b)` and `s:top(b)`, but this can be handled by including the equality of these two terms (after the substitution) as a conjunct in the new goal G6. It is the use of induction that results in the introduction of (tail) recursion, as demonstrated by the introduction of `;clr` at the end of the "Prog" column term that represents the program `clr`.

Assumptions A7 and A9 are theorems that can be proved by induction from the stack specification. These represent some of our common working assumptions about stacks. A7 of figure 6.3 states that pop-ing a non-nil stack `W:U` results in a sub-stack of the original stack. Step G8 in figure 6.3 is produced by resolution

Label	Assert	Goal	Prog	Rule
A7	$\mathtt{s{:}nil} < \mathtt{W{:}U} \rightarrow$ $\mathtt{W;pop(U){:}U} < \mathtt{W{:}U}$			theorem
G8		$\mathtt{s{:}nil} < \mathtt{s{:}d} \wedge$ $\mathtt{s{:}nil} < \mathtt{s{:}b} \wedge$ $\mathtt{s{:}b} \leq \mathtt{s;pop(d){:}d} \wedge$ $\mathtt{s{:}top(b)} =$ $\mathtt{s;pop(d){:}top(b)}$	$\mathtt{s;pop(d);clr}$	res G6,A7 $\mathtt{Z_2} \leftarrow \mathtt{pop(d)}$ $\mathtt{U} \leftarrow \mathtt{d}$ $\mathtt{W} \leftarrow \mathtt{s}$

Figure 6.3: Synthesis Step 3: pop less than theorem

Label	Assert	Goal	Prog	Rule
A9	$\mathtt{s{:}b} < \mathtt{W{:}U} \rightarrow$ $\mathtt{s{:}b} \leq \mathtt{W;pop(U){:}U}$			theorem
G10		$\mathtt{s{:}nil} < \mathtt{s{:}d} \wedge$ $\mathtt{s{:}nil} < \mathtt{s{:}b} \wedge$ $\mathtt{s{:}b} < \mathtt{s{:}d}$	$\mathtt{s;pop(d);clr}$	res G8,A9 $\mathtt{W} \leftarrow \mathtt{s}$ $\mathtt{U} \leftarrow \mathtt{d}$

Figure 6.4: Synthesis Step 4: pop less than or equal theorem

from G6 and A7 and results in a fluent term `pop(d);clr` in the program to derive. This is a fundamental step in the derivation and it demonstrates that the common intuitions in programming are based upon derived results such as A7 rather than the axioms of the specifications. The final conjunct of G8 is an instance of a frame axiom asserting that state-changing operations of different stacks do not interfere. For brevity, frame axioms are applied automatically during resolution.

A9 in figure 6.4 states that if a given stack `s:b` is strictly less than another stack `W:U`, then `s:b` is less than or equal to the result of pop-ing `W:U`. Application of this theorem simplifies the goal, as shown in G10.

Figure 6.5 applies a transitivity axiom A9 to reduce the goal.

In figure 6.6 a stack theorem states that a sub-stack having a different top

Label	Assert	Goal	Prog	Rule
A11	$(\mathtt{W{:}S_1} < \mathtt{W{:}S_2} \wedge$ $\mathtt{W{:}S_2} < \mathtt{W{:}S_3}) \rightarrow$ $\mathtt{W{:}S_1} < \mathtt{W{:}S_3}$			stack axiom
G12		$\mathtt{s{:}nil} < \mathtt{s{:}b} \wedge$ $\mathtt{s{:}b} < \mathtt{s{:}d}$	`s;pop(d);clr`	res G10,A11 $\mathtt{W} \leftarrow \mathtt{s}$ $\mathtt{S_1} \leftarrow \mathtt{nil}$ $\mathtt{S_2} \leftarrow \mathtt{b}$ $\mathtt{S_3} \leftarrow \mathtt{d}$
G13		$\mathtt{s{:}b} < \mathtt{s{:}d}$	`s;pop(d);clr`	res G12,A3

Figure 6.5: Synthesis Step 5: transitivity axiom

Label	Assert	Goal	Prog	Rule
A14	$\mathtt{s{:}b} \leq \mathtt{W{:}U} \wedge$ $\neg(\mathtt{W{:}top(b)} =$ $\mathtt{W{:}top(U)}) \rightarrow$ $\mathtt{s{:}b} < \mathtt{W{:}U}$			theorem
G15		$\mathtt{s{:}b} \leq \mathtt{s{:}d} \wedge$ $\neg(\mathtt{s{:}top(b)} = \mathtt{s{:}top(d)})$	`s;pop(d);clr`	res G13,A14 $\mathtt{W} \leftarrow \mathtt{s}$ $\mathtt{U} \leftarrow \mathtt{d}$

Figure 6.6: Synthesis Step 6: non-equal top

than the super-stack must be strictly less than the super-stack. This is used to introduce into the derivation a term that will appear in the program in the final step.

The final step of the derivation produces the recursive program `clr`, which, due to its "linear" form, can be easily converted to an equivalent iterative form[MW87b]. Λ is a fluent having no effect.

Label	Assert	Goal	Prog	Rule
G16		$\neg$(s:top(b) = s:top(d))	s;pop(d);clr	res G15,A4
G17		*true*	s;if top(b) = top(d) then Λ else pop(d);clr	res G5, G16 $Z_1 \leftarrow \Lambda$

Figure 6.7: Synthesis Step 7: recursion introduction

6.2 Module Implementation Models

Mathematical models of a set of assertions of a specification can be constructed by (1) choosing a representation for the objects of the universe and (2) choosing functions which satisfy the specification, i.e., in a CM specification we can substitute for each classified term its representative in the model and for each function symbol the chosen actual function.

Actual functions, i.e., fluents representing module implementations, can be derived by the Manna-Waldinger program synthesis method, by (1) choosing a situational term in some implementation representation for each classified term and (2) deriving an actual function that satisfies all specification assertions that include the function.

For example, we can implement a "table list" module in this way. The single-instance table-list module specified in figure 6.2 extends a stack (read the table-list operations `insert`, `delete`, `current` as `push`, `pop`, `top`) in two stages:

1. A traversing stack extends a stack by allowing read access to any stored elements of the stack (via a "current" position), but no capability to change the traversing stack except by the usual stack operations.

2. A table-list module additionally allows new elements to be inserted and

T1		`st(s)`
T2	`(W,A) st(W), chr(A)` $\rightarrow$	`st(W;insert(A))`
T3	`(W,A) st(W), chr(A)` $\rightarrow$	`W;insert(A);delete = W`
T4	`(W,A) st(W), chr(A)` $\rightarrow$	`W;insert(A):current = W:A`
T5	`(W,A) st(W), chr(A)` $\rightarrow$	`st(W;insert(A);goleft)`
T6	`(W,A) st(W), chr(A)` $\rightarrow$	`W;insert(A);insert(B);goleft = W;insert(B);goleft;insert(A)`
T7	`(W,A) st(W), chr(A)` $\rightarrow$	`W;insert(A);goleft;goright = W;insert(A)`

Figure 6.8: Table-List CM Specification

deleted at the "current" position in a traversing stack.

We can implement the table-list by (1) choosing as representation two stack modules, `l` (left) and `r` (right), such that

1. `W;insert(X)` is represented by `W;push(l,X)`

2. `W;insert(X);goleft` is represented by `W;push(r,X)`

and (2) deriving the implementation of **delete**, also shown below. Note that each actual function of the "implementation model" can be derived in this way independently of the derivation of each other actual function.

Goal G1 of the implementation derivation in figure 6.2 consists of the **delete** axiom T3 in which we have substituted the stack representation for the classified term and a variable `T` for the situational term **delete**, i.e., program, to derive.

Goal G3 is the result of resolution between G1 and A2, an axiom in the multiple-instance stack specification, with the substitutions shown. The result program `pop(l)` is an implementation for delete in the proposed model.

Label	Assert	Goal	Prog	Rule
G1		$\mathtt{w};\mathtt{push}(\mathtt{l},\mathtt{a});\mathtt{T} = \mathtt{w}$	T	
A2	$\mathtt{W};\mathtt{push}(\mathtt{S},\mathtt{X});\mathtt{pop}(\mathtt{S}) = \mathtt{W}$			Axiom
G3		*true*	pop(l)	res G1,A2 S ← l X ← a T ← pop(l) W ← w

Figure 6.9: Table-List **delete** derivation

6.3 Related Work

Manna and Waldinger have developed program synthesis techniques for applicative programs[MW87c] and imperative programs[MW87b][MW87a] based on their deductive techniques[MW89][MW86]. They have also investigated imperative languages from a situational viewpoint[MW81], but did not apply their techniques to the verification of programs calling modules.

6.4 Summary

This chapter is a simple application of Manna and Waldinger's imperative program synthesis, with one exception: they[MW87b] use assertions expressed in a full first-order language. In contrast, the classified model specification assertions used in this chapter are restricted to a Horn-with-equality subset of full first-order logic, although the derivations are not restricted to this language subset. The stack representation for the blocks problem solved a difficulty experienced by Manna and Waldinger: with their full first-order axioms they could not complete the synthesis without "strengthening"[MW87b] the theorem to prove in some manual way. No such strengthening is required in the synthesis of this chapter. This

improvement appears to be due to the choice of axioms that ensure an initial term model.

This chapter is also a demonstration of an application of the situational CM language.

Chapter 7

A Software Engineering Technique

The software engineering technique presented in this chapter uses the specification concepts of the previous chapters to write in a common notation both *behavior requirement* and *implementation design* specifications. The first type of specification is the result of a problem analysis process that documents *what* is required of software without regard to *how* it is implemented. The second is a plan for an implementation that has the behavior prescribed by the first, but also addresses issues of implementation and ongoing maintenance.

For example, the kwic CM specification given earlier is a requirements specification because it defines the required behavior but it does not directly address implementation issues. Note that although kwic is expressed in terms of parameterized `SQUEUE` and `QUEUE`, the *behavior* of the module specification as described by the export interface is independent of these submodules. Sometimes the distinction between requirement and design specification is blurred with executable specifications because the specification is an implementation (OBJ[FGMO87]), but not the only possible implementation. Even if the behavior requirements specification is executable, it may not be the best form of implementation because there may be another implementation that executes faster (e.g., an imperative program) or is more modifiable.

A kwic implementation specification for modules that are more modifiable is given by Parnas[Par72b], although he also allows random access to the output

lines and considers characters, not words to be atomic. The main difference between the kwic requirements specification of this dissertation and Parnas' design specification is that he decomposes the problem with future maintainability in mind. Instead of describing the problem in terms of the convenient (off-the-shelf) `SORTING-QUEUE-OF-LINE` and `QUEUE-OF-WORD` parameterized specifications Parnas selects modules:

Line Holder stores all input lines. Each line is entered sequentially but can be accessed randomly by indexing to a line and a word within a line. Future maintainability is enhanced by hiding within this module all issues related to storage of data.

Circular Shifter is responsible for producing all circular shifts of all stored data. Future maintainability is enhanced by hiding within this module all issues related to the algorithm for producing circular shifts, e.g., whether circular shifts are stored redundantly or computed from the line holder data on demand.

Alphabetizer is responsible for sorting all data. Future maintainability is enhanced by hiding within this module both the sorting algorithm and the sorting time, e.g., batch sorting or incremental as required.

The kwic requirements specification of a previous chapter (summarized in figure 7.1) has a module dependency structure in which the kwic specification is expressed in terms of the parameterized squeue(queue(word)) and queue(word) specifications. In this decomposition the storage of data is the responsibility of both queue and sorting queue, the rotation method is tied to the queue and the sorting algorithm to the sorting queue. This requirements specification does not

address the separation of concerns, and future maintainability, that is in the Parnas design specification. The kwic requirements have already been given as a CM specification and the design specification could also be given as the additional constraint that the exported operations should be implemented by the modules Line Holder, Circular Shifter and Alphabetizer. This implementation constraint is similar to the implementation of the Table-List specification by two stacks, as described in a previous chapter. The behavior of the specified module is fixed by the requirements specification, but the means of implementing it can range from the original requirement specification (if it can be executed) to an imperative (situational) implementation based upon modules such as Line Holder, Circular Shifter and Alphabetizer. Just as a Table-List module implementation was derived in a previous chapter from the Table-List axioms and the axioms for an implementation consisting of two stacks, so too can an implementation of the kwic module be derived from the requirements specification, which describes the desired behavior, and the axioms for the implementation modules Line Holder, Circular Shifter and Alphabetizer.

Finally, by the results of the previous chapter such an implementation can be expressed in the CM (situational) language, so that behavior requirements specification, implementation design specification and an actual implementation (or model) can all be described in the same CM language if the implementation is restricted to operations that are described by CM specifications.

7.1 The Technique

A software engineering technique that encourages the description of *all* operations by module specifications could contain the (perhaps overlapping) activities

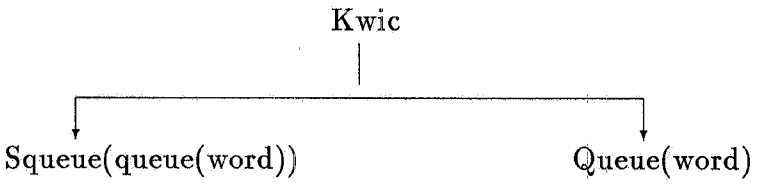


Figure 7.1: Kwic Requirements Module Dependency Structure

of analysis, design and implementation.

Analysis is the process of determining the requirements of software and documenting these as[Par84]:

- Specification: a formal system requirements description, as described in this dissertation. This description prescribes behavior without suggesting an implementation.
- Assumptions: issues that are likely to remain true throughout the life of the system.
- Likely changes (or, secrets): issues that must be easy to change throughout the life of the system.
- Subsets: portions of the system that can be incrementally implemented.
- Other issues: Hardware, timing and accuracy constraints.

Design is the selection of implementation module(s), i.e., modules that may be different from those used in the requirements specification to describe the software under construction. The requirements description need not be decomposed in a way that supports the likely changes, but the design decomposition must.

Implementation is the derivation of programs satisfying the requirements specification and calling implementation modules, which possibly call programs within other implementation modules.

We can choose to describe system requirements in terms of the modules that we use for the implementation, but this might not be convenient. Conversely, we can choose to implement a system in terms of modules that are convenient for the requirements description, but this might not be efficiently executed or easily maintained. For example, the kwic requirements could have been described in terms of the three modules of Parnas, but these are more complex than the sorting queue and queue specifications. Conversely, the sorting queue and queue parts of the kwic requirements specification are convenient, but they do not support the likely changes to data structure or algorithms.

Corresponding to each step above there are the following software structures[1].

Analysis has the module requirements specification structure which is defined by a set of specification construction operations upon a set of smaller (possibly standard) specifications, for example by the specification library interconnect language LIL[Gog86]. The structure is a partial order between module specifications that are related by their definitional dependency upon other modules. These definitional dependencies can also be described[Par84] as the assumptions the writer of one module specification can depend upon in the other module(s). Except for the need for certain sub-modules common to several super-modules, the structure could be hierarchical. An example of such a structure is the dependency of the kwic specification upon the parameterized specifications for `SQUEUE` and `QUEUE`.

[1] *Structure* is defined as a set of objects and a relation among the objects

Design has the module work assignment structure[Par84] which defines the work assignments given to each member of a team of programmers (or automatic program synthesizers), where the assignments are related by a "is contained in" relation (hierarchy).

Since a requirements decomposition (e.g., queue and squeue for kwic) can also be considered a design, albeit not necessarily one that supports the identified changes, it is possible to construct a work assignment structure on the requirements decomposition. Since the requirements decomposition is not constructed with the changes in mind, we can expect that any work assignment might share change issues with other work assignments and that maintainability might not be an attribute of an implementation based upon such a design. For example, the `SQUEUE` and `QUEUE` modules of kwic requirements decomposition share the storage of data.

Conversely, a design decomposition (e.g., line holder, circular shifter and alphabetizer) can also be considered a requirements statement, although it might not be as simple to understand as a decomposition that does not take into account possible changes. Since a design decomposition can consist of specifications that are defined in terms of other specifications, it can certainly have the definitional dependency structure.

7.1.1 The Module Work Assignment Structure

A motivating criteria for work sub-division as a design structure is that the unit of work assignment can also be the unit of change for system modification. This is because work that can be described for independent creation as a cohesive unit is only weakly coupled, via an interface specification, with other such work

assignments, hence changes that are not associated with the couplings are independently applied. This suggests that prior to the initial work sub-division we should partition the design decisions into those that can be committed, known as assumptions or unlikely changes, and design decisions that can be deferred, known as secrets or likely changes. Since we wish to apply this criterion to arbitrarily large systems and we also wish to achieve work assignments small enough that they can be discarded at small cost, we must repeat the sub-division process to achieve an ordering of work assignments based upon the subset relation. When a work assignment is expressed in this light, its previously deferred decisions are in turn partitioned into a set that must be decided and a set that is deferred.

How To Discover Modules

The process of discovering modules can be based upon the criteria described above for the module *work assignment* structure. The process for describing modules in terms of other modules can be based upon the module *dependency* structure. Any module decomposition of a system has both of these module structures. For example, the kwic decomposition into Line Holder, Circular Shifter and Alphabetizer modules has a work assignment structure consisting of those three modules and a module dependency structure which is determined by the specification descriptions, as described below.

7.2 Example: Kwic Design

The Kwic design specification is described in this section with the three design modules described above. This example is small enough to illustrate the implementation technique, but too small to really show the power of the design

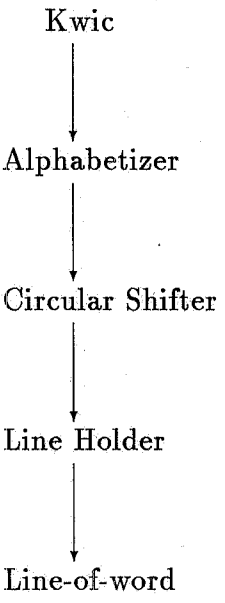


Figure 7.2: Kwic Design Module Dependency Structure

structure.

The dependency structure for this decomposition is shown in figure 7.2 and the work assignment structure is shown in figure 7.3, where each module has its likely change in parenthesis below.

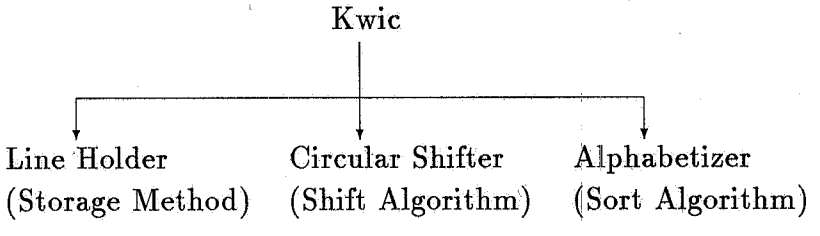


Figure 7.3: Kwic Design Module Work Assignment Structure

7.2.1 Line Holder Module

The line holder module is specified below in two parts. The first part is a supporting "line-of-word" ADT in figure 7.4. The second part is the line-holder in figure 7.5. Since they are similar, both specifications could have been defined as specific instances of a parameterized `INDEXED-LIST(DAT)` module, i.e., `INDEXED-LIST(WORD)` for line-of-word and `INDEXED-LIST(LINE)` for the line holder. An `INDEXED-LIST(DAT)` module provides a list where each entry is indexed by successively larger natural numbers. This restricts the list to strictly sequential additions, yet allows random access and removal of list items. A parameterized module was avoided to not obscure, for this decomposition, the easy comparison of the module dependency structure, as defined by the specification text, and the module work assignment structure.

In each specification the basic classified objects are defined inductively by the first two assertions (L1,L2 of figure 7.4 and H1,H2 of figure 7.5, where the antecedent in the second assertion of each specification enforces the sequential insertion restriction. The next two assertions of each specification (L3,L4 and H3,H4) define a count of the number of objects ever added (not the current number stored since the LineHolder delete axiom does not adjust these counts). The fifth assertion in each specification defines a selector operation to "get" the object added, by index.

The remaining operations (L6-L13) of the line-of-word specification define head and tail operations which support a `rot1` operation to rotate a line by one word. A partial order on line-of-word is also defined. The final operation (H6) of the line holder module is the delete-line operation `del1`. There is a definitional dependency (see figure 7.2) between the line holder and the line-of-word.

L1	$\mathtt{ln(newln)}$	(line base)
L2	$(\mathtt{L_{ln}}, \mathtt{W_{wd}}, \mathtt{J_{nat}})\ \mathtt{J_{nat}{=}L_{ln}{:}wc} \rightarrow \mathtt{ln(L_{ln}; addw(W_{wd}, J_{nat}))}$	(line gen)
L3	$\mathtt{newln{:}wc} = 0$	(word count base)
L4	$(\mathtt{L_{ln}}, \mathtt{W_{wd}}, \mathtt{J_{nat}})\ \mathtt{L_{ln}; addw(W_{wd}, J_{nat}){:}wc} = \mathtt{L_{ln}{:}wc} + 1$	(wrd cnt gen)
L5	$(\mathtt{L_{ln}}, \mathtt{W_{wd}}, \mathtt{J_{nat}}, \mathtt{J'_{nat}})\ \mathtt{L_{ln}; addw(W_{wd}, J_{nat}){:}getw(J'_{nat})} =$ $\mathtt{if\ J_{nat} = J'_{nat}\ then\ W_{wd}\ else\ L_{ln}{:}getw(J'_{nat})}$	(get word)
L6	$(\mathtt{W_{wd}})\ \mathtt{head(newln; addw(W_{wd}))} = \mathtt{addw(W_{wd})}$	(head base)
L7	$(\mathtt{L_{ln}}, \mathtt{W_{wd}}, \mathtt{W'_{wd}})\ \mathtt{head(L_{ln}; addw(W_{wd}); addw(W'_{wd}))} =$ $\mathtt{head(L_{ln}; addw(W_{wd}))}$	(head gen)
L8	$(\mathtt{W_{wd}})\ \mathtt{tail(newln; addw(W_{wd}))} = \mathtt{newln}$	(tail base)
L9	$(\mathtt{L_{ln}}, \mathtt{W_{wd}})\ \mathtt{tail(L_{ln}; addw(W_{wd}); addw(W'_{wd}))} =$ $\mathtt{tail(L_{ln}; addw(W_{wd})); addw(W'_{wd})}$	(tail generator)
L10	$(\mathtt{W_{wd}})\ \mathtt{rot1(newln; addw(W_{wd}))} = \mathtt{newln; addw(W_{wd})}$	(rotate 1 base)
L11	$(\mathtt{L_{wd}}, \mathtt{W_{wd}}, \mathtt{W'_{wd}})\ \mathtt{rot1(L_{ln}; addw(W_{wd}); addw(W'_{wd}))} =$ $\mathtt{tail(L_{ln}; addw(W_{wd}); addw(W'_{wd})); head(L_{ln}; addw(W_{wd}))}$	(rot 1 gen)
L12	$(\mathtt{L_{ln}}, \mathtt{L'_{ln}}, \mathtt{W_{wd}})\ \mathtt{L_{ln}{=}L'_{ln}} \rightarrow \mathtt{L_{ln} < L'_{ln}; W_{wd}})$	(partial order base)
L13	$(\mathtt{L_{ln}}, \mathtt{L'_{ln}}, \mathtt{W_{wd}}, \mathtt{W'_{wd}})\ \mathtt{L_{ln}{=}L'_{ln}, W_{wd}{<}W'_{wd}} \rightarrow$ $\mathtt{L_{ln}; W_{wd} < L'_{ln}; W'_{wd}}$	(partial order gen)

Figure 7.4: Line-of-Word CM Specification

7.2.2 Circular Shifter Module

The circular shifter (figure 7.6) module provides all circular shifts of each line stored in the line holder module. Since the method of generating all shifts is a secret of this module it is possible that the module stores all shifts or it is possible that the module generates all circular shifts directly from the data stored in the line holder. It is only the *behavior* of the module (i.e., the values returned by the operations) that is prescribed by the specification, so no particular implementation is prescribed by structuring the specification as described below.

H1 $\mathtt{lh(newlh)}$ (lh base)

H2 $(\mathtt{S_{lh}}, \mathtt{L_{ln}}, \mathtt{I_{nat}})\ \mathtt{I_{nat}}{=}\mathtt{S_{lh}}{:}\mathtt{lc} \rightarrow \mathtt{lh}(\mathtt{S_{lh}}; \mathtt{L_{ln}}; \mathtt{addl}(\mathtt{I_{nat}}))$ (lh gen)

H3 $\mathtt{newlh{:}lc} = 0$ (lc base)

H4 $(\mathtt{S_{lh}}, \mathtt{L_{ln}}, \mathtt{I_{nat}})\ \mathtt{S_{lh}}; \mathtt{L_{ln}}; \mathtt{addl}(\mathtt{I_{nat}}){:}\mathtt{lc} = \mathtt{S_{lh}}{:}\mathtt{lc} + 1$ (lc gen)

H5 $(\mathtt{S_{lh}}, \mathtt{L_{ln}}, \mathtt{I_{nat}}, \mathtt{I'_{nat}}, \mathtt{J'_{nat}})\ \mathtt{S_{lh}}; \mathtt{L_{ln}}; \mathtt{addl}(\mathtt{I_{nat}}){:}\mathtt{getl}(\mathtt{I'_{nat}}, \mathtt{J'_{nat}}) =$
$\mathtt{if}\ \mathtt{I_{nat}} = \mathtt{I'_{nat}}\ \mathtt{then}\ \mathtt{L_{ln}}{:}\mathtt{getw}(\mathtt{J'_{nat}})\ \mathtt{else}\ \mathtt{L_{lh}}{:}\mathtt{getl}(\mathtt{I'_{nat}}, \mathtt{J'_{nat}})$ (get line)

H6 $(\mathtt{S_{lh}}, \mathtt{L_{ln}}, \mathtt{I_{nat}}, \mathtt{I'_{nat}})\ \mathtt{S_{lh}}; \mathtt{L_{ln}}; \mathtt{addl}(\mathtt{I_{nat}}); \mathtt{dell}(\mathtt{I'_{nat}}) =$
$\mathtt{if}\ \mathtt{I_{nat}} = \mathtt{I'_{nat}}\ \mathtt{then}\ \mathtt{S_{lh}}\ \mathtt{else}\ \mathtt{S_{lh}}; \mathtt{dell}(\mathtt{I'_{nat}}); \mathtt{L_{ln}}; \mathtt{addl}(\mathtt{I_{nat}})$ (delete line)

Figure 7.5: LineHolder CM Specification

S1 $\mathtt{newlh{:}sc} = 0$ (sc base)

S2 $(\mathtt{S_{lh}}, \mathtt{L_{ln}}, \mathtt{I_{nat}})\ \mathtt{S_{lh}}; \mathtt{L_{ln}}; \mathtt{adds}(\mathtt{I_{nat}}){:}\mathtt{sc} = \mathtt{S_{lh}}{:}\mathtt{sc} + 1$ (sc gen)

S3 $(\mathtt{S_{lh}}, \mathtt{L_{ln}}, \mathtt{I_{nat}})\ \mathtt{S_{lh}}; \mathtt{L_{ln}}; \mathtt{addl}(\mathtt{I_{nat}}) = \mathtt{S_{lh}}; \mathtt{kin}(\mathtt{S_{lh}}, \mathtt{L_{ln}}, \mathtt{L_{ln}}{:}\mathtt{wc})$ (addl equiv)

S4 $\mathtt{kin}((\mathtt{S_{lh}}, \mathtt{L_{ln}}, 0) = \Lambda$ (kin base)

S5 $(\mathtt{S_{lh}}, \mathtt{L_{ln}}, \mathtt{I_{nat}})\ \mathtt{kin}(\mathtt{S_{lh}}, \mathtt{L_{ln}}, \mathtt{I_{nat}} + 1) =$
$\mathtt{L_{ln}}; \mathtt{adds}(\mathtt{S_{lh}}{:}\mathtt{sc}); \mathtt{kin}(\mathtt{S_{lh}}; \mathtt{L_{ln}}, \mathtt{rot1}(\mathtt{L_{ln}}), \mathtt{I_{nat}})$ (kin gen)

S6 $(\mathtt{S_{lh}}, \mathtt{L_{ln}}, \mathtt{I_{nat}}, \mathtt{I'_{nat}})\ \mathtt{S_{lh}}; \mathtt{L_{ln}}; \mathtt{adds}(\mathtt{I_{nat}}); \mathtt{dels}(\mathtt{I'_{nat}}) =$
$\mathtt{if}\ \mathtt{I_{nat}} = \mathtt{I'_{nat}}\ \mathtt{then}\ \mathtt{S_{lh}}\ \mathtt{else}\ \mathtt{S_{lh}}; \mathtt{dels}(\mathtt{I'_{nat}}); \mathtt{L_{ln}}; \mathtt{adds}(\mathtt{I_{nat}})$ (delete sl)

Figure 7.6: Circular Shifter

The line holder is the basic storage mechanism for the kwic specification and assertions S3-S5 state that an **addl** (add line) operation is equivalent to a sequence of **adds** (add shifted line) operations, i.e., one **adds** operation for each shifted line generated by the **kin** (kwic input) operation. The Λ symbol is the identity element of the monoid with fluent objects and the operation ";". The (hidden) operation **kin** generates circular shifted lines from the kwic input line. There is a definitional dependency (see figure 7.2) between this module and the line holder module, but they are independent in the work assignment structure shown in figure 7.3.

A1 $(\mathtt{L_{ln}}, \mathtt{I_{nat}})$ $\mathtt{newlh{:}ml} = \mathtt{newln}$ (max line base)

A2 $(\mathtt{S_{lh}}, \mathtt{L_{ln}}, \mathtt{I_{nat}})$ $\mathtt{S_{lh}}; \mathtt{L_{ln}}; \mathtt{adds}(\mathtt{I_{nat}}){:}\mathtt{ml} =$ (max line generator)
$\mathtt{if}\ \mathtt{L_{ln}} > \mathtt{S_{lh}}{:}\mathtt{ml}\ \mathtt{then}\ \mathtt{L_{ln}}\ \mathtt{else}\ \mathtt{S_{lh}}{:}\mathtt{ml}$

A3 $(\mathtt{S_{lh}}, \mathtt{L_{ln}}, \mathtt{I_{nat}})$ $\mathtt{S_{lh}}; \mathtt{L_{ln}}; \mathtt{adds}(\mathtt{I_{nat}}){:}\mathtt{mi} =$ (max index generator)
$\mathtt{if}\ \mathtt{L_{ln}} > \mathtt{S_{lh}}{:}\mathtt{ml}\ \mathtt{then}\ \mathtt{I_{nat}}\ \mathtt{else}\ \mathtt{S_{lh}}{:}\mathtt{mi}$

Figure 7.7: Alphabetizer CM Module

7.2.3 Alphabetizer Module

The alphabetizer module (figure 7.7) provides the index, `mi`, of the alphabetically first circular shifted line. This line is returned by the max-line operation `ml`. The sorting method and the sort time (i.e., sort lines as they are inserted or only upon request of a max line) are secrets of the module. There is a definitional dependency between this module and the circular shifter module, but they are independent in the work assignment structure.

7.3 Related Work

Goguen[Gog90] describes an algebraic approach to refinement in which he states that a specification is a structured theory and a refinement is a structured theory morphism. A structured theory is a set of specifications that are arranged in hierarchies and are generic so they can be reused in as many contexts as possible. Refinements can also be structured because the development of a large program will generally involve a large number of refinement steps; in particular, we want to be able to parameterize refinements and reuse them, as well as compose them.

Parnas[Par72a] was the first to state the software engineering technique described in this chapter. He did not, however, distinguish between requirement

and design specifications. Conversely, Goguen[Gog81] and Ehrig[EK81] described (executable) specifications for the kwic problem, but did not address implementation techniques, such as the Parnas implementation decomposition, that are of practical value to a maintenance programmer.

Specifying an implementation design as well as the behavioral requirements also has application in ameliorating the "Mythical man-month" phenomenon [FPB75] in which adding programmers to a project does not necessarily result in more productivity. Since the design modules separate issues perhaps better than requirement modules, it is easier to add programmers by giving each a separate work assignment. This approach is also discussed in [Par71].

Another approach to the distinction between specification and design is that of [GHW85] where a common specification language has a different associated design language for each possible implementation language.

A requirements specification technique such as [Hen80] is related to the CM requirements specification technique since they both describe the value of operations applied to terms of the specification language. In [Hen80] some of the terms are of sort state, or of sort mode (sets of states), but these can handled within the CM technique.

7.4 Summary

Requirement and design specifications can both be described by a common CM specification language. A requirement specification is expressed in terms of convenient submodules. A design specification constrains the implementation to a selected set of maintainable submodules such that the whole system has the behavior described by the requirements specification.

Any decomposition can have both a dependency structure between the modules and a work assignment structure. The dependency structure records the way in which each specification is written in terms of other specification(s). This is different from the work assignment structure which records the inclusion relationship of module secrets. Both the requirement and design specifications are CM module specifications. By the results of the previous chapter, implementations can also be considered terms of the CM language.

Finally, building systems by connecting off-the-shelf components does not guarantee maintainability – it must be designed and this is the main difference between requirements and design specifications.

Chapter 8

Conclusions, Contributions And Future Research

The main conclusions and contributions that can be drawn from this work and some possibilities for future research are described below.

8.1 Conclusions And Contributions

The Classified Model (Horn with equality) specification technique has a defined syntax and semantics which has been formalized as an alternative to Many Sorted and Order Sorted techniques for the specification of abstract data types (ADTs) and modules (ADTs with hidden state sorts). The main contribution of this formalization is that a variant of the Order Sorted Model (OSM) rules of inference can be used within the classified technique and that the unclassified, or error terms, cause no difficulties in practice. That is, if an assertion containing just classified terms is derivable from a normal specification then there is a derivation which contains just classified terms.

The basic classified method may be called "specification-in-the-small" because it deals with the construction of individual assertions. "Specification-in-the-large" is addressed by existing parameterization techniques that can be used to glue component specifications into larger units. Results are cited to support the claim that Horn with equality is a sufficiently restrictive language to qualify as a "liberal institution" and therefore inherits many existing parameterization techniques. Further results are cited to support the claim that Horn with equality is the most general

first-order language that admits initial models. The specification of a small but often cited example (kwic) in terms of off-the-shelf specifications is offered as as an example of specification-in-the-large.

A situational logic is defined within the classified framework and it is shown how this can be used to define modules by isolating the hidden state terms. The kwic example is reworked in the situational format so that it is more readable than the original version because nested terms have been "linearized" by the situational operations ";" (production) and ":" (return). These operations are shown to have a relationship to corresponding functions of Parnas' method of specification by trace assertions[PB78].

The situational form of the classified method finds application in a generalization of Hoare logic that is "safe" for imperative programs which restrict their use of procedures and functions to the calls of a module specified by the classified method.

Another application of the classified situational logic is in the synthesis of programs according to the Manna and Waldinger technique[MW87b]. An example is given of a blocks-world synthesis which is simpler than a published synthesis by Manna and Waldinger of the same problem. The simplification is perhaps due to the use of axioms having an initial model. Also, a synthesis is given for the implementation of one operation for a "table-list" module in terms of two stack modules. One feature of this synthesis is that each operation of the table-list module can be derived independently of the others since they are all related by the specification.

A software engineering technique supports the use of the classified techniques for either requirements (what to do) or design (how to do it) specifications. The design specification could be considered an implementation representation of the

requirements and programs could be synthesized according to the method described for the table-list example. It is shown how requirement specifications differ from design specifications and how design specifications support program maintenance.

In summary, this work presents a specification technique and some of its practical applications.

8.2 Future Work

The main motivation of this work has been to develop the classified approach as a specification and derivation technique. It should be a straight-forward task to apply the Manna and Waldinger derivation checker to these hand-crafted derivations in the program synthesis problem. A more challenging task is to use a proof guidance system such as Manna and Waldinger's to derive these proofs.

In the synthesis of programs we produce situational logic terms (fluents) that can be considered imperative programs. Future work could show that the synthesized term can be considered a term of a logic with a reduced set of rules of inference and options (i.e., possible substitutions) for each rule. That is, the synthesized situational term is executed under a single, simple rule of inference that corresponds to sequential execution of subterms in a left-to-right order, with recursion and if-then-else.

A simpler way to execute programs of classified logic is to encode them in an OSA language such as OBJ3. As described earlier, the classified technique leaves to the implementor of a particular language such as OBJ3 the choice of strength for the type checking system. This encoding corresponds to a prototyping system and OBJ3 is already used this way, although with component specifications that

correspond to requirements decompositions rather than design decompositions (although these could be handled in OBJ3).

More ambitious future work could focus upon the use of classified specifications in planning applications more sophisticated than the blocks-world application.

Appendix A

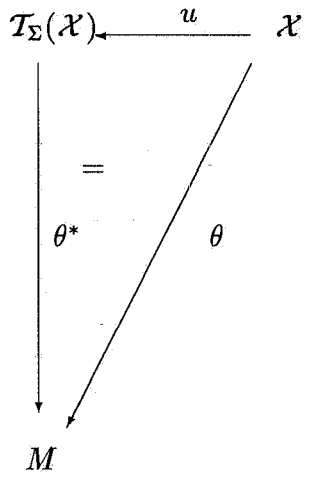
Classified Horn Logic With Equality

Classified Horn logic with equality is treated formally by giving model and proof theories for the specification language defined in Chapter 2, including theorems for soundness and completeness and the existence of a free model of a specification.

A.1 Free Σ−term Model

Theorem 1 (Free Σ−term Model) *The Σ−term model $\mathcal{T}_\Sigma(\mathcal{X})$ is a free Σ-model over $\mathcal{X}$ in the class of Σ-models.*

Proof: Let the inclusion $u: \mathcal{X} \rightarrow \mathcal{T}_\Sigma(\mathcal{X})$ (a sorted assignment) be the universal mapping of definition 17. Let M be a Σ−model with interpretation α. It must be shown that for any sorted assignment $\theta: \mathcal{X} \rightarrow D_M$ that the extended sorted assignment $\theta^*: D_{\mathcal{T}_\Sigma(\mathcal{X})} \rightarrow D_M$ is the required unique Σ−homomorphism such that $\theta = \theta^* \circ u$.



θ^* satisfies the requirements of a homomorphism (defn 15):

1. For function symbols, definition 11 for θ^* is applied directly.

2. For predicate symbols, there are three cases:

   (a) The identity predicate case can be shown easily by structural induction over terms.

   (b) Any sort symbol s has an interpretation in the term model $\mathcal{T}_\Sigma(\mathcal{X})$ as the set of variables with that subscript. Since $\theta^*[\![\mathsf{X}_\mathsf{s}]\!] = \theta[\![\mathsf{X}_\mathsf{s}]\!]$ and θ is a sorted assignment, the condition for a homomorphism is satisfied.

   (c) Any other predicate symbol has an empty truth set in the term model, so the condition for a homomorphism is satisfied vacuously.

To show that θ^* is the unique Σ-homomorphism which extends θ, assume there is another Σ-homomorphism

$$\gamma\colon D_{\mathcal{T}_\Sigma(\mathcal{X})} \to D_M$$

such that $\theta = \gamma \circ u$. It is shown below by structural induction that $\gamma = \theta^*$.

1. Base case for variables: $\theta^*[\![X_s]\!] = \theta[\![X_s]\!] = \gamma[\![X_s]\!]$ by assumption for all variable symbols X_s since γ extends θ.

2. Induction case: Assume $f(t_0, \ldots, t_{n-1}) \in D_{\mathcal{T}_\Sigma(X)}$ and $\theta^*[\![t_i]\!] = \gamma[\![t_i]\!]$ for $i = 0, \ldots, n-1$.

1.	$\theta^*[\![f(t_1, \ldots, t_n)]\!] = \alpha[\![f]\!](\theta^*[\![t_1]\!], \ldots, \theta^*[\![t_n]\!])$		defn 15 (homomorphism)
2.	$= \alpha[\![f]\!](\gamma[\![t_1]\!], \ldots, \gamma[\![t_n]\!])$		assumption
3.	$= \gamma[\![f(t_1, \ldots, t_n)]\!]$		defn 15 (homomorphism)

A.2 Quotient Models

Quotient models are important because they are used in the construction of a free model of a specification. The following definitions, which support the free model construction of the next section, are either standard or are minor variations of standard definitions.

Definition 20 (Equivalence relation) *A relation R on a set D is called an* equivalence relation *on D if R is reflexive, symmetric and transitive.*

Definition 21 (Congruence Relation) *Let Σ be a signature and M a Σ-model with universe D_M and interpretation function α, then an equivalence relation R on D_M, is called a* congruence relation *if for each $d_i, d_i' \in D_M$ such that $(d_i, d_i') \in R$ for $1 \leq i \leq n$ the following conditions hold:*

1. *for each n-ary function symbol* f, $(\alpha[\![f]\!](d_0, \ldots, d_{n-1}),\ \alpha[\![f]\!](d_0', \ldots, d_{n-1}')) \in R$

2. *for each n-ary predicate symbol* P,
   $<d_0, \ldots, d_{n-1}> \in \alpha[\![P]\!]$ *iff* $<d_0', \ldots, d_{n-1}'> \in \alpha[\![P]\!]$

Definition 22 (Congruence Class) *Let Σ be a signature, M be a Σ-model and R be a congruence relation on D_M. A congruence class $[a]$ of R is a subset of $D_M \times D_M$: $[a] = \{b : (b, a) \in R\}$*

Definition 23 (Kernel) *Let M and N be Σ-models and let $f: M \rightarrow N$ be a Σ-homomorphism. The* kernel of f, *$ker(f)$, is the set*

$$\{(a, b) : f(a) = f(b)\}.$$

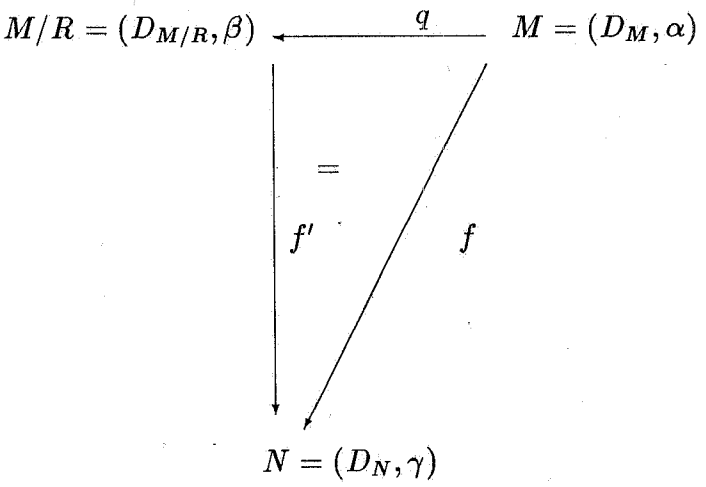
Note that $Ker(f)$ is *not necessarily* a congruence relation on M, a principle difference between this work and the equational case.

Definition 24 (Quotient Model $^{M}/_{R}$) *Let M be a Σ-model with universe D_M and interpretation function α. Let R be a congruence relation on D_M and let $d_i \in D_M$ for $i = 1, \ldots, n-1$. The quotient model $^{M}/_{R}$ has universe the set of R-congruence classes and interpretation function β such that:*

1. *for each n-ary function symbol* f: $\beta[\![\mathsf{f}]\!]([d_0], \ldots, [d_{n-1}]) = [\alpha[\![f]\!](d_0, \ldots, d_{n-1})]$
2. *for each n-ary predicate symbol* P:
   $<[d_0], \ldots, [d_{n-1}]> \in \beta[\![\mathsf{P}]\!]$ *iff* $<d_0, \ldots, d_{n-1}> \in \alpha[\![P]\!]$

The quotient model is well defined since the relation R is a congruence.

Theorem 2 (Universal Property Of Quotients) *Let Σ be a signature, M a Σ-model and R a congruence relation on M. Then $q: M \rightarrow {}^{M}/_{R}$ defined by $q(a) = [a]$ for $a \in M$ is a $\Sigma-$homomorphism (the quotient homomorphism) and satisfies the following universal property: Let N be a $\Sigma-$model and $f: M \rightarrow N$ be any homomorphism such that $R \subseteq ker(f)$, then there is a unique Σ-homomorphism $f': {}^{M}/_{R} \rightarrow N$ such that $f = f' \circ q$.*



Proof:

The mapping q is a $\Sigma-$homomorphism since definition 24 satisfies the homomorphism criteria in definition 15.

To prove the universal property of q define $f'{:}^{M/R} \rightarrow N$ by $f'([a]) = f(a)$ for all $a \in M$. Uniqueness is easily shown by observing that for any other $\Sigma-$homomorphism $f''{:}^{M/R} \rightarrow N$ with $f = f'' \circ q$ it must be the case that $f''([a]) = f(a)$ for all $a \in M$.

First, f' is well defined:

1.	$[a] = [b]$	Assumption
2.	$(a, b) \in R$	1, defn 22 (congruence class)
3.	$(a, b) \in ker(f)$	2, $R \subseteq ker(f)$
4.	$f(a) = f(b)$	3, def 23 (kernel)
5.	$f'([a]) = f'([b])$	4, def of f'

Second, f' is a $\Sigma-$homomorphism. Let α be the interpretation for M, β for $^{M/R}$ and γ for N.

For $\mathbf{g}$ an n-ary Σ−function symbol and $[\alpha[\![\mathsf{t_i}]\!]] \in D_{M/R}$ for $\mathsf{i} = 0, \ldots, \mathsf{n}-1$

1.	$f'(\beta[\![\mathbf{g}]\!](\beta[\![\mathsf{t}_0]\!], \ldots, \beta[\![\mathsf{t}_{n-1}]\!]) = f'(\beta[\![\mathbf{g}]\!]([\alpha[\![\mathsf{t}_0]\!]], \ldots, [\alpha[\![\mathsf{t}_{n-1}]\!]])$	defn of q homo
2.	$= f'([\alpha[\![\mathbf{g}]\!](\alpha[\![\mathsf{t}_0]\!], \ldots, \alpha[\![\mathsf{t}_{n-1}]\!])])$	defn of q homo
3.	$= f(\alpha[\![\mathbf{g}]\!](\alpha[\![\mathsf{t}_0]\!], \ldots, \alpha[\![\mathsf{t}_{n-1}]\!]))$	defn of f'
4.	$= \gamma[\![\mathbf{g}]\!](f(\alpha[\![\mathsf{t}_0]\!], \ldots, f(\alpha[\![\mathsf{t}_{n-1}]\!])))$	f a homo
5.	$= \gamma[\![\mathbf{g}]\!](f'([\alpha[\![\mathsf{t}_0]\!]], \ldots, f'([\alpha[\![\mathsf{t}_{n-1}]\!]])))$	defn of f'
6.	$= \gamma[\![\mathbf{g}]\!](f'(\beta[\![\mathsf{t}_0]\!], \ldots, f'(\beta[\![\mathsf{t}_{n-1}]\!])))$	defn of q

For P a n-ary Σ−predicate symbol and $[\alpha[\![\mathsf{t_i}]\!]] \in D_{M/R}$ for $\mathsf{i} = 0, \ldots, \mathsf{n}$:

1.	$<\beta[\![\mathsf{t}_0]\!], \ldots, \beta[\![\mathsf{t}_{n-1}]\!]> \in \beta[\![\mathsf{P}]\!]$	assumption
2.	$<[\alpha[\![\mathsf{t}_0]\!]], \ldots, [\alpha[\![\mathsf{t}_{n-1}]\!]]> \in \beta[\![\mathsf{P}]\!]$	defn of $\mathbf{q}$
3.	$<\alpha[\![\mathsf{t}_0]\!], \ldots, \alpha[\![\mathsf{t}_{n-1}]\!]> \in \alpha[\![\mathsf{P}]\!]$	defn of quotient
4.	$<f(\alpha[\![\mathsf{t}_0]\!]), \ldots, f(\alpha[\![\mathsf{t}_{n-1}]\!])> \in \gamma(\mathsf{P})$	f a homomorphism
5.	$<f'([\alpha[\![\mathsf{t}_0]\!]]), \ldots, f'([\alpha[\![\mathsf{t}_{n-1}]\!]])> \in \gamma(\mathsf{P})$	defn of f'
6.	$<f'(\beta[\![\mathsf{t}_0]\!]), \ldots, f'(\beta[\![\mathsf{t}_{n-1}]\!])> \in \gamma[\![\mathsf{P}]\!]$	defn of q

A.3 Proof Theory

The development of this proof theory parallels the Order Sorted Model proof theory of [GM87b].

A.3.1 Existence Of The S-model $\mathcal{T}_S(\mathsf{X})$

The free model for a specification $S = (\Sigma, \Gamma)$ is based upon a congruence $\equiv_S$ on $\Sigma(\mathsf{X})$-terms. This congruence is generated by the assertions Γ and, because it is based on provability using the rules of inference, results in a simple proof

of completeness for these rules. This is the same congruence and free model construction used in [GM87b].

Theorem 3 (Congruence $\equiv_S$) *Let $S = (\Sigma, \Gamma)$ be a specification and let* t *and* t' *be $\Sigma(\mathcal{X})$-terms, then the property*

$\mathsf{t} \equiv_S \mathsf{t}'$ *iff* $(\mathcal{X})$ $\mathsf{t} == \mathsf{t}'$ *is derivable from Γ using the rules of inference 1-5*

defines a congruence relation $\equiv_S$ on $\Sigma(\mathcal{X})$-terms.

Proof: The first three rules of inference (see Chapter 2) define an equivalence relation and the fourth ensures that the property defines a congruence relation.

Definition 25 (Quotient Term Model $\mathcal{T}_\Sigma(\mathcal{X})/_{\equiv_S}$) *Let $S = (\Sigma, \Gamma)$ be a specification and let $\mathcal{X}$ be the set of variables in S, then the quotient term model is $\mathcal{T}_\Sigma(\mathcal{X})/_{\equiv_S}$.*

By definition 10 for $\mathcal{T}_\Sigma(\mathcal{X})$ and definition 24 for a quotient model, $\mathcal{T}_\Sigma(\mathcal{X})/_{\equiv_S}$ has a universe the set of congruence classes of $\equiv_S$ and an interpretation function α such that

1. for the identity predicate symbol:
   $\alpha[\![==]\!] = \{<d, d> : d \text{ is a congruence class of } \equiv_S \}$;
2. any sort predicate symbol s is interpreted as just the congruence classes of the sorted variables that have that sort subscript, i.e.,
   $\alpha[\![\mathsf{s}]\!] = \{[\mathsf{X}_\mathsf{s}] : \mathsf{X}_\mathsf{s} \text{ is a variable with sort subscript } \mathsf{s} \}$;
3. any other predicate symbol P is interpreted as the empty set, i.e., $\alpha[\![\mathsf{P}]\!] = \{\}$;

Definition 26 (Specification Term Model $\mathcal{T}_S(\mathcal{X})$) *Let $S = (\Sigma, \Gamma)$ be a specification and let $\mathcal{X}$ be the set of variables in S, then the specification term model $\mathcal{T}_S(\mathcal{X})$ is just like the quotient term model, except that the interpretation function α gives an S-model structure for each n-ary predicate symbol P in Σ:*

$\alpha[\![\mathsf{P}]\!] = \{<[\mathsf{t}_0], \ldots, [\mathsf{t}_{n-1}]> : (\mathcal{X})\ \mathsf{P}(\mathsf{t}_0, \ldots, \mathsf{t}_{n-1})$ *is derivable from* Γ *using the rules of inference 1-5* $\}$.

The specification term model $\mathcal{T}_S(\mathsf{X})$ is well defined because the relation generated by provability using these rules of inference is a congruence in which (by definition using rule 4) any choice of representatives for the terms of the quotient model will suffice in the interpretation. That is, by inference rule 4 the definition is independent of the representatives t_i.

The following theorem shows that the quotient term model defined above also satisfies the specification $S = (\Sigma, \Gamma)$, i.e., it is a $S-$model.

Theorem 4 (S-model $\mathcal{T}_S(\mathcal{X})$) *$\mathcal{T}_S(\mathcal{X})$ is a $S-$model.*

Proof: Let $S = (\Sigma, \Gamma)$ be a specification and let $(\mathcal{Y})\ \mathsf{B}_0, \ldots, \mathsf{B}_{m-1} \rightarrow \mathsf{A}$ be an assertion in Γ, where

1. A is of the form $\mathsf{Q}(\mathsf{t}_0, \ldots, \mathsf{t}_{n-1})$,
2. B_i is of the form $\mathsf{P}_i(\mathsf{t}^i_0, \ldots, \mathsf{t}^i_{n_i-1})$ for all $0 \leq i \leq m-1$.

Assume a specification term model $\mathcal{T}_S(\mathcal{X})$ as defined above with interpretation function α.

It is required to show $\alpha^*[\![(\mathcal{Y})\ \mathsf{B}_0, \ldots, \mathsf{B}_{m-1} \rightarrow \mathsf{A}]\!] = T$. This is done, according to definition 13 (Truth In A Model), by selecting an arbitrary sorted assignment

function θ_0 and showing, using definition 11 (Assignment Function), that any ground instance of the assertion is true in the model $\mathcal{T}_S(\mathcal{X})$.

Let $\theta_0: \mathcal{Y} \rightarrow D_{\mathcal{T}_S(\mathcal{X})}$ be an arbitrary sorted assignment function sending $\mathsf{Y_s}$ in $\mathcal{Y}$ to $[\mathsf{t}] \in D_{\mathcal{T}_S(\mathcal{X})}$. By definition 25 we can choose a representative $\mathsf{t} \in D_{\mathcal{T}_\Sigma(\mathcal{X})}$ for each $[\mathsf{t}] = \theta_0(\mathsf{Y_s})$ such that this function can be factored as:

$$\theta_0 = q \circ \theta$$

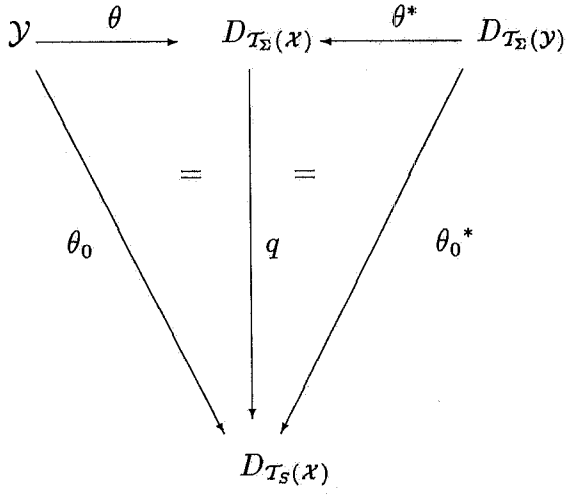
where

1. $\theta: \mathcal{Y} \rightarrow D_{\mathcal{T}_\Sigma(\mathcal{X})}$ is a substitution;

2. $q: D_{\mathcal{T}_\Sigma(\mathcal{X})} \rightarrow D_{\mathcal{T}_S(\mathcal{X})}$ is the quotient homomorphism that sends t to $[\mathsf{t}]$ and is extended in definition 26;

3. by defn 26), $(\mathcal{X})\ \mathsf{s}(\theta \mathsf{Y_s})$ is derivable from Γ for each $\mathsf{Y_s} \in S$ since θ_0 is a sorted assignment. Similarly for the extension θ^* of θ: $(\mathcal{X})\ \mathsf{s}(\theta^* \mathsf{Y_s})$ is derivable.

Since by Theorem 1 $\mathcal{T}_\Sigma(\mathcal{Y})$ is the free Σ-model over $\mathcal{Y}$ in the class of all Σ-models there is only one homomorphism from $\mathcal{T}_\Sigma(\mathcal{Y})$ to any Σ-model, i.e., $\theta_0^* = q \circ \theta^*$.



1. Suppose $<\theta_0^*[\![t_0^i]\!], \ldots, \theta_0^*[\![t_{n_i-1}^i]\!]> \in \alpha[\![P_i]\!]$ for $i = 0, \ldots, m-1$
2. $<[\theta^* t_0^i], \ldots, [\theta^* t_{n_i-1}^i]> \in \alpha[\![P_i]\!]$ for $i = 0, \ldots, m-1$ — 1, since $\theta_0^* = q \circ \theta^*$
3. $(\mathcal{X})$ $\theta^* B_i$, $i = 0, \ldots, m$ are derivable — 2, defn 26 (truth set for a predicate)
4. $(\mathcal{X})$ $s(\theta^* Y_s)$ is derivable — Shown above
5. $(\mathcal{X})$ $\theta^* A$ is derivable — 3,4 by Modus Ponens rule
6. $<[\theta^*[\![t_0]\!]], \ldots, [\theta^*[\![t_{n-1}]\!]]> \in \alpha[\![Q]\!]$ — 5, defn 26 (truth set for a predicate)
7. $<\theta_0^*[\![t_0]\!], \ldots, \theta_0^*[\![t_{n-1}]\!]> \in \alpha[\![Q]\!]$ — 6, since $\theta_0^* = q \circ \theta^*$.

Since the conditions of definition 13 for truth in a model are satisfied:

$$\alpha^*[\![(\mathcal{Y})\ B_0, \ldots, B_{m-1} \rightarrow A]\!] = T.$$

A.3.2 Soundness And Completeness Of The Rules Of Inference

Theorem 5 (Soundness) *Any formula* $(\mathcal{X})$ $P(t_0, \ldots, t_{n-1})$ *that is derivable from a specification S by rules 1-5 is satisfied in all S-models.*

Proof: To prove soundness of each rule it is sufficient to show that the validity of the formulas in the premise of the rule implies the validity of the formulas in the conclusion of the rule, i.e., an untrue conclusion cannot be derived from a true premise. Let:

- Σ be a signature and let M be an arbitrary Σ-model with universe D_M and interpretation function α.
- θ be an arbitrary sorted assignment $\theta: \mathcal{X} \rightarrow D_M$.

The proof of soundness of each rule follows the statement of the rule.

1. Reflexivity: $(\mathcal{X})$ $t == t$ is derivable. By induction on the structure of terms:

(a) Base case: $(\mathsf{X_s})\ \mathsf{X_s} == \mathsf{X_s}$

1. for all $d \in D_M$, $<d, d> \in \alpha[\![==]\!]$ — Σ-Model defn 9
2. $<\theta^*[\![\mathsf{X_s}]\!], \theta^*[\![\mathsf{X_s}]\!]> \in \alpha[\![==]\!]$ — 1, assignment defn 11
3. $\alpha^*[\![(\mathsf{X_s})\ \mathsf{X_s} == \mathsf{X_s}]\!] = T$ — 2, truth for quantification defn 13

(b) Induction step: Assume $\alpha^*[\![(\mathcal{X}_\mathsf{i})\ \mathsf{t_i} == \mathsf{t_i}]\!] = T$ for $\mathsf{i} = 0, \ldots, \mathsf{n}-1$ and prove

$\alpha^*[\![(\mathcal{X})\ \mathsf{f}(\mathsf{t_0}, \ldots, \mathsf{t_{n-1}}) == \mathsf{f}(\mathsf{t_0}, \ldots, \mathsf{t_{n-1}})]\!] = T$, where $\mathcal{X} = \cup\mathcal{X}_\mathsf{i}$ and f is an n-ary function symbol.

1. $\theta^*[\![\mathsf{t_i}]\!] = \theta^*[\![\mathsf{t_i}]\!]$, $\mathsf{i} = 1, \ldots, \mathsf{n}$ — assumption by defn 13, defn 11
2. $\alpha[\![\mathsf{f}]\!](\theta^*[\![\mathsf{t_0}]\!], \ldots, \theta^*[\![\mathsf{t_{n-1}}]\!]) = \alpha[\![\mathsf{f}]\!](\theta^*[\![\mathsf{t_0}]\!], \ldots, \theta^*[\![\mathsf{t_{n-1}}]\!])$ — 1, $D_M =$
3. $\theta^*[\![\mathsf{f}(\mathsf{t_0}, \ldots, \mathsf{t_{n-1}})]\!] = \theta^*[\![\mathsf{f}(\mathsf{t_0}, \ldots, \mathsf{t_{n-1}})]\!]$ — 2, assignment defn 11
4. $\theta^*[\![\mathsf{f}(\mathsf{t_0}, \ldots, \mathsf{t_{n-1}}) == \mathsf{f}(\mathsf{t_0}, \ldots, \mathsf{t_{n-1}})]\!] = T$ — 3, assignment defn 11
5. $\alpha^*[\![(\mathcal{X})\ \mathsf{f}(\mathsf{t_0}, \ldots, \mathsf{t_{n-1}}) == \mathsf{f}(\mathsf{t_0}, \ldots, \mathsf{t_{n-1}})]\!] = T$ — 4, defn 13

2. Symmetry: If $(\mathcal{X})\ \mathsf{t} == \mathsf{t}'$ is derivable then $(\mathcal{X})\ \mathsf{t}' == \mathsf{t}$ is derivable.

1. Assume $\alpha^*[\![(\mathcal{X})\ \mathsf{t} == \mathsf{t}']\!] = T$
2. $\theta^*[\![\mathsf{t} == \mathsf{t}']\!] = T$ — 1, truth for quantification defn 13
3. $\theta^*[\![\mathsf{t}]\!] = \theta^*[\![\mathsf{t}']\!]$ — 2, assignment defn 11
4. $\theta^*[\![\mathsf{t}']\!] = \theta^*[\![\mathsf{t}]\!]$ — 3, symmetry in D_M
5. $\theta^*[\![\mathsf{t}' == \mathsf{t}]\!] = T$ — 4, assignment defn 11
6. $\alpha^*[\![(\mathcal{X})\ \mathsf{t}' == \mathsf{t}]\!] = T$ — 5, truth for quantification defn 13

3. Transitivity: If $(\mathcal{X}')\ \mathsf{t} == \mathsf{t}'$ and $(\mathcal{X}'')\ \mathsf{t}' == \mathsf{t}''$ are derivable and if $\mathcal{X} = \mathcal{X}' \cup \mathcal{X}''$ then $(\mathcal{X})\ \mathsf{t} == \mathsf{t}''$ is derivable.

1. Assume $\alpha^*[\![(\mathcal{X}')\ \mathsf{t} == \mathsf{t}']\!] = T$ and $\alpha^*[\![(\mathcal{X}'')\ \mathsf{t}' == \mathsf{t}'']\!] = T$

2. $\theta^*[\![\mathsf{t} == \mathsf{t}']\!] = T$ and $\theta^*[\![\mathsf{t}' == \mathsf{t}'']\!] = T$ 1, truth for quantification defn 13
3. $\theta^*[\![\mathsf{t}]\!] = \theta^*[\![\mathsf{t}']\!]$ and $\theta^*[\![\mathsf{t}']\!] = \theta^*[\![\mathsf{t}'']\!]$ 2, assignment defn 11
4. $\theta^*[\![\mathsf{t}]\!] = \theta^*[\![\mathsf{t}'']\!]$ 3, transitivity in D_M
5. $\theta^*[\![\mathsf{t} == \mathsf{t}'']\!] = T$ 4, assignment defn 11
6. $\alpha^*[\![(\mathcal{X})\ \mathsf{t} == \mathsf{t}'']\!] = T$ 5, truth for quantification defn 13

4. Substitutivity: if $(\mathcal{X}_\mathsf{i})\ \mathsf{t}_\mathsf{i} == \mathsf{t}'_\mathsf{i}$ is derivable and if $\mathcal{X} = \cup\mathcal{X}_i$ for $\mathsf{i} = 1, \ldots, \mathsf{n}$ then:

   (a) for any n-ary function symbol f in Σ,

   $(\mathcal{X})\ \mathsf{f}(\mathsf{t}_0, \ldots, \mathsf{t}_{n-1}) == \mathsf{f}(\mathsf{t}'_0, \ldots, \mathsf{t}'_{n-1})$ is derivable;

   1. Assume $\alpha^*[\![(\mathcal{X}_\mathsf{i})\ \mathsf{t}_\mathsf{i} == \mathsf{t}'_\mathsf{i}]\!] = T$, $\mathsf{i} = 0, \ldots, \mathsf{n}-1$ hypothesis
   2. $\theta^*[\![\mathsf{t}_\mathsf{i}]\!] = \theta^*[\![\mathsf{t}'_\mathsf{i}]\!]$, $\mathsf{i} = 1, \ldots, \mathsf{n}$ 1, defn 11, 13
   3. $\alpha[\![(\mathcal{X})\ \mathsf{f}(\mathsf{t}_0, \ldots, \mathsf{t}_{n-1}) == \mathsf{f}(\mathsf{t}_0, \ldots, \mathsf{t}_{n-1})]\!] = T$ rule 1 soundness
   4. $\theta^*[\![\mathsf{f}(\mathsf{t}_0, \ldots, \mathsf{t}_{n-1})]\!] = \theta^*[\![\mathsf{f}(\mathsf{t}_0, \ldots, \mathsf{t}_{n-1})]\!]$ 3, defn 11, 13
   5. $\alpha[\![\mathsf{f}]\!](\theta^*[\![\mathsf{t}_0]\!], \ldots, \theta^*[\![\mathsf{t}_{n-1}]\!]) = \alpha[\![\mathsf{f}]\!](\theta^*[\![\mathsf{t}_0]\!], \ldots, \theta^*[\![\mathsf{t}_{n-1}]\!])$ 4, defn 11
   6. $\alpha[\![\mathsf{f}]\!](\theta^*[\![\mathsf{t}_0]\!], \ldots, \theta^*[\![\mathsf{t}_{n-1}]\!]) = \alpha[\![\mathsf{f}]\!](\theta^*[\![\mathsf{t}'_0]\!], \ldots, \theta^*[\![\mathsf{t}'_{n-1}]\!])$ 2,5, D_M subs
   7. $\theta^*[\![\mathsf{f}(\mathsf{t}_0, \ldots, \mathsf{t}_{n-1})]\!] = \theta^*[\![\mathsf{f}(\mathsf{t}'_0, \ldots, \mathsf{t}'_{n-1})]\!]$ 6, defn 11
   8. $\alpha^*[\![(\mathcal{X})\ \mathsf{f}(\mathsf{t}_0, \ldots, \mathsf{t}_{n-1}) == \mathsf{f}(\mathsf{t}'_0, \ldots, \mathsf{t}'_{n-1})]\!] = T$ 7, defn 11, 13

   (b) for any n-ary predicate symbol P in Σ, if $(\mathcal{X})\ \mathsf{P}(\mathsf{t}_0, \ldots, \mathsf{t}_{n-1})$ is derivable then $(\mathcal{X})\ \mathsf{P}(\mathsf{t}'_0, \ldots, \mathsf{t}'_{n-1})$ is derivable.

   1. Assume $\alpha^*[\![(\mathcal{X}_\mathsf{i})\ \mathsf{t}_\mathsf{i} == \mathsf{t}'_\mathsf{i}]\!] = T$, $\mathsf{i} = 1, \ldots, \mathsf{n}$ hypothesis
   2. $\theta^*[\![\mathsf{t}_\mathsf{i}]\!] = \theta^*[\![\mathsf{t}'_\mathsf{i}]\!]$, $\mathsf{i} = 1, \ldots, \mathsf{n}$ 1, defn 11, 13,
   3. $\alpha^*[\![(\mathsf{X})\ \mathsf{P}(\mathsf{t}_0, \ldots, \mathsf{t}_{n-1})]\!] = T$ hypothesis
   4. $<\theta^*[\![\mathsf{t}_0]\!], \ldots, \theta^*[\![\mathsf{t}_{n-1}]\!]> \in \alpha(P)$ 3, defn 11, 13

5. $<\theta^*[\![t'_0]\!], \ldots, \theta^*[\![t'_{n-1}]\!]> \in \alpha(P)$ 2,4, substitution in D_M
6. $\alpha^*[\![(\mathcal{X}')\ P(t'_0, \ldots, t'_{n-1}) = T$ 5, defn 11, 13

5. Modus Ponens: If $(\mathcal{X})\ B_0, \ldots, B_{m-1} \rightarrow A$ is derivable and if $\theta: \mathcal{X} \rightarrow \Sigma(\mathcal{Y})$ is any substitution such that:

   (a) each $(\mathcal{Y})\ \theta B_i$ is derivable for $i = 0, \ldots m-1$, and

   (b) $(\mathcal{Y})\ s(\theta X_s)$ is derivable for each $X_s \in \mathcal{X}$,

   then $(\mathcal{Y})\ \theta A$ is derivable.

   Let $\theta_0: \mathcal{Y} \rightarrow D_M$ be an arbitrary sorted assignment.

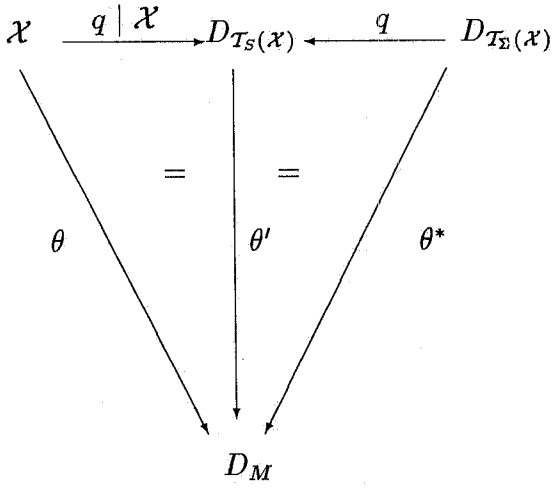
1. Assume $\alpha^*[\![(\mathcal{Y})\ \theta B_i]\!] = T$, $i = 0, \ldots m-1$ hypothesis
2. $\theta_0^*[\![\theta B_i]\!] = T$, $i = 0, \ldots m-1$ 1, truth defn 13
3. Assume $\alpha^*[\![(\mathcal{X})\ B_0, \ldots, B_{m-1} \rightarrow A]\!] = T$ hypothesis
4. if $\theta_0^*[\![\theta B_i]\!] = T$ $i = 0, \ldots m-1$ then $\theta_0^*[\![\theta A]\!] = T$ 3, truth defn 13
5. $\theta_0^*[\![\theta A]\!] = T$ 2,4 propositional logic
8. $\alpha^*[\![(\mathcal{Y})\ \theta A]\!] = T$ 5, truth defn 13,

Theorem 6 (Completeness) *For any specification $S = (\Sigma, \Gamma)$, any formula $(\mathcal{X})\ P(t_0, \ldots, t_{n-1})$ that is satisfied in all S-models is derivable from S by rules 1-5.*

Proof: Suppose $(\mathcal{X})\ P(t_0, \ldots, t_{n-1})$ is satisfied by all $S-$models, then it is satisfied by the specification term model $\mathcal{T}_S(\mathcal{X})$ defined above and $<[t_0], \ldots, [t_{n-1}]> \in \alpha[\![P]\!]$. Then, by definition 26 of $\alpha[\![P]\!]$, $(\mathcal{X})\ P(t_0, \ldots, t_{n-1})$ is derivable by rules 1-5.

Theorem 7 ($\mathcal{T}_S(\mathcal{X})$ Is A Free $S-$Model) *Let $S = (\Sigma, \Gamma)$ be a specification, then $\mathcal{T}_S(\mathcal{X})$ is a free $S-$model over $\mathcal{X}$ in the class of all $S-$models.*

Proof: Let $S = (\Sigma, \Gamma)$ be a specification, let $q: D_{\mathcal{T}_\Sigma(\mathcal{X})} \rightarrow D_{\mathcal{T}_S(\mathcal{X})}$ be the quotient $\Sigma-$homomorphism and α be the interpretation function of definitions 25 and 26. Let M be a $S-$model with interpretation function β and let $\theta: \mathcal{X} \rightarrow D_M$ be an arbitrary sorted assignment. It is required to show that there is a unique $\Sigma-$homomorphism $\theta': D_{\mathcal{T}_S(\mathcal{X})} \rightarrow D_M$ such that $\theta = \theta' \circ q$. The proof proceeds in two steps. First the existence of θ' is shown and then its uniqueness.



The existence of θ' is shown in two parts corresponding to the two part definition of the quotient homomorphism q in definition 25 and its extension in definition 26.

By the soundness theorem, each derivable equation $(\mathcal{X})$ $\mathsf{t}_1 == \mathsf{t}_2$ is satisfiable in M. That is, $\theta^*[\![\mathsf{t}_1]\!] = \theta^*[\![\mathsf{t}_2]\!]$ which means that $\equiv_S \subseteq ker(\theta^*)$. Applying Theorem 2, the universal property of quotients, there is a unique homomorphism $\theta': \mathcal{T}_\Sigma(\mathcal{X})/_{\equiv_S} \rightarrow M$ such that $\theta^* = \theta' \circ q$.

The second part of the existence proof deals with the homomorphism condition for predicates (definition 15).

1. $<[\mathsf{t}_0],\ldots,[\mathsf{t}_{n_i}]> \in \alpha[\![\mathsf{P}]\!]$ iff — By def 26 for representatives t_i of $[\mathsf{t}_i]$
2. $(\mathcal{X})$ $\mathsf{P}(\mathsf{t}_0,\ldots\mathsf{t}_{n-1})$ is derivable from S iff — Soundness, Completeness Thm
3. $(\mathcal{X})$ $\mathsf{P}(\mathsf{t}_0,\ldots\mathsf{t}_{n-1})$ is satisfied in all S-models.
4. $<\theta^*[\![\mathsf{t}_0]\!],\ldots,\theta^*[\![\mathsf{t}_{n_i}]\!]> \in \beta[\![\mathsf{P}]\!]$ — 3, M is an S-model.

The uniqueness of θ' is shown by assuming there is another homomorphism $\theta'': D_{\mathcal{T}_S(\mathcal{X})} \rightarrow D_M$ such that $\theta = \theta'' \circ q$. Since $\mathcal{T}_\Sigma(\mathcal{X})$ is a free model on X, θ is uniquely extended to θ^* such that $\theta^* = \theta'' \circ q$. By theorem 2 (the universal property of quotients) there is only one homomorphism satisfying this property and so $\theta' = \theta''$.

A.4 Translating CM To An Unsorted Horn Subset

In Chapter 1 it is claimed that a CM specification is co-extensive with its translation to a Horn specification with unsorted quantifiers. To show this, it is necessary to:

1. define a translation from the CM language to a subset of the Horn language with unsorted quantifiers;
2. show that a CM specification is co-extensive with its translation by proving that logical consequence in CM agrees with logical consequence in the Horn subset.

Any CM assertion is easily relativized to an assertion in a Horn subset by omitting the sort subscripts from sorted variable symbols and by including in the antecedent of the assertion a sort predicate for each of the altered variables.

Definition 27 (Sort Translation) *Let $S = (\Sigma, \Gamma)$ be a CM specification with variables $\mathcal{X}$. Define a sort translation function γ by:*

1. *for each variable symbol* $\mathsf{X}^{\mathsf{i}}_{\mathsf{s}_\mathsf{i}} \in \mathcal{X}$, $\gamma(\mathsf{X}^{\mathsf{i}}_{\mathsf{s}_\mathsf{i}}) = \mathsf{X}^{\mathsf{i}}$;
2. *for each n-ary function symbol* f, $\gamma(\mathsf{f}(\mathsf{t}_0, \ldots, \mathsf{t}_{n-1})) = \mathsf{f}(\gamma\mathsf{t}_0, \ldots, \gamma\mathsf{t}_{n-1})$
3. *for each n-ary predicate symbol* P, $\gamma(\mathsf{P}(\mathsf{t}_0, \ldots, \mathsf{t}_{n-1})) = \mathsf{P}(\gamma\mathsf{t}_0, \ldots, \gamma\mathsf{t}_{n-1})$
4. *for each Horn formula,* $\gamma((\mathsf{X}^0_{\mathsf{s}_0}, \ldots, \mathsf{X}^{n-1}_{\mathsf{s}_{n-1}})\, \mathsf{B}_0, \ldots, \mathsf{B}_{m-1} \rightarrow \mathsf{A}) =$
   $(\mathsf{X}^0, \ldots, \mathsf{X}^{n-1})\, \mathsf{s}_0(\mathsf{X}^0), \ldots, \mathsf{s}_{n-1}(\mathsf{X}^{n-1}), \gamma\mathsf{B}_0, \ldots, \gamma\mathsf{B}_{m-1} \rightarrow \gamma\mathsf{A}$

If Γ is a set of CM assertions, then $\gamma(\Gamma)$ is the sorted translation of Γ.

The definition of truth in a model for a translated formula is like definition 13 for a CM formula, except θ is an assignment function instead of a sorted assignment function.

Definition 28 (Truth In A Model (Unsorted Variables)) *The* truth value *of a closed formula $(\mathcal{X})\,\mathcal{F}$ with unsorted variables in a model M with universe D_M and interpretation function α is defined by extending α to a function α^*_u:*

$\alpha^*_u[\![(\mathcal{X})\,\mathcal{F}]\!] = T$ *if* $\theta^*_u[\![\mathcal{F}]\!] = T$ *for any (unsorted) assignment* $\theta_u : \mathcal{X} \rightarrow D_M$, *and* $\alpha^*_u[\![(\mathcal{X})\,\mathcal{F}]\!] = F$ *otherwise.*

Satisfaction ($\models$) with α^*_u is defined analogously to the α^* case.

A CM specification can be related to its unsorted relativisation by the following theorem which shows that a CM formula $(\mathcal{X})\,\mathcal{F}$ is a logical consequence of a CM specification iff the translation of $(\mathcal{X})\,\mathcal{F}$ is a logical consequence of the translation of the CM specification. First, a sort lemma:

Lemma 1 (Sort) *Let $S = (\Sigma, \Gamma)$ be a CM specification, let γ be a sort translation function, let $M = (D_M, \alpha)$ be any S-model and suppose $(\mathcal{X})\,\mathcal{F} \in \Gamma$, then*

$$\alpha^*[\![(\mathcal{X})\,\mathcal{F}]\!] = T \text{ iff } \alpha_u^*[\![\gamma((\mathcal{X})\,\mathcal{F})]\!] = T$$

Proof: Let $(\mathcal{X})\,\mathcal{F}$ be $(\mathsf{X}^0_{\mathsf{s}_0}, \ldots, \mathsf{X}^{\mathsf{n}-1}_{\mathsf{s}_{\mathsf{n}-1}})\,\mathsf{B}_0, \ldots, \mathsf{B}_{\mathsf{m}-1} \rightarrow \mathsf{A}$, let θ be any *sorted* assignment and let $\theta = \theta_u \circ \gamma$, then:

1. $\alpha^*[\![(\mathcal{X})\,\mathcal{F}]\!] = T$ iff — defn 13
2. $\theta^*[\![\mathsf{B}_0, \ldots, \mathsf{B}_{\mathsf{m}-1} \rightarrow \mathsf{A}]\!] = T$ iff — defn 11
3. if $\theta^*[\![\mathsf{B}_0]\!] = T, \ldots, \theta^*[\![\mathsf{B}_{\mathsf{m}-1}]\!] = T$ then $\theta^*[\![\mathsf{A}]\!] = T$ iff — $\theta = \theta_u \circ \gamma$,
4. if $\theta_u^*[\![\mathsf{s}_0(\mathsf{X}^0)]\!] = T, \ldots, \theta_u^*[\![\mathsf{s}_{\mathsf{n}-1}(\mathsf{X}^{\mathsf{n}-1})]\!] = T, \theta_u^*[\![\gamma\mathsf{B}_0]\!] = T, \ldots, \theta_u^*[\![\gamma\mathsf{B}_{\mathsf{m}-1}]\!] = T$ then $\theta_u^*[\![\gamma\mathsf{A}]\!] = T$ iff — defn 11
5. $\theta_u^*[\![\mathsf{s}_0(\mathsf{X}^0), \ldots, \mathsf{s}_{\mathsf{n}-1}(\mathsf{X}^{\mathsf{n}-1}), \gamma\mathsf{B}_0, \ldots, \gamma\mathsf{B}_{\mathsf{m}-1} \rightarrow \gamma\mathsf{A}]\!] = T$ iff — defn 13
6. $\alpha_u^*[\![\gamma((\mathcal{X})\,\mathcal{F})]\!] = T$

Theorem 8 (Sort) *Let $S = (\Sigma, \Gamma)$ be a CM specification and let γ be a sort translation function, then*

$$\Gamma \models (\mathcal{X})\,\mathcal{F} \text{ iff } \gamma(\Gamma) \models \gamma((\mathcal{X})\,\mathcal{F})$$

Proof: Let $M = (D_M, \alpha)$ be a Σ-model.

1. Assume $\Gamma \models (\mathcal{X})\,\mathcal{F}$
2. Assume for all $\gamma((\mathcal{Y})\,\mathcal{G}) \in \gamma(\Gamma)$, $M \models \gamma((\mathcal{Y})\,\mathcal{G})$
3. for all $\gamma((\mathcal{Y})\,\mathcal{G}) \in \gamma(\Gamma)$, $\alpha_u^*[\![\gamma((\mathcal{Y})\,\mathcal{G})]\!] = T$ — 2, defn 14
4. for all $(\mathcal{Y})\,\mathcal{G} \in \Gamma$, $\alpha^*[\![(\mathcal{Y})\,\mathcal{G}]\!] = T$ — 3, Sort Lemma
5. $M \models \Gamma$, i.e. M is an S-model — 1,4 defn 14

6. $M \models (\mathcal{X})\,\mathcal{F}$ — 1,5 defn 14
7. $\alpha^*[\![(\mathcal{X})\,\mathcal{F}]\!] = T$ — 6, defn 14
8. $\alpha_u^*[\![\gamma((\mathcal{X})\,\mathcal{F})]\!] = T$ — 7, Sort Lemma
9. $M \models \gamma((\mathcal{X})\,\mathcal{F})$ — 8, defn 14
10. $\gamma(\Gamma) \models \gamma((\mathcal{X})\,\mathcal{F})$ — 2,9 defn 14

1. Assume $\gamma(\Gamma) \models \gamma((\mathcal{X})\,\mathcal{F})$
2. Assume for all $(\mathcal{Y})\,\mathcal{G} \in \Gamma$, $M \models (\mathcal{Y})\,\mathcal{G}$
3. for all $(\mathcal{Y})\,\mathcal{G} \in \Gamma$, $\alpha^*[\![(\mathcal{Y})\,\mathcal{G}]\!] = T$ — 2, defn 14
4. for all $\gamma((\mathcal{Y})\,\mathcal{G}) \in \gamma(\Gamma)$, $\alpha_u^*[\![\gamma((\mathcal{Y})\,\mathcal{G})]\!] = T$ — 3, Sort Lemma
5. $M \models \gamma(\Gamma)$ — 1,4 defn 14
6. $M \models \gamma((\mathcal{X})\,\mathcal{F})$ — 1,5 defn 14
7. $\alpha_u^*[\![\gamma((\mathcal{X})\,\mathcal{F})]\!] = T$ — 6, defn 14
8. $\alpha^*[\![(\mathcal{X})\,\mathcal{F}]\!] = T$ — 7, Sort Lemma
9. $M \models (\mathcal{X})\,\mathcal{F}$ — 8, defn 14
10. $\Gamma \models (\mathcal{X})\,\mathcal{F}$ — 2,9 defn 14

A.5 CM Induction Principle

Let $S = (\Sigma, \Gamma)$ be a CM specification, let $\mathsf{s} \in \Sigma$ be a sort symbol and let $(\mathsf{X_s}, \mathsf{X_{s_1}}, \ldots, \mathsf{X_{s_k}})\,\mathcal{P}$ be a closed atomic formula to prove by induction on the variable $\mathsf{X_s}$.

Suppose also that S has a set of variable symbols $\mathcal{X}$ and that the declaration assertions of S for sort $\mathsf{s} \in \Sigma$ are $(\mathcal{X}^{\mathsf{i}})\,\mathcal{A}_{\mathsf{i}} \rightarrow \mathsf{s}(\mathsf{t}_{\mathsf{i}})$, $\mathsf{i} = 1, \ldots, \mathsf{n}$, where the body $\mathcal{A}_{\mathsf{i}}$ of each declaration assertion is either empty or a set of atomic assertions. $\mathcal{X}^{\mathsf{i}} \subset \mathcal{X}$ is the set of variables occurring in the i-th declaration assertion and $\mathcal{X}^{\mathsf{i}}_{\mathsf{s}} \subset \mathcal{X}^{\mathsf{i}}$ is the subset of s-sorted variables in the i-th declaration assertion.

Define also:

1. $\gamma_i: \mathcal{X}_s^j \to \mathcal{C}_i$ for $i = 1, \ldots, n$, is a sorted substitution of all the variables of sort s in the i-th declaration assertion by new constant symbols not already appearing in S, where $\mathcal{C} = \cup \mathcal{C}_i$ and all $\mathcal{C}_i$ are disjoint;

2. $s(c_j)$ for all $c_j \in \mathcal{C}$, $j = 1, \ldots, m$, i.e., the new constant symbols are all of sort s;

3. $\sigma_j: X_s \to c_j$, $j = 1, \ldots, m$, is a substitution for the variable X_s of $\mathcal{P}$ by the j-th new constant. This substitution results in an induction hypothesis.

4. $\sigma_i': X_s \to \gamma_i t_i$, $i = 1, \ldots, n$, is a substitution for the variable X_s of $\mathcal{P}$ by the (transformed) term of the head of the i-th declaration assertion. The transformation γ_i makes the term t ground in the sort s. This substitution results in a base case to prove when the term t_i contains no variable symbols of sort s and in an induction step to prove otherwise.

The CM Induction Principle is:

If by assuming the *induction hypothesis* that the following are all derivable:

$$(X_{s_1}, \ldots, X_{s_k})\, \sigma_j \mathcal{P},\ j = 1, \ldots, m$$

we can derive all of the following *desired conclusions*:

$$(X_{s_1}, \ldots, X_{s_k})\, \sigma_i' \mathcal{P},\ i = 1, \ldots, n$$

then we can conclude

$$(X_s, X_{s_1}, \ldots, X_{s_k})\, \mathcal{P}$$

in the initial model.

The proof that the CM induction principle holds in the initial model $\mathcal{T}_S$ is by induction on the number of quantifiers in the following base and induction cases.

Theorem 9 (CM Induction Principle, Base Case) *The CM induction principle holds for formulas quantified over just one variable.*

Proof: Let $\mathcal{P}$ be $\mathsf{P}(\mathsf{X_s})$ and suppose that the induction principle is applied to $(\mathsf{X_s})\,\mathsf{P}(\mathsf{X_s})$. It is necessary to show in the free model $\mathcal{T}_S(\mathcal{X})$ with interpretation function α that $\alpha[\![\mathsf{t}]\!] \in \alpha[\![\mathsf{s}]\!]$ implies $\alpha[\![\mathsf{t}]\!] \in \alpha[\![\mathsf{P}]\!]$. This is also sufficient since by the surjectivity (by construction) of α, we know that every element of the free model has a term representative.

Assume the contrary and let t be a term with the property that $\alpha[\![\mathsf{t}]\!] \in \alpha[\![\mathsf{s}]\!]$ but $\alpha[\![\mathsf{t}]\!] \notin \alpha[\![\mathsf{P}]\!]$. Suppose also that t is of minimum depth[1] among all such terms with this property. There are two cases. If $\sigma'_{\mathsf{i}}\mathcal{P}$ is a base case, then it is derivable and by CM Soundness the assumption is contradicted. Assume then the other case that t contains a variable of sort s. Since $\mathsf{t} = \sigma'_{\mathsf{i}}\mathsf{t}_{\mathsf{i}}$ for some $1 \leq \mathsf{i} \leq \mathsf{n}$ is a *minimal* element for which P does not hold, i.e., $\alpha[\![\sigma'_{\mathsf{i}}\mathsf{t}_{\mathsf{i}}]\!] \notin \alpha[\![\mathsf{P}]\!]$, there is a subterm $\sigma_{\mathsf{j}}\mathsf{X_s}$ of t for some $1 \leq \mathsf{j} \leq \mathsf{m}$ for which P holds. But by the desired conclusion part of the CM induction principle (and CM Soundness) then also P must hold for $\mathsf{t} = \sigma'_{\mathsf{i}}\mathsf{t}_{\mathsf{i}}$, a contradiction. Therefore the assumption that there is such a t must be false, hence, $\alpha[\![\mathsf{t}]\!] \in \alpha[\![\mathsf{P}]\!]$.

Theorem 10 (CM Induction Principle, Induction Case) *If the CM induction principle holds for formulas quantified over k variables, then the CM induction principle holds for formulas quantified over k+1 variables.*

Proof: Let $\mathcal{P}$ be $\mathsf{P}(\mathsf{X_s}, \mathsf{X}_{\mathsf{s}_1}, \ldots, \mathsf{X}_{\mathsf{s}_k})$ and suppose that the induction principle is applied to $(\mathsf{X_s}, \mathsf{X}_{\mathsf{s}_1}, \ldots, \mathsf{X}_{\mathsf{s}_k})\,\mathsf{P}(\mathsf{X_s}, \mathsf{X}_{\mathsf{s}_1}, \ldots, \mathsf{X}_{\mathsf{s}_k})$. Assume also that the induction

[1]Term depth is well founded because it is inductively defined for $\mathcal{T}_\Sigma(X)$ by: 1) for a constant symbol the depth is zero and 2) for a complex term the depth is one greater than the maximum depth of its subterms.

principle holds for all formulas quantified over k variables. It is necessary to show in the free model $\mathcal{T}_S(\mathcal{X})$ with interpretation function α that $\alpha[\![\mathsf{t}]\!] \in \alpha[\![\mathsf{s}]\!]$ implies that $\alpha[\![<\mathsf{t}, \mathsf{u}_1, \ldots, \mathsf{u}_k>]\!] \in \alpha[\![\mathsf{P}]\!]$.

Assume the contrary and let t be a term with the property that $\alpha[\![\mathsf{t}]\!] \in \alpha[\![\mathsf{s}]\!]$ but $\alpha[\![<\mathsf{t}, \mathsf{u}_1, \ldots, \mathsf{u}_k>]\!] \notin \alpha[\![\mathsf{P}]\!]$. Suppose also that t is of minimum depth among all such terms with this property. There are two cases. If $\sigma_i'\mathcal{P}$ is a base case, then it is derivable and by CM Soundness the assumption is contradicted. Assume then the other case that t contains a variable of sort s. Since $\mathsf{t} = \sigma_i'\mathsf{t}_i$ for some $1 \leq \mathsf{i} \leq \mathsf{n}$ is a *minimal* element for which $\alpha[\![<\sigma_i'\mathsf{t}_i, \mathsf{u}_1, \ldots, \mathsf{u}_k>]\!] \notin \alpha[\![\mathsf{P}]\!]$ (P does not hold), there is a subterm $\sigma_j\mathsf{X}_s$ of t for some $1 \leq \mathsf{j} \leq \mathsf{m}$ for which $\alpha[\![<\sigma_j\mathsf{X}_s, \mathsf{u}_1, \ldots, \mathsf{u}_k>]\!] \in \alpha[\![\mathsf{P}]\!]$ (P holds), assuming CM induction holds for formulas with k variables. But by the desired conclusion section of the CM induction principle then also P must hold for $\mathsf{t} = \sigma_i'\mathsf{t}_i$, i.e., $\alpha[\![<\sigma_i'\mathsf{t}_i, \mathsf{u}_1, \ldots, \mathsf{u}_k>]\!] \in \alpha[\![\mathsf{P}]\!]$, a contradiction. Therefore the assumption that there is such a t must be false, hence, $\alpha[\![\mathsf{t}]\!] \in \alpha[\![\mathsf{s}]\!]$ implies $\alpha[\![<\mathsf{t}, \mathsf{u}_1, \ldots, \mathsf{u}_k>]\!] \in \alpha[\![\mathsf{P}]\!]$.

Bibliography

[ACH76] E. A. Ashcroft, M. Clint, and C. A. R. Hoare. Remarks on "program proving: Jumps and functions by M. Clint and C. A. R. Hoare". *Acta Informatica*, 6:317–318, 1976.

[Apt81] Krzysztof R. Apt. Ten years of Hoare's logic: A survey – part 1. *ACM Trans. on Programming Languages and Systems*, 3(4):431–483, 1981.

[BEPP87] E.K. Blum, H. Ehrig, and F. Parisi-Presicce. Algebraic specification of modules and their basic interconnections. *JCSS*, 34:293–339, 1987.

[BG77] R. M. Burstall and J. A. Goguen. Putting theories together to make specifications. In *Proc. Fifth Int. Joint Conf. Artificial Intelligence*, volume 5, pages 1045–1058, 1977.

[BL70] G. Birkoff and D. Lipson. Heterogeneous algebras. *J. Combinatorial Theory*, 8:115–133, 1970.

[EK81] H. Ehrig and H.J. Kreowski. Example: Kwic-index generation. In *Lecture Notes in Computer Science, Vol 134*, pages 78–83. Springer-Verlag, 1981.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics.* Springer-Verlag, 1985.

[FGMO87] Kokichi Fatatsugi, Joseph Goguen, José Meseguer, and Koji Okada. Parameterized programming in OBJ2. In *9-th International Conference on Software Engineering*, pages 51–60. IEEE Computer Society, 1987.

[Flo67] R. W. Floyd. Assigning meanings to programs. *Proc. Symp. on Appl. Math.*, 19, 1967.

[FPB75] Jr. Frederick P. Brooks. *The Mythical Man-Month.* Addison Wesley, 1975.

[GB84] J. A. Goguen and R. M. Burstall. Introducing institutions. In *Lecture Notes in Computer Science, Vol 164*, pages 221–256. Springer-Verlag, 1984.

[GH78] J.V. Guttag and J.J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[GHW85] J.V. Guttag, J.J. Horning, and J.M. Wing. Larch in five easy pieces. Technical report, DEC Systems Research Center, 1985.

[GJM85] Joseph A. Goguen, J. Jouannaud, and José Meseguer. Operational semantics of order-sorted algebra. In *Lecture Notes in Computer Science, Vol 194*. Springer-Verlag, 1985.

[GM82] J. Goguen and J. Meseguer. Universal realization, persistent interconnection and implementation of abstract modules. In *9-th International Colloquim on Automata, Languages and Programming (LNCS 140)*, pages 265–281. Springer-Verlag, 1982.

[GM87a] J. Goguen and J. Meseguer. Order sorted algebra solves the constructor-selector, multiple representation and coercion problems. In *LNCS*. Springer Verlag, 1987.

[GM87b] Joseph A. Goguen and José Meseguer. Models and equality for logical programming. In *TAPSOFT 87*, pages 1–22. Springer-Verlag, 1987.

[GM87c] Joseph A. Goguen and José Meseguer. *Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics*, pages 417–477. MIT Press, 1987.

[GM89] Joseph A. Goguen and José Meseguer. Order-sorted algebra i: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. Technical Report SRI-CSL-89-10, SRI International, 1989.

[Gog81] J. A. Goguen. Ordinary specification of kwic-index generation. In *Lecture Notes in Computer Science, Vol 134*, pages 114–117. Springer-Verlag, 1981.

[Gog86] J. A. Goguen. Reusing and interconnecting software components. *Computer*, February 1986.

[Gog90] Joseph A. Goguen. An algebraic approach to refinement. In *VDM 1990*, pages 1–14. Springer-Verlag, 1990.

[GTW78] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. *An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types*, pages 80–149. Prentice-Hall, 1978.

[Hen80] K.L. Henniger. Specifying software requirements for complex systems: New techniques and their applications. *IEEE Trans. Software Engineering*, SE-6:2–13, 1980.

[HO80] Gerard Huet and Dereck C. Oppen. Equations and rewrite rules: A survey. Technical Report STAN-CS-80-785, Stanford University, 1980.

[KL83] B. Kutzler and F. Lichtenberger. *Bibliography on Abstract Data Types.* Springer-Verlag, 1983.

[Maj77] Mila E. Majster. Limits of the "algebraic" specification of abstract data types. *SIGPLAN Notices*, pages 37–42, October 1977.

[Mak87] J. A. Makowsky. Why Horn formulas matter in computer science: Initial structures and generic examples. *J. of Computer and System Science*, 34:266–292, 1987.

[MG85] José Meseguer and Joseph A. Goguen. Initiality, induction and computability. In *Algebraic Methods in Semantics*, pages 459–551. Cambridge University Press, 1985.

[MW81] Zohar Manna and Richard Waldinger. Problematic features of programming languages: A situational-calculus approach. *Informatica*, 16, 1981.

[MW86] Zohar Manna and Richard Waldinger. Special relations in automated deduction. *J.A.C.M*, 33, 1986.

[MW87a] Zohar Manna and Richard Waldinger. The deductive synthesis of imperative lisp programs. In *AAAI-87*, 1987.

[MW87b] Zohar Manna and Richard Waldinger. How to clear a block: A theory of plans. *J. of Automated Reasoning*, 3:343–377, 1987.

[MW87c] Zohar Manna and Richard Waldinger. The origin of a binary-search paradigm. *Science of Computer Programming*, 9:37–83, 1987.

[MW89] Zohar Manna and Richard Waldinger. *The Logical Basis for Computer Programming, vol.2: Deductive Techniques.* Addison-Wesley, 1989.

[Par71] David L. Parnas. Information distribution aspects of design methodology. *Proc IFIP Congress*, 1971.

[Par72a] David L. Parnas. On the criteria to be used in decomposing systems into modules. *C.A.C.M.*, December 1972.

[Par72b] David L. Parnas. A technique for software module specification with examples. *C.A.C.M.*, May 1972.

[Par84] David L. Parnas. Software engineering principles. *INFOR Canadian Journal of Operations Research and Information Processing*, 44(4), November 1984.

[PB78] D.L. Parnas and W. Bartussek. Using traces to write abstract specifications for software. In *Lecture Notes in Computer Science 65*. Springer-Verlag, 1978.

[Wad82] W.W. Wadge. Classified algebras. Technical report, University of Warwick, 1982.

[Wad90] W.W. Wadge. Higher order Horn logic programming. Technical report, University of Victoria, 1990.

[Wan79] Michell Wand. Final algebra semantics and data type extensions. *JCSS*, 19:27–44, 1979.

[Wik87] Ake Wikstrom. *Functional Progamming Using Standard ML.* Prentice Hall, 1987.