Model-theoretic foundations for program development: basic concepts and motivation*

Donald Sannella
Department of Computer Science
University of Edinburgh
Edinburgh, UK.
e-mail: dts@dcs.ed.ac.uk

Andrzej Tarlecki
Institute of Informatics
Warsaw University
and
Institute of Computer Science
Polish Academy of Sciences
Warsaw, Poland.
e-mail: tarlecki@mimuw.edu.pl

7th March 1995

Abstract: This paper presents in an informal way the main ideas underlying work on the model-theoretic foundations of algebraic specification and formal program development. An attempt is made to offer an overall view, rather than new results, and to focus on the basic motivation behind the technicalities presented elsewhere.

Introduction

The long-term goal of work on algebraic specification is to provide a formal basis to support the systematic development of correct programs from specifications by means of verified refinement steps. There has been a large body of technical work directed towards this important goal. Many interesting concepts have been introduced and quite a number of non-trivial results have been stated and proved (see [BKLOS 91] for a review and a comprehensive list of references). Instead of providing yet another piece in the puzzle, in this paper we sketch on a rather informal level our views on how some of the existing pieces fit into an overall picture of what is important in the light of the ultimate goal. We focus on the motivations for certain technicalities that we think are of crucial importance, only suggesting, rather than presenting in full detail, the technicalities themselves. A past paper with similar aims is [GHW 82].

The literature already mentions many of the points we make here, such as the use of "institutions" to ensure sufficient generality of the proposed framework, and the use of "constructor implementations" to capture the essence of program development steps (including "design" steps that involve a decomposition into independent programming tasks). Some of these ideas have been hidden amongst the technical definitions and results, and we think they are worth restating here more prominently, with more careful arguments in some cases.

*This is an essentially revised and expanded version of [ST 92].

For example, we provide a more detailed justification for the use of model classes, rather than theories, as the appropriate semantic domain for specifications. We also give a simple explanation of the somewhat subtle interplay between behavioural equivalence, "stability" and refinement in formal development.

Examples are provided to illustrate some of the points we make. Chosen for simplicity rather than to impress, they are kept as small as possible, and some are contrived just to illustrate a particular point. This should not be taken as an indication that large examples cannot be handled, of course. For the sake of concreteness, and to show how the ideas fit into the context of existing programming languages, in examples we use notation and concepts borrowed from the Standard ML (SML) programming language [Pau 91].

Since the emphasis here is on motivation and intuition, citations to the literature refer the interested reader to papers where complete technical details may be found. These topics are also covered in detail in a forthcoming monograph [ST 95].

1 The logical framework

The overall aim of work on algebraic specification is to provide semantic foundations for the development of programs that are *correct* with respect to their requirements specifications. In other words, the program developed must exhibit the required input/output behaviour. We view the correctness of a program as its most crucial property. Other desirable properties (efficiency, robustness, reliability etc.) are disregarded in this work. Of course, this does not mean that they are unimportant, but this approach does not provide any formal means for their analysis.

The assumption that the correctness of the input/output behaviour of a program takes precedence over all its other properties allows us to abstract away from concrete details of code and algorithms, and to model program functions as mathematical functions. Such functions are never considered in isolation, but always in units (program modules) comprising a collection of related functions together with the data domains they operate on. At this level of abstraction we are dealing directly with the information essential for the analysis of program correctness, without the burden of irrelevant details. This leads to the most fundamental assumption underlying work on algebraic specification: programs are modelled as many-sorted *algebras*. This assumption fits most directly into the *functional programming* paradigm, but there is a natural way of generalizing it to handle e.g. imperative programs; see below.

We refrain from recalling the formal definition of many-sorted algebra (see e.g. [EM 85]). It is enough to know that an algebra consists of a collection of *carriers* (sets of data) and *operations* on them. Algebras are classified by *signatures*, naming the algebra components (sorts and operations) and thus providing the basic vocabulary for using the program and for making assertions about its properties. The class of all Σ-algebras (algebras over the signature Σ) will be denoted by $Alg(\Sigma)$. For any program P, the algebra it denotes is written as $[\![P]\!] \in Alg(Sig(P))$, where $Sig(P)$ is the underlying signature of P.

For any signature, we need a logical system for describing properties of algebras over that signature. Many-sorted equational logic (cf. [GM 85, EM 85]) is the most commonly-used system for this purpose, at least in the area of algebraic specification. Properties of Σ-algebras (or rather, of their operations) may be described by universally-quantified *equations* over Σ, via the definition of what it means for a Σ-algebra A to *satisfy* a Σ-equation φ, written $A \models \varphi$. This also determines a notion of *logical consequence*: a set of axioms Φ entails an equation φ, written $\Phi \models \varphi$, if every algebra that satisfies all the axioms in Φ also satisfies φ. Here is a simple example, where a signature is accompanied by a list of axioms, presented using a hopefully self-explanatory notation:

$$
\begin{array}{ll}
\textbf{sorts} & nat, list \\
\textbf{opns} & 0 : nat \\
& succ : nat \to nat \\
& nil : list \\
& cons : nat \times list \to list \\
& head : list \to nat \\
& tail : list \to list \\
\textbf{axioms} & \forall x{:}nat.\, \forall l{:}list.\, head(cons(x,l)) = x \\
& \forall x{:}nat.\, \forall l{:}list.\, tail(cons(x,l)) = l
\end{array}
$$

For example now:

$$\left\{\begin{array}{l} \forall x{:}nat.\, \forall l{:}list.\, head(cons(x,l)) = x, \\ \forall x{:}nat.\, \forall l{:}list.\, tail(cons(x,l)) = l \end{array}\right\} \models \forall x, y{:}nat.\, head(tail(cons(x, cons(y, nil)))) = y$$

Very rarely in the process of program development does the user work with just a single signature: operations and sorts of data are renamed, added and hidden according to what seems most suitable. To take account of this, signatures are equipped with a notion of *signature morphism* (cf. [EM 85]). A signature morphism $\sigma : \Sigma \to \Sigma'$ maps the sort and operation names of Σ to those of Σ'. This determines in a natural way a translation of any Σ-equation φ to a Σ'-equation $\sigma(\varphi)$, and on the semantic level, a translation of any Σ'-algebra $A' \in Alg(\Sigma')$ to its *reduct* $A'|_\sigma \in Alg(\Sigma)$ — notice the change of direction! (When $\sigma : \Sigma \to \Sigma'$ is a signature inclusion, we will write $A'|_\Sigma$ for $A'|_\sigma$.) A crucial property is that these two translations are compatible with satisfaction: for any Σ-equation φ and Σ'-algebra A', $A'|_\sigma \models \varphi$ iff $A' \models \sigma(\varphi)$.

The above framework is often criticised (quite rightly!) as rather restrictive and cumbersome to use in practice. Some important features of programs, for example non-termination and higher-order functions, are difficult to model in algebras; equations are often not expressive enough to conveniently capture properties that one may want to state as requirements. For instance, one may wish to add to the list of axioms above the following two properties which cannot be expressed in equational logic and which are stated here as axioms in first-order logic with equality (using the standard notation for negations of equalities):

$$
\begin{array}{ll}
\textbf{axioms} & \forall n{:}nat.\, succ(n) \neq 0 \\
& \forall x{:}nat.\, \forall l{:}list.\, cons(x,l) \neq nil
\end{array}
$$

Fortunately, this deficiency is relatively easy to overcome using the concept of *institution*. This concept was introduced by Goguen and Burstall [GB 84] to capture the informal notion of a logical system and was strongly influenced by the understanding of this notion in the theory of specifications (see [Bar 74] for an early account of *abstract model theory* covering similar ideas approached from the viewpoint of classical logic and model theory, and [BF 85] for a compendium of more recent work in this area).

An institution defines a notion of signature together with for any signature Σ, a set of Σ-sentences, a class of Σ-models and a satisfaction relation between Σ-models and Σ-sentences. Moreover, signatures come equipped with a notion of signature morphism. Any signature morphism induces a translation of sentences and a translation of models (the latter going in the opposite direction as above). The only semantic requirement is that when we change signatures using a signature morphism, the induced translations of sentences and of models preserve the satisfaction relation. Many standard logical systems have been presented explicitly as institutions, see e.g. [GB 92] and [ST 95]. These include first-order predicate logic with and without equality, and logical systems for specifying partial functions, exception handling, and simple imperative programs. (Most of the examples in this paper are couched in first-order predicate logic with equality.) It should be easy to see that any usual logical system with a well-defined model theory fits into this mould.

Everything below, barring concrete examples, works in the framework of an arbitrary institution, even though for the reader's convenience we avoid "institutional jargon" and use more familiar terms (signature, algebra, axiom). Consequently, everything in this paper applies to many different concepts of "signature", "algebra" and "axiom" used in the theory and practice of software specification. This point of view gives rise to "reusable" methodologies, theorems, and (ultimately) tools, all of which can be seen as parameterised by an arbitrary institution. See e.g. [BV 87, ST 87, ST 88a, ST 88b, SST 92, DGS 93] for work on various aspects of software specification and development that is generic in this sense. Other formulations of general logic have been used for similar purposes, see e.g. [FS 88] and [EBO 93].

Strict followers of the early approaches to algebraic specification might view this generalization as an alarming departure, and might protest that this is not algebraic specification at all. In our view the essential idea of algebraic specification is the stress on "algebra-like" models and the use of logical axioms to describe such models. The use of ordinary many-sorted algebras and equations is but a special case of this. Just as it was necessary to generalise from classical single-sorted algebras to many-sorted algebras in order to deal with programs handling several kinds of data, it is necessary to adopt more complicated models to deal with other features of programming languages (polymorphism, higher-order functions, infinite behaviour, updateable references, lazy evaluation, etc.). The essence is that we need a notion of a semantic structure that is detailed enough to capture the program properties we want to analyse and abstract enough to make this analysis feasible. Moreover, to specify and reason about programs, we need a logical system with a model theory based on such structures; again, there is a tradeoff here between expressive power and ease of use. There seems to be no single kind of semantic structure that suffices for all purposes, and different logical systems are appropriate for the analysis of different facets of program behaviour. The multiplicity of logical frameworks seems to be a natural state of affairs rather than indicative of a failure to find the right approach.

2 Specifications

What is a specification? Clearly, since our aim is a formal approach to software development, specifications must be objects as formal as (for example) programs are. That is, we have to have a formal language to write specifications down and to provide a vehicle on which formal techniques to manipulate specifications may be based. It is important for such a *specification language* to provide a collection of convenient notational conventions that are easy to understand and use. One of the basic constituents of a specification will be a list of axioms the specified program is required to satisfy.

A specification formalism must offer means for building complex *structured* specifications by combining and extending simpler ones. A specification of a real-life system typically states a huge number of properties, and building such a specification in an unstructured, monolithic way would result in a long list of axioms which would be neither understandable nor useful. Moreover, the structure of a specification may be used to express intangible aspects of the specifier's knowledge of the problem, such as the degree to which the entities and concepts described in the specification are interrelated. For this purpose, a specification language must provide some *specification-building operations* used to put together small specifications to form more complex ones [BG 77, BG 80]. Then, an understanding of a large specification is achieved via an understanding of its components. This is the principle of *compositionality*: the meaning of a composite object depends only on the meanings of its immediate sub-components.

Various other activities involving specifications can exploit their structure. For example, proofs of consequences of a specification can be usefully guided by its structure [SB 83], [HST 94] (cf. Section 5). But this principle must not be taken too far: for example, the structure of a specification should not constrain the final structure of its implementations. This is one of the consequences of the famous dogma that a specification should describe only the *what's* of the specified software without constraining any of its *how's*. Requiring the structure of the initial specification to be preserved in its implementation would be unrealistic and unreasonable, even though this has been explicitly suggested by some (e.g. [GB 80, MA 91]) and is implicit in the approaches taken by others. The aims of structuring requirements specifications are often contradictory with the aims of structuring software. See for instance [FJ 90] for a nice discussion of a practical example where such a discrepancy occurs. Section 6 gives a simple example illustrating this point, and Section 7 indicates how the design of the structure of an implementation may be brought into our framework.

Choosing appropriate specification-building operations to be included in a specification language is a non-trivial task, even though most specification languages share certain common operations such as those given below. The choice involves a certain trade-off between the expressive power of the specification language and the ease of understanding and dealing with the operations. One way to circumvent this problem is to first develop a *kernel* language consisting of a minimal set of very powerful, but perhaps difficult to use operations, and then build on top of it a higher-level, more user-friendly language, perhaps sacrificing some of the expressive power to achieve ease of use and ease of understanding. Such an approach has been taken with the ASL kernel specification language [SW 83, ST 88a, Wir 86], on top of which languages such as PLUSS [Gau 84] and Extended ML [ST 86] have been built.

In this paper we will neither present nor use a full-blown specification language. In examples we will rely on the following three simple specification-building operations:

Basic specifications: The specification

$$\begin{array}{ll} \textbf{sorts} & S \\ \textbf{opns} & \Omega \\ \textbf{axioms} & \Phi \end{array}$$

describes algebras over the signature with sorts S and operations Ω that satisfy the axioms Φ.

Enrichment: The specification

$$\textbf{enrich } SP \textbf{ by } \begin{array}{ll} \textbf{sorts} & S \\ \textbf{opns} & \Omega \\ \textbf{axioms} & \Phi \end{array}$$

describes algebras that add the sorts S and operations Ω to algebras described by SP in such a way that the axioms Φ are satisfied.

Hiding: The specification

$$\begin{array}{lll} \textbf{hide} & \textbf{sorts} & S \\ & \textbf{opns} & \Omega \\ \textbf{in}\ SP & & \end{array}$$

describes those algebras obtained by removing the sorts S and operations Ω from algebras described by SP.

In examples we will omit keywords like **sorts** when the corresponding list of items is empty.

Example 1

Here are some simple specifications:

$$\begin{array}{lll} BOOL = & \textbf{sorts} & bool \\ & \textbf{opns} & true : bool \\ & & false : bool \\ & \textbf{axioms} & true \neq false \\ & & \forall x{:}bool.\, x = true \vee x = false \end{array}$$

$$\begin{array}{lll} INT = & \multicolumn{2}{l}{\textbf{enrich}\ BOOL\ \textbf{by}} \\ & \textbf{sorts} & int \\ & \textbf{opns} & 0 : int \\ & & succ : int \rightarrow int \\ & & pred : int \rightarrow int \\ & & le : int \times int \rightarrow bool \\ & \textbf{axioms} & \ldots \text{induction scheme for } int \ldots \\ & & \forall x{:}int.\, pred(x) \neq x \wedge succ(x) \neq x \\ & & \forall x, y{:}int.\, pred(x) = pred(y) \Longrightarrow x = y \\ & & \forall x, y{:}int.\, succ(x) = succ(y) \Longrightarrow x = y \\ & & \forall x{:}int.\, le(x, x) = true \\ & & \forall x, y{:}int.\, le(x, y) = true \wedge le(y, x) = true \Longrightarrow x = y \\ & & \forall x, y, z{:}int.\, le(x, y) = true \wedge le(y, z) = true \Longrightarrow le(x, z) = true \end{array}$$

$$
\begin{array}{lll}
\mathit{INTLIST} = & \textbf{enrich } \mathit{INT} \textbf{ by} & \\
& \textbf{sorts} & \mathit{list} \\
& \textbf{opns} & \mathit{nil} : \mathit{list} \\
& & \mathit{cons} : \mathit{int} \times \mathit{list} \to \mathit{list} \\
& & \mathit{head} : \mathit{list} \to \mathit{int} \\
& & \mathit{tail} : \mathit{list} \to \mathit{list} \\
& & \mathit{append} : \mathit{list} \times \mathit{list} \to \mathit{list} \\
& & \mathit{is\_in} : \mathit{int} \times \mathit{list} \to \mathit{bool} \\
& \textbf{axioms} & \ldots \text{induction scheme for } \mathit{list} \ldots \\
& & \forall x{:}\mathit{int}.\, \forall l{:}\mathit{list}.\, \mathit{cons}(x,l) \neq l \\
& & \forall x, x'{:}\mathit{int}.\, \forall l, l'{:}\mathit{list}. \\
& & \qquad \mathit{cons}(x,l) = \mathit{cons}(x',l') \Longrightarrow x = x' \wedge l = l' \\
& & \forall x{:}\mathit{int}.\, \forall l{:}\mathit{list}.\, \mathit{head}(\mathit{cons}(x,l)) = x \\
& & \forall x{:}\mathit{int}.\, \forall l{:}\mathit{list}.\, \mathit{tail}(\mathit{cons}(x,l)) = l \\
& & \forall l{:}\mathit{list}.\, \mathit{append}(\mathit{nil},l) = l \\
& & \forall x{:}\mathit{int}.\, \forall l, l'{:}\mathit{list}.\, \mathit{append}(\mathit{cons}(x,l),l') = \mathit{cons}(x,\mathit{append}(l,l')) \\
& & \forall x{:}\mathit{int}.\, x \in \mathit{nil} = \mathit{false} \\
& & \forall x, y{:}\mathit{int}.\, \forall l{:}\mathit{list}.\, x \neq y \Longrightarrow x \in \mathit{cons}(y,l) = x \in l \\
& & \forall x{:}\mathit{int}.\, \forall l{:}\mathit{list}.\, x \in \mathit{cons}(x,l) = \mathit{true}
\end{array}
$$

We're glossing over the details of induction schemes here — think of each of these as either an infinite set of first-order axioms given by the usual elementary induction scheme, or (not equivalently!) as a single second-order axiom. An alternative which is more usual in specification languages is to introduce a separate specification-building operation that restricts the class of admissible realizations of a specification to reachable algebras only, see e.g. [SW 83], [Wir 86] (this is equivalent to the additional second-order axiom). Yet another potential possibility is to restrict the class of algebras considered to reachable algebras from the very beginning [BW 82].

$$
\begin{array}{lll}
\mathit{SORT1} = & \textbf{enrich } \mathit{INTLIST} \textbf{ by} & \\
& \textbf{opns} & \mathit{is\_sorted} : \mathit{list} \to \mathit{bool} \\
& & \mathit{sort} : \mathit{list} \to \mathit{list} \\
& \textbf{axioms} & \mathit{is\_sorted}(\mathit{nil}) = \mathit{true} \\
& & \forall x{:}\mathit{int}.\, \forall l{:}\mathit{list}.\, \mathit{is\_sorted}(\mathit{cons}(x,l)) = \mathit{true} \iff \\
& & \qquad\qquad ((\forall y{:}\mathit{int}.\, \mathit{is\_in}(y,l) = \mathit{true} \Longrightarrow \mathit{le}(x,y) = \mathit{true}) \\
& & \qquad\qquad \wedge\, \mathit{is\_sorted}(l) = \mathit{true}) \\
& & \forall l{:}\mathit{list}.\, \mathit{is\_sorted}(\mathit{sort}(l)) = \mathit{true} \\
& & \forall l{:}\mathit{list}.\, \forall x{:}\mathit{int}.\, \mathit{is\_in}(x,l) = \mathit{is\_in}(x,\mathit{sort}(l))
\end{array}
$$

$$\mathit{SORT} = \ \textbf{hide opns } \mathit{is\_sorted} \textbf{ in } \mathit{SORT1}$$

The use of the hiding operation in *SORT* means that the *is_sorted* operation doesn't appear in algebras described by *SORT*. It appears in the specification as an auxiliary operation which allows us to formulate the axioms for *sort* conveniently.

The above does not take into account that typical programming languages like SML provide booleans, integers and lists as built-in types. Rather than re-specifying and then re-implementing them from scratch, we could follow Extended ML [KST 94a, KST 94b] by assuming that all programs implicitly extend all the built-in types and values available, so programs and specifications may freely refer to them. Modifying the above specifications along these lines, assuming that the built-in types and values of SML are available, yields the following:

$$
\begin{array}{lll}
INT = & \textbf{opns} & le : \mathtt{int} \times \mathtt{int} \to \mathtt{bool} \\
 & \textbf{axioms} & \forall x{:}\mathtt{int}.\, le(x,x) = \mathtt{true} \\
 & & \forall x,y{:}\mathtt{int}.\, le(x,y) = \mathtt{true} \wedge le(y,x) = \mathtt{true} \Longrightarrow x = y \\
 & & \forall x,y,z{:}\mathtt{int}.\, le(x,y) = \mathtt{true} \wedge le(y,z) = \mathtt{true} \Longrightarrow le(x,z) = \mathtt{true}
\end{array}
$$

$$
\begin{array}{lll}
INTLIST = & \multicolumn{2}{l}{\textbf{enrich } INT \textbf{ by}} \\
 & \textbf{opns} & head : \mathtt{int\ list} \to \mathtt{int} \\
 & & tail : \mathtt{int\ list} \to \mathtt{int\ list} \\
 & & is\_in : \mathtt{int} \times \mathtt{int\ list} \to \mathtt{bool} \\
 & \textbf{axioms} & \forall x{:}\mathtt{int}.\, \forall l{:}\mathtt{int\ list}.\, head(x::l) = x \\
 & & \forall x{:}\mathtt{int}.\, \forall l{:}\mathtt{int\ list}.\, tail(x::l) = l \\
 & & \forall x{:}\mathtt{int}.\, is\_in(x,\mathtt{nil}) = \mathit{false} \\
 & & \forall x,y{:}\mathtt{int}.\, \forall l{:}\mathtt{int\ list}.\, x \neq y \Longrightarrow is\_in(x,y::l) = is\_in(x,l) \\
 & & \forall x{:}\mathtt{int}.\, \forall l{:}\mathtt{int\ list}.\, is\_in(x,x::l) = \mathtt{true}
\end{array}
$$

$$
\begin{array}{lll}
SORT1 = & \multicolumn{2}{l}{\textbf{enrich } INTLIST \textbf{ by}} \\
 & \textbf{opns} & is\_sorted : \mathtt{int\ list} \to \mathtt{bool} \\
 & & sort : \mathtt{int\ list} \to \mathtt{int\ list} \\
 & \textbf{axioms} & is\_sorted(\mathtt{nil}) = \mathtt{true} \\
 & & \forall x{:}\mathtt{int}.\, \forall l{:}\mathtt{int\ list}. \\
 & & \quad is\_sorted(x::l) = \mathtt{true} \iff \\
 & & \quad ((\forall y{:}\mathtt{int}.\, is\_in(y,l) = \mathtt{true} \Longrightarrow le(x,y) = \mathtt{true}) \\
 & & \quad \ \wedge\ is\_sorted(l) = \mathtt{true}) \\
 & & \forall l{:}\mathtt{int\ list}.\, is\_sorted(sort(l)) = \mathtt{true} \\
 & & \forall l{:}\mathtt{int\ list}.\, \forall x{:}\mathtt{int}.\, is\_in(x,l) = is\_in(x,sort(l))
\end{array}
$$

$$SORT = \ \textbf{hide opns } is\_sorted \textbf{ in } SORT1$$

We will work with this version of these specifications throughout the rest of the paper.

In examples throughout the rest of the paper, as in the above specifications, all signatures are taken to implicitly include all of the built-in type and value names of SML and all algebras extend the interpretation of those names give by the SML semantics. □

3 Semantics of specifications

Any specification language must be given a precise, formal semantics. The very concept of "correct" program is meaningless in the absence of a definition of what it is supposed to compute, and a specification can only provide such a definition if it has an unambiguously-defined meaning.

Before we start assigning meanings to specifications, it is necessary to decide what kind of mathematical objects to use to represent the meanings of specifications, i.e. to decide what specifications *denote*. Whatever the full answer is, a specification at least determines the underlying signature of the specified program. For any specification SP, we write this signature as $Sig(SP)$. Then, one may attempt to give a semantics of specifications on (at least) three different levels:

- Presentation level: a specification SP denotes a set of axioms over $Sig(SP)$ (this set may be required to be finite or at least recursive or recursively enumerable). At this level, the meaning of a specification is close to the syntactic form in which specifications are written; the semantics extracts the axioms, resolves references to other specifications, etc.

- Theory level: a specification SP denotes a *theory* over $Sig(SP)$, that is, a set of $Sig(SP)$-axioms that is closed under logical consequence. This theory is much larger[1] than the set of axioms that are explicitly given in the specification. It is always infinite[1], usually not recursive and sometimes not recursively enumerable; thus the meaning of a specification is no longer strictly syntactic. The semantics performs the closure under logical consequence.

- Model-class level: a specification SP denotes a class of $Sig(SP)$-algebras. At this level, the meaning of a specification is entirely non-syntactic, except for the signature part. The semantics abstracts away from the axioms, taking into account only their possible realizations.

The ultimate role of any specification is to describe a class of programs which we want to view as its correct realizations. Since we have already decided to model programs as algebras, *specifications ultimately determine classes of algebras.* Given the natural mappings from presentations to theories and from theories to model classes, this holds whichever one of these three levels we choose for the semantic domain.

For any specification SP, the semantics of SP determines the class of all *models* of SP, denoted by $[\![SP]\!] \subseteq Alg(Sig(SP))$. This class contains algebras that model programs which are considered to be correct realizations of SP. (There is a subtle issue involved in ensuring that *all* such algebras are admitted as models; see Section 9.) This semantics determines a notion of logical consequence of a specification: a specification SP entails an axiom φ, written $SP \models \varphi$, if φ holds in every model of SP.

Of course, a specification SP may admit a number of different program behaviours, and hence we cover so-called *loose* specifications. Or it might admit no models at all, in which case it is called *inconsistent.*

The semantics of a specification formalism is usually presented by giving a number of semantic clauses, one for each specification-building operation. Each clause defines the meaning of a specification built using the given operation in terms of the meanings of its component specifications. This style of presentation gives a compositional semantics.

The following defines the models of specifications formed using the three specification-building operations that were informally presented earlier; cf. e.g. [ST 88a].

Basic specifications:

$$[\![\textbf{sorts } S \textbf{ opns } \Omega \textbf{ axioms } \Phi]\!] = \{A \in Alg(\Sigma) \mid A \models \Phi\}$$

where $\Sigma = Sig(\textbf{sorts } S \textbf{ opns } \Omega \textbf{ axioms } \Phi)$ is the signature having sorts S and operations Ω.

Enrichment:

$$[\![\textbf{enrich } SP \textbf{ by sorts } S \textbf{ opns } \Omega \textbf{ axioms } \Phi]\!] = \{A \in Alg(\Sigma) \mid A|_{Sig(SP)} \in [\![SP]\!] \text{ and } A \models \Phi\}$$

[1]Of course, this depends on the logic involved, but for example in equational logic every theory contains all the trivially true axioms like $\forall x{:}s.\, x = x$.

where $\Sigma = Sig(\textbf{enrich } SP \textbf{ by sorts } S \textbf{ opns } \Omega \textbf{ axioms } \Phi)$ is the signature $Sig(SP)$ with additional sorts S and operations Ω.

Hiding:

$$[\![\textbf{hide sorts } S \textbf{ opns } \Omega \textbf{ in } SP]\!] = \{A|_{\Sigma} \mid A \in [\![SP]\!]\}$$

where $\Sigma = Sig(\textbf{hide sorts } S \textbf{ opns } \Omega \textbf{ in } SP)$ is the signature $Sig(SP)$ with sorts S and operations Ω removed.

Thus, specification-building operations semantically correspond simply to functions on classes of algebras.

Example 1 (continued)
The semantics of the specification-building operations can be used to calculate the meanings of specifications like those in Example 1 (Section 2), with the proviso given there concerning the built-in types and values of SML. For example, $Sig(INTLIST)$ extends the built-in type and value names of SML by le, $head$, $tail$ and is_in, and $[\![INTLIST]\!]$ is the class of algebras over this signature that extend the interpretation of the built-in names given by SML with operations le, $head$, $tail$ and is_in defined in such a way that the axioms of $INTLIST$ are satisfied. Then, $Sig(SORT1)$ extends $Sig(INTLIST)$ by is_sorted and $sort$, and $[\![SORT1]\!]$ is the class of algebras over this signature that enrich the algebras in $[\![INTLIST]\!]$ so that the axioms of $SORT1$ are satisfied. Finally, $Sig(SORT)$ extends $Sig(INTLIST)$ by $sort$ only, and $[\![SORT]\!]$ is the class of algebras over this signature that result from the algebras in $[\![SORT1]\!]$ by removing the interpretation of is_sorted. □

For any signature Σ, there is a Galois connection between classes of Σ-algebras and sets of Σ-axioms, assigning to any set of axioms the class of all algebras that satisfy them, and to any class of algebras the set of all axioms that hold in them (see [GB 92]). The closed elements of this Galois connection are theories; these are in one-to-one correspondence with closed (i.e., definable by sets of axioms) classes of algebras. It follows from this that the theory level is less expressive as a semantic domain for specifications than either the presentation or the model-class level. The latter two are, however, incomparable: there are properties that can be naturally studied at the presentation level (for example, finiteness of an axiomatisation) with no natural counterpart at the model-class level, and vice versa.

It is not immediately obvious that working at the model-class level brings any essential benefits over working with closed classes of algebras only, or equivalently, working at the theory level. It is not clear whether non-closed classes of algebras ever arise as meanings of specifications; even if they do arise, it is not clear whether this makes any difference for the use of specifications. The following example, built in the institution of equational logic, exhibits both of these phenomena:

$$SP \left\{ \begin{array}{l} \textbf{enrich} \\ \quad SP_1 \left\{ \begin{array}{l} \textbf{hide opns } a \textbf{ in} \\ \quad SP_0 \left\{ \begin{array}{ll} \textbf{sorts} & s, s' \\ \textbf{opns} & a : s \\ & b, c : s' \end{array} \right. \end{array} \right. \\ \textbf{by axioms } \forall x{:}s.\, b = c \end{array} \right.$$

This example relies on the following well-known fact [GM 85]: $\forall x{:}s.\, b = c$ does not imply $b = c$, although it implies $b = c$ for $Sig(SP)$-algebras with non-empty carrier of sort s.

Now, according to the above definitions, $[\![SP_0]\!]$ is the class of all algebras (over the indicated signature) and $[\![SP_1]\!]$ consists of all algebras that are reducts of $Sig(SP_0)$-algebras, obtained by removing the constant a. Consequently, $[\![SP_1]\!]$ contains only those algebras having a non-empty carrier of sort s. Then, selecting from $[\![SP_1]\!]$ the algebras that satisfy $\forall x{:}s.\, b = c$ yields the class $[\![SP]\!]$ — and all these algebras satisfy $b = c$ (since for the algebras in $[\![SP_1]\!]$, $b = c$ follows from $\forall x{:}s.\, b = c$). Thus, under the model-class interpretation, the property $b = c$ is a consequence of the specification SP.

On the other hand, at the theory level, SP_0 would clearly have to denote the trivial equational theory, and so would SP_1 (there are no equations capable of expressing the fact that a carrier is non-empty). Then, the additional axiom $\forall x{:}s.\, b = c$ in the context of the theory denoted by SP_1 does not entail the equation $b = c$. Thus, under the theory-level interpretation, $b = c$ is *not* a consequence of the specification SP.

This discrepancy (and similar examples one may construct without relying on the "empty carriers" phenomenon) faces us with the necessity to choose between theories and classes of algebras as the basic semantic domain for specifications. The choice is obvious: the objects of ultimate interest here are programs, which are modelled as algebras, while axioms and theories are nothing more than logical means for describing them. In our view, the lack of agreement between theories and classes of algebras clearly demonstrates that theories are *not* in general adequate as denotations of specifications. But see [DGS 93] for a different point of view.

The alert reader may have noticed that the above example depends crucially on the use of equational theories. If we reinterpret the example in the institution of first-order predicate logic with equality, then the class $[\![SP_1]\!]$ becomes definable (by the axiom $\exists x{:}s.\, true$) and the discrepancy between the theory level and the model class level semantics of SP disappears. This is an instance of a general phenomenon: as the expressive power of the logical system in use increases, the gap between the theory level and the model class level semantics narrows. For example, in the institution of second order logic [HS 95], any class of models of a specification built using the specification-building operations presented above is definable by a set of axioms. The presence or absence of such a gap also depends on the expressive power of the specification-building operations in use. For example, if we use only basic specifications and **enrich** (leaving out hiding, as in Larch [GH 93] and ACT ONE [EM 85]), then there is no gap, no matter what institution we use.

4 Specification engineering

The point of constructing a specification is so that it may be used to define a programming task by precisely delimiting the range of program behaviours that are to be regarded as permissible. This statement of the programming task represents a rather idealized view that the requirements specification we start with accurately reflects the real-life requirements the customer expects to be satisfied. Thus formulated, the programming task allows for no change to the original specification. Of course, this is not very realistic, and we envisage that a programmer working with an appropriate support system will be able to try slightly different ideas, to modify the original formal specification, and to experiment with a whole collection (or better, a tree) of specifications.

The problem of getting the original requirements specification right is the topic of "requirements engineering" [Dav 90]. Even though there has been much work on this, we are not convinced that all the potential of the use of formal specifications has been explored.

Naturally, there always will be a gap between the informal wishes of the customer and their formalised version couched as a precise, unambiguous, entirely formal specification. Nevertheless, some methods involved in the process of passing from the former to the latter may and should find a more precise formulation. Some steps in this direction are taken in [RAHL 94] and [Li 94].

One major problem within this area is how we can make formal specifications easier to understand and construct. We have already mentioned part of the answer in the previous section: we need good specification languages with good structuring operations allowing specifications to be built and understood in a systematic, modular fashion.

Another aspect here is that once a specification is built, the customer should be able to "play" with it, to test whether or not the specification indeed expresses the properties he expects. A traditional approach to this is to engage a team of programmers to build a *prototype*, a quickly assembled but necessarily bad and simplified realization. This can then be given to the customer to test. Of course, such an approach is indispensable for some aspects of the software to be developed. For example, there can be no better way to test a user interface than by playing with some version of it; going through sample sessions with the system seems to be the only way for a user to get a feel for what working with the system will be like. In general, however, the prototyping approach has a number of disadvantages. First, it involves some extra work to produce a system that is then thrown away. More importantly, if the original specification is loose (and it usually is) then any prototype will incorporate choices between the alternative behaviours permitted by the specification, and these choices need not necessarily be mirrored in the final implementation. Consequently, the user may conclude that the system will have some properties that are not ensured by the specification at all, and this undermines the sense of the whole exercise. See [HJ 89] for further convincing arguments in this direction.

The overhead of prototyping may be avoided through the use of a rapid prototyping system like RAP [Huß 85]. This demands that requirements specifications be written in an executable specification language, not far from high-level programming languages like Standard ML [Pau 91]. In the fundamental trade-off between executability and expressiveness, it is clearly the latter that is of central importance in a language intended for writing requirements specifications, so such a strong restriction seems highly undesirable.

We believe that for many purposes prototyping should be replaced by *theorem proving* (see [GH 80] for a similar observation). To check whether a given specification indeed embodies a desirable property, it seems most appropriate to state this property explicitly and then try to prove that it is a consequence of the specification. This is the most general form of specification testing; the more usual approaches via rapid prototyping, symbolic evaluation, term rewriting etc. can easily be seen as special cases, or rather as special techniques of theorem proving applicable in particular situations.

Example 1 (continued)

In *INTLIST*, the axioms for *head*, *tail* and *is_in* virtually constitute a prototype implementation in e.g. SML. In an SML prototype, we would for example be able to evaluate expressions like $head(tail(\texttt{[2,4]}))$, $head(tail(\texttt{[2,4,2]}))$, $head(tail(\texttt{[2,4,5,8]}))$, obtaining an integer value (`4` in all these cases). However, rather than testing all these instances, we are able to *prove* directly from the specification a more general fact:

$$INTLIST \models \forall l{:}\texttt{int list}.\, head(tail(\texttt{2::4::}l)) = \texttt{4}$$

We should point out here though that some rapid prototyping systems allow the user to do somewhat more than evaluating just ground instances of $head(tail(\texttt{2::4::}l))$. For example, in RAP [Huß 85], we could in fact evaluate $head(tail(\texttt{2::4::}l))$ obtaining $\texttt{4}$, as expected.

We can also prove the following fact:

$$SORT \models le(1,2) = \texttt{true} \Longrightarrow head(sort(\texttt{[1,2,1]})) = \texttt{1}$$

However:

$$SORT \not\models le(1,2) = \texttt{true} \Longrightarrow sort(\texttt{[1,2,1]}) = \texttt{[1,1,2]}$$
$$SORT \not\models le(1,2) = \texttt{true} \Longrightarrow sort(\texttt{[1,2,1]}) = \texttt{[1,2]}$$

even though a naive prototype implementation would probably satisfy one of these two equations. This would be misleading and potentially dangerous since $SORT$ is loose: it does not specify whether or not $sort$ should preserve repetitions. Sorting functions yielding either of these two results would be acceptable, and so would $sort(\texttt{[1,2,1]}) = \texttt{[1,1,2,2]}$. □

5 Proof systems for specifications

The above discussion indicated a need for formal proof systems for deriving consequences of specifications. Proof is also required for verifying the correctness of refinement steps, see below. There are two levels at which proof is necessary: first, we have to be able to derive consequences of axioms in the underlying institution ($\Phi \vdash \varphi$); second, we have to be able to derive consequences of a specification built in a structured way ($SP \vdash \varphi$). The first problem is familiar from logic, but the second has received much less attention. Here are inference rules that allow such consequences to be derived from specifications built using the specification-building operations introduced above; see [SB 83, ST 88a, Far 92, Wir 93].

$$\frac{\varphi \in \Phi}{\textbf{sorts } S \textbf{ opns } \Omega \textbf{ axioms } \Phi \vdash \varphi}$$

$$\frac{SP \vdash \varphi}{\textbf{enrich } SP \textbf{ by sorts } S \textbf{ opns } \Omega \textbf{ axioms } \Phi \vdash \varphi}$$

$$\frac{\varphi \in \Phi}{\textbf{enrich } SP \textbf{ by sorts } S \textbf{ opns } \Omega \textbf{ axioms } \Phi \vdash \varphi}$$

$$\frac{SP \vdash \varphi}{\textbf{hide sorts } S \textbf{ opns } \Omega \textbf{ in } SP \vdash \varphi} \quad \varphi \text{ is a } (Sig(SP) \setminus \langle S, \Omega \rangle)\text{-axiom}$$

$$\frac{SP \vdash \varphi_1 \quad \cdots \quad SP \vdash \varphi_n \qquad \{\varphi_1, \ldots \varphi_n\} \vdash \varphi}{SP \vdash \varphi}$$

The last of these rules constitutes the link between the two levels of proof.

Example 1 (continued)

Here is an example of how these rules may be used in the proof of a simple consequence of the specification $SORT$. For simplicity, all universal quantifiers in formulae are omitted.

$$\cfrac{\cfrac{\cfrac{(1) \qquad\qquad (2) \qquad\qquad (3)}{INTLIST \vdash is\_in(2, \texttt{0::2::}l) = \texttt{true}}}{SORT1 \vdash is\_in(2, \texttt{0::2::}l) = \texttt{true}}}{SORT \vdash is\_in(2, \texttt{0::2::}l) = \texttt{true}}$$

where (1) is the following proof tree:

$$\dfrac{\dfrac{(\mathit{is\_in}(x, x::l) = \mathtt{true}) \in \text{axioms of } \mathit{INTLIST}}{\mathit{INTLIST} \vdash \mathit{is\_in}(x, x::l) = \mathtt{true}} \qquad \{\mathit{is\_in}(x, x::l) = \mathtt{true}\} \vdash \mathit{is\_in}(2, 2::l) = \mathtt{true}}{\mathit{INTLIST} \vdash \mathit{is\_in}(2, 2::l) = \mathtt{true}}$$

(2) is the following derivation:

$$\dfrac{(x \neq y \Longrightarrow \mathit{is\_in}(x, y::l) = \mathit{is\_in}(x, l)) \in \text{axioms of } \mathit{INTLIST}}{\mathit{INTLIST} \vdash x \neq y \Longrightarrow \mathit{is\_in}(x, y::l) = \mathit{is\_in}(x, l)}$$

and (3) is the following entailment:

$$\{\mathit{is\_in}(2, 2::l) = \mathtt{true}, x \neq y \Longrightarrow \mathit{is\_in}(x, y::l) = \mathit{is\_in}(x, l)\} \vdash \mathit{is\_in}(2, \mathtt{0}::\mathtt{2}::l) = \mathtt{true}$$

□

The above rules provide a *sound* extension to any proof system for the underlying institution: if $\Phi \models \varphi$ whenever $\Phi \vdash \varphi$ for all Φ and φ, then $SP \models \varphi$ whenever $SP \vdash \varphi$ for all SP and φ. *Completeness* ($SP \models \varphi$ implies $SP \vdash \varphi$) is harder to achieve. Even if the proof system for the underlying institution is complete, the above rules do not in general yield a complete proof system for consequences of structured specifications. Whenever there is a discrepancy between model class level and theory level semantics as discussed in Section 3, no complete compositional proof system may be given. This does not exclude the existence of non-compositional complete proof systems that "massage" the structure of specifications in the course of proof, see [Far 92, Wir 93].

Good theorem provers that implement such proof systems are needed. In addition to proof search procedures used in work on automatic theorem proving, they should include heuristics that exploit the structure of specifications to guide proof search, see [SB 83, HST 94].

Example 1 (continued)
A theorem prover attempting to prove that

$$\mathit{SORT} \vdash \forall l{:}\mathtt{int\ list}.\ \mathit{is\_in}(2, \mathtt{0}::\mathtt{2}::l) = \mathtt{true}$$

should not waste time searching through the consequences of the axioms added in $\mathit{SORT}1$, but should go straight down to the level of $\mathit{INTLIST}$ where most of the work of the proof needs to be done. The following heuristics would provide the necessary guidance:

- To prove **hide** ... **in** $SP \vdash \varphi$, try to prove $SP \vdash \varphi$.
- To prove **enrich** SP **by sorts** S **opns** Ω **axioms** $\ldots \vdash \varphi$, if φ doesn't use any of the new operation names in Ω, try to prove $SP \vdash \varphi$.

The latter of the above simple heuristics, even though it does not yield a complete proof method, in practice often helps to greatly reduce the proof search space (cf. [SB 83, HST 94]).

□

It would also be extremely useful for a theorem prover, in the case where it fails to find a proof, to provide the user with readable information on where the proof attempt breaks down, and perhaps even how the specification may be augmented to make the proof go through — a desirable feature which few contemporary theorem provers exhibit.

6 Program development

Given a specification SP, the programming task it defines is to construct a program P that is a correct realization of SP, that is, such that $[\![P]\!] \in [\![SP]\!]$.

There can be no universal recipe that would ensure successful development of a program implementing a given specification. All we can hope to offer are methodologies, and perhaps some particular techniques and heuristics oriented towards specific problem areas.

Perhaps the most fundamental point is that it is neither easy nor desirable to leap in a single bound over the gap between a high-level user-oriented requirements specification and the realm of programs full of technical decisions and algorithmic details. Program development should proceed systematically in a stepwise fashion, gradually enriching the original requirements specification with more and more detail, incorporating more and more design and implementation decisions. Such decisions include choosing between the options of behaviour left open by the specification, between the algorithms that realize this behaviour, between data representation schemes, etc. Each such decision should be recorded separately, as a separate step hopefully consisting of a local modification to the specification. The program development process is then a sequence of such small, easy to understand and easy to verify steps:

$$SP_0 \rightsquigarrow SP_1 \rightsquigarrow \cdots \rightsquigarrow SP_n$$

In such a chain, SP_0 is the original requirements specification and $SP_{i-1} \rightsquigarrow SP_i$ for any $i = 1, \ldots, n$ is an individual *refinement step*. The aim is to reach a specification (here, SP_n) that is an exact description of a program in full detail, with all the technical decisions incorporated (it may simply *be* a program, if our specification formalism is rich enough).

Example 1 (continued)

The following adds to the specification *SORT* the decision that the sorting operation *sort* should preserve the number of occurrences of elements:

$$
\begin{array}{lll}
\mathit{SORTperm} = & \multicolumn{2}{l}{\textbf{hide}\ \mathit{count}\ \textbf{in}} \\
& \multicolumn{2}{l}{\quad \textbf{enrich}\ \mathit{SORT}\ \textbf{by}} \\
& \quad\quad \textbf{opns} & \mathit{count} : \mathtt{int} \times \mathtt{int\ list} \to \mathtt{int} \\
& \quad\quad \textbf{axioms} & \forall x{:}\mathtt{int}.\, \mathit{count}(x, \mathtt{nil}) = 0 \\
& & \forall x, y{:}\mathtt{int}.\, \forall l{:}\mathtt{int\ list}. \\
& & \qquad\qquad x \neq y \Longrightarrow \mathit{count}(x, y::l) = \mathit{count}(x, l) \\
& & \forall x{:}\mathtt{int}.\, \forall l{:}\mathtt{int\ list}.\, \mathit{count}(x, x::l) = \mathtt{1+}\mathit{count}(x, l) \\
& & \forall x{:}\mathtt{int}.\, \forall l{:}\mathtt{int\ list}.\, \mathit{count}(x, l) = \mathit{count}(x, \mathit{sort}(l))
\end{array}
$$

Then we choose the algorithm (insertion sort) and "code" *sort* but leave the additional insertion operation *insert* specified only.

$$
\begin{array}{lll}
\mathit{INS} = & \multicolumn{2}{l}{\textbf{enrich}\ \mathit{INTLIST}\ \textbf{by}} \\
& \quad \textbf{opns} & \mathit{insert} : \mathtt{int} \times \mathtt{int\ list} \to \mathtt{int\ list} \\
& \quad \textbf{axioms} & \forall x{:}\mathtt{int}.\, \forall l{:}\mathtt{int\ list}.\, \exists l1, l2{:}\mathtt{int\ list}. \\
& & \qquad \mathit{insert}(x, l) = l1\mathtt{@}(x::l2) \wedge l = l1\mathtt{@}l2 \\
& & \qquad \wedge\ (\forall l1'{:}\mathtt{int\ list}.\, \forall y{:}\mathtt{int}.\, l1 = l1'\mathtt{@[}y\mathtt{]} \Longrightarrow \mathit{le}(y, x) = \mathtt{true}) \\
& & \qquad \wedge\ (\forall l2'{:}\mathtt{int\ list}.\, \forall y{:}\mathtt{int}.\, l2 = y::l2' \Longrightarrow \mathit{le}(x, y) = \mathtt{true})
\end{array}
$$

$$
\begin{array}{rl}
SORTins = & \textbf{hide } insert \textbf{ in} \\
& \quad \textbf{enrich } INS \textbf{ by} \\
& \qquad \begin{array}{ll}
\textbf{opns} & sort : \texttt{int list} \to \texttt{int list} \\
\textbf{axioms} & sort(\texttt{nil}) = \texttt{nil} \\
& sort(x::l) = insert(x, sort(l))
\end{array}
\end{array}
$$

Finally, we code *insert*:

$$
\begin{array}{rl}
INSdone = & \textbf{enrich } INTLIST \textbf{ by} \\
& \qquad \begin{array}{ll}
\textbf{opns} & insert : \texttt{int} \times \texttt{int list} \to \texttt{int list} \\
\textbf{axioms} & insert(x, \texttt{nil}) = [x] \\
& le(x,y) = \texttt{true} \Longrightarrow insert(x, y::l) = x::y::l \\
& le(x,y) = \texttt{false} \Longrightarrow insert(x, y::l) = y::insert(x,l)
\end{array}
\end{array}
$$

$$
\begin{array}{rl}
SORTdone = & \textbf{hide } insert \textbf{ in} \\
& \quad \textbf{enrich } INSdone \textbf{ by} \\
& \qquad \begin{array}{ll}
\textbf{opns} & sort : \texttt{int list} \to \texttt{int list} \\
\textbf{axioms} & sort(\texttt{nil}) = \texttt{nil} \\
& sort(x::l) = insert(x, sort(l))
\end{array}
\end{array}
$$

The above constitutes a sequence of development steps:

$$SORT \rightsquigarrow SORTperm \rightsquigarrow SORTins \rightsquigarrow SORTdone$$

SORTdone may be viewed as a final implementation of the original specification since the axioms in *INTLIST*, *INSdone* and *SORTdone* amount to SML code (this disregards the fact that *le* is only specified as a partial order, rather than being coded as a specific order relation). We will make this more explicit in the next section. □

A formal definition of such refinement steps $SP \rightsquigarrow SP'$ must incorporate the requirement that any correct final realization of SP' must be a correct realization of SP. This leads to the following straightforward definition [SW 83, ST 88b]:

$$SP \rightsquigarrow SP' \quad \textbf{iff} \quad [\![SP']\!] \subseteq [\![SP]\!]$$

(This presupposes that $Sig(SP) = Sig(SP')$.)

Example 1 (continued)

The refinement steps in the above example satisfy the definition. (The reader is encouraged to check the details.) This is trivial for the step $SORT \rightsquigarrow SORTperm$, since *SORTperm* just adds a constraint on the class of models of *SORT*. For $SORTperm \rightsquigarrow SORTins$, it is necessary to prove that to each model of *SORTins*, we can add *count* and *is_sorted* so that the axioms of *SORTperm* are satisfied. Since *count* and *is_sorted* are determined by the corresponding axioms in *SORTperm*, this amounts to proving that the "code" for *sort* satisfies the axioms of *SORTperm*, assuming that *insert* satisfies the axioms in *INS*, and *count* and *is_sorted* satisfy their axioms. Finally, $SORTins \rightsquigarrow SORTdone$ requires a proof that the "code" for *insert* in *INSdone* satisfies the axioms in *INS*.

It is perhaps worth noticing that $[\![SORTperm]\!] = [\![SORTins]\!] = [\![SORTdone]\!]$; this means that the last two refinement steps are semantically trivial although this does not mean that the proofs are trivial. The reader may be worried by the fact that it then follows that, for example, $SORTdone \rightsquigarrow SORTperm$. The notion of refinement is not fine enough to capture the sense in which *SORTdone* is "closer" to a program than *SORTperm* is. A more elaborate notion of refinement, which provides a place to record "progress" towards a program, will be presented in the next section. □

The definition of refinement ensures that the correctness of the final outcome of the stepwise development process may be inferred from the correctness of the individual refinement steps:

$$\frac{SP_0 \leadsto SP_1 \leadsto \cdots \leadsto SP_n \qquad A \in [\![SP_n]\!]}{A \in [\![SP_0]\!]}$$

The proof is by an easy induction on the length of the refinement sequence.

Notice that if the final specification SP_n represents an individual program P, i.e. $[\![SP]\!] = \{[\![P]\!]\}$, then the conclusion that $A \in [\![SP_0]\!]$ for all $A \in [\![SP_n]\!]$ is just our original statement of the program development task: $[\![P]\!] \in [\![SP_0]\!]$.

An indirect way to prove the correctness of the final outcome is to notice a stronger fact, namely that consecutive refinements can be composed (referred to as "vertical composability" [GB 80]):

$$\frac{SP \leadsto SP' \qquad SP' \leadsto SP''}{SP \leadsto SP''}$$

The above gives a formal view of the stepwise development methodology. As mentioned before, there can be no universal recipe for coming up with useful refinements of a given specification — necessarily, this is the place where the developer's invention is required. Once a refinement step is proposed, though, we should be able to prove it correct, that is, we should have some formalism for proving the inclusion between the corresponding model classes. Of course, this must incorporate a theorem prover for the underlying logic, and for proving consequences of structured specifications, as discussed above. A new need that arises here is that of proving entailments between two structured specifications (we write $SP' \models SP$ to state that every model of SP' is a model of SP, yet another formulation of $SP \leadsto SP'$ that we will use in this context). If the structures of SP and SP' match exactly (and the specification-building operations used are monotonic w.r.t. inclusion of model classes — this is typically the case) then this problem may be reduced to proving that individual axioms (from SP) are consequences of certain specifications (parts of SP') via the following fact (referred to as "horizontal composability" [GB 80] for the specification-building operation op):

$$\frac{SP_1 \leadsto SP'_1 \qquad \cdots \qquad SP_n \leadsto SP'_n}{op(SP_1, \ldots, SP_n) \leadsto op(SP'_1, \ldots, SP'_n)}$$

Unfortunately, the structures of the two specifications need not coincide, which makes such a reduction very non-trivial. The only work on this important problem we are aware of is [Far 92, Wir 93].

Example 1 (continued)

The refinements in Example 1 illustrate the point that the structure of the final implementation differs from that of the original specification, even though the difference is only that different auxiliary operations are used (*is_sorted* versus *insert*). The essential change happens in the step *SORTperm* $\leadsto$ *SORTins*. There is no way to build this refinement using the horizontal composability rule: stripping off the hiding operations from both *SORTperm* and *SORTins* (naively disregarding the fact that different things are hidden) yields two incomparable specifications. See [Far 92, Wir 93] for proof rules that allow the user to handle this situation in a different, non-compositional way. □

Horizontal composability should not be misread as a directive to decompose a specification $SP = op(SP_1, \ldots, SP_n)$ into separate tasks $SP_1, \ldots, SP_n$. It is possible for the design decisions taken in the solutions of these separate tasks to conflict so that even once we have obtained realizations of $SP_1, \ldots, SP_n$, it might not be possible to combine these to form a realization of SP.

Example 2

Consider the following specification:

$$\begin{array}{rl} SPc = & \textbf{enrich opns} \quad c : \texttt{int} \\ & \qquad\quad \textbf{axioms} \ \ 1 < c < 15 \\ & \textbf{by axioms} \ 10 < c < 27 \end{array}$$

Since

$$\left(\begin{array}{ll} \textbf{opns} & c : \texttt{int} \\ \textbf{axioms} & 1 < c < 15 \end{array}\right) \rightsquigarrow \left(\begin{array}{ll} \textbf{opns} & c : \texttt{int} \\ \textbf{axioms} & 1 < c < 12 \end{array}\right)$$

and

$$\left(\begin{array}{ll} \textbf{opns} & c : \texttt{int} \\ \textbf{axioms} & 10 < c < 27 \end{array}\right) \rightsquigarrow \left(\begin{array}{ll} \textbf{opns} & c : \texttt{int} \\ \textbf{axioms} & 14 < c < 20 \end{array}\right)$$

we also have

$$SPc \rightsquigarrow \left(\begin{array}{l} \textbf{enrich opns} \quad c : \texttt{int} \\ \qquad\quad \textbf{axioms} \ \ 1 < c < 12 \\ \textbf{by axioms} \ 14 < c < 20 \end{array}\right)$$

However, even though SPc is consistent, and both of the resulting component specifications are consistent as well, the resulting composed specification to which SPc is refined is inconsistent!

This happens because the two specification arguments to the **enrich** operation implicitly share a loosely specified part (c : `int` in the example). If the decisions constraining this common part in separate developments of the two specifications are different, as above, then putting the resulting specifications together may yield inconsistency. □

An issue which may seem worrying here is that we have not put into our definition of refinement any requirement that the refined specification is consistent. Indeed, this can be seen as a problem, since an inconsistent specification cannot be implemented by any program, and so it opens a blind valley in the program development process. From this point of view, it would be worthwhile to be able to check consistency of each specification as soon as it is formulated. Unfortunately, in general (for any sufficiently powerful specification framework) this is an undecidable property. Fortunately, inconsistency of specifications cannot lead to incorrect programs: if we arrive at a program at some point in the development process, then this program is by definition consistent (it has a unique model) and consequently, all the specifications leading to it must have been consistent as well.

The proposed methodology of stepwise refinement does not and cannot be expected to guarantee success. Apart from inconsistencies, there are many sources of "blind valleys" and failures in the development process: there might be no computable realization of a specification, there might be no "computationally feasible" realization, we might not be clever enough to find a realization, we might run out of money to finish the project, etc.

Example 3

Consider a specification of natural numbers with the pre-ordering *le* specified by the axiom:

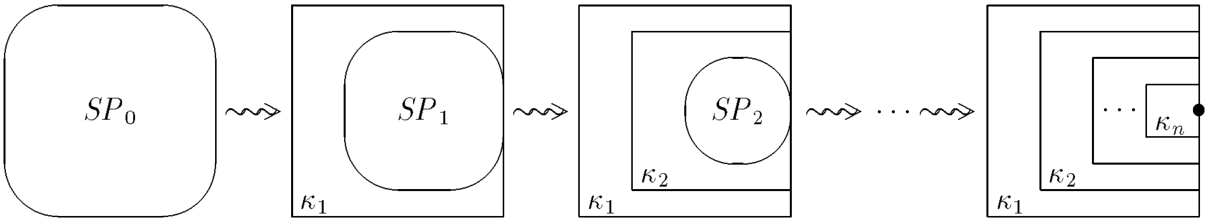
$$le(m,n) = \mathtt{true} \iff \forall x{:}nat.\, M_m \downarrow x \Longrightarrow M_n \downarrow x$$

where for all natural numbers k and x, the predicate $M_k \downarrow x$ is specified to mean that the Turing machine with Gödel number k terminates on input x. This specification is consistent but it has no computable models since the halting problem is undecidable. □

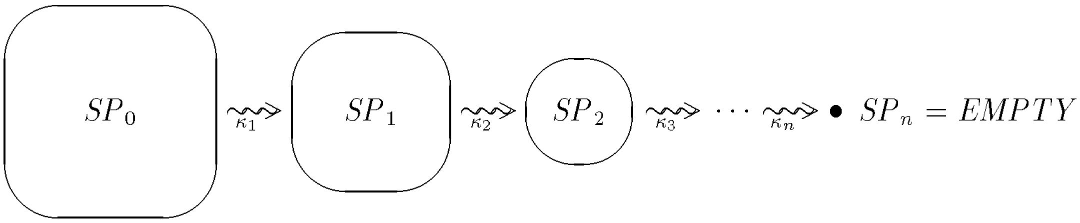
The main feature of the methodology we really can ensure is its *safety*: if we arrive at a program, then it is a correct realization of the original specification.

7 Constructor implementations

The simple notion of specification refinement is mathematically elegant and powerful enough (in the context of a sufficiently rich specification language) to handle all concrete examples of interest. However, it is not very convenient to use in practice. During the process of developing a program, the successive specifications incorporate more and more details arising from successive design decisions. Thereby, some parts become fully determined, and remain unchanged as a part of the specification until the final program is obtained.



It is more convenient to avoid such clutter by separating the finished parts from the specification, putting them aside, and proceeding with the development of the unresolved parts only.



It is important for the finished parts $\kappa_1, \ldots, \kappa_n$ to be independent of the particular choice of realization for what is left: they should act as constructions extending any realization of the unresolved part to a realization of what is being refined.

Example 1 (continued)

An instance of the situation illustrated above may be found in the consecutive refinement steps *SORTperm* $\leadsto$ *SORTins* $\leadsto$ *SORTdone*. Simply note that the "code" for *sort* introduced in *SORTins*, and the operation that hides *insert*, are still present in the same form in *SORTdone*. □

Each κ_i above amounts to what is known as a *parameterised program* [Gog 84] with input interface SP_i and output interface SP_{i-1}. Given a program P that is a correct realization of SP_i, the parameterised program κ_i may be instantiated to yield a program $\kappa_i(P)$ that realizes SP_{i-1}. A programming language that supports stepwise development in the style suggested here needs to provide syntax and modularisation facilities for defining parameterised programs and their instantiations. For example, in the Standard ML programming language [Pau 91] parameterised programs are called *functors*, and instantiation amounts to *functor application*.[2] Once the development is finally finished (that is, when nothing is left unresolved, as above) we can successively instantiate the parameterised programs $\kappa_n, \ldots, \kappa_1$ to obtain a correct realization of the original specification SP_0.

Semantically, each parameterised program κ_i defines a function (which we will call a *constructor*) on algebras, $[\![\kappa_i]\!] : Alg(Sig(SP_i)) \rightarrow Alg(Sig(SP_{i-1}))$, and the semantics of instantiation is simply function application: if $[\![P]\!] \in Alg(Sig(SP_i))$, then $[\![\kappa_i(P)]\!] = [\![\kappa_i]\!]([\![P]\!])$. In practice, κ_i provides a definition of the components (carriers and operations) of a $Sig(SP_{i-1})$-algebra, given the components of a $Sig(SP_i)$-algebra.

Example 1 (continued)

Consider the refinement $SORTperm \rightsquigarrow SORTins$ in which "code" for *sort* is first introduced. Using a notation like that of Standard ML, a parameterised program corresponding to this step can be expressed as follows:

```
functor K1(X:INS):SORTperm =
   struct
      open X
      fun sort(nil) = nil
        | sort(x::l) = insert(x,sort(l))
   end
```

(The reader is asked to find the obvious correspondence between the names used here and those used — in a different font — in the specifications.)

Recall that *INS* is the part of *SORTins* that remains after "peeling off" *sort* and the operation of hiding *insert*, the part of the specification whose implementation is fixed in this step. Notice that the functor definition provides not only code for *sort* but also (implicitly) realizes the hiding of *insert* since *insert* is not present in the functor result signature.

The next refinement step, $SORTins \rightsquigarrow SORTdone$, which introduces code for *insert*, corresponds to the following parameterised program:

```
functor K2(X:INTLIST):INS =
   struct
      open X
      fun insert(x,nil) = [x]
        | insert(x,y::l) =
                if le(x,y)
                then insert(x,y::l) = x::y::l
                else insert(x,y::l) = y::insert(x,l)
```

[2] In the following we disregard the fact that functor application in SML is not guaranteed to terminate. The technicalities may be modified to capture this by modelling parameterised programs as *partial* (rather than total) functions and adding the obvious definedness condition in the definition of constructor implementation [ST 89]. We resist the temptation to adopt this slightly more complex approach for the sake of clarity of presentation.

```
end
```

The code for *sort*, which in the original refinement step was still present in *SORTdone*, has been dealt with in the previous step. Thus in this step we are able to focus on what remains, namely the *insert* operation, without the distraction of the surrounding context.

The axioms in *INTLIST* may be translated directly into SML code, and we can choose a particular realization to implement *le*, giving the following parameterised program:

```
functor K3(X:EMPTY):INTLIST =
   struct
      fun le(n,m:int) = n <= m
      fun head(x::_) = x
      fun tail(_::l) = l
      fun is_in(_,nil) = false
        | is_in(x,y::l) = if x=y then true else is_in(x,l)
   end
```

Here, `EMPTY` stands for the empty SML signature `sig end`.

To finish the example, we need to provide a parameterised program corresponding to the refinement step $SORT \rightsquigarrow SORTperm$. Since all that is done in this step is to impose a (non-constructive) restriction on the class of permissible realizations of *sort*, this is trivial:

```
functor K0(X:SORTperm):SORT = X
```

□

The above considerations motivate a more elaborate version of the notion of refinement of the previous section, known as *constructor implementation* [ST 88b]. We write $SP \underset{\kappa}{\rightsquigarrow} SP'$ to say that a specification SP' implements a specification SP via κ, where κ is a parameterised program denoting a constructor $[\![\kappa]\!] : Alg(Sig(SP')) \rightarrow Alg(Sig(SP))$, and define this as follows:

$$SP \underset{\kappa}{\rightsquigarrow} SP' \quad \textbf{iff} \quad [\![\kappa]\!]([\![SP']\!]) \subseteq [\![SP]\!]$$

Here, $[\![\kappa]\!]([\![SP']\!])$ is the image of $[\![SP']\!]$ under $[\![\kappa]\!]$.

Example 1 (continued)

The following are examples of constructor implementations: $SORT \underset{\mathtt{K0}}{\rightsquigarrow} SORTperm$, $SORTperm \underset{\mathtt{K1}}{\rightsquigarrow} INS$, $INS \underset{\mathtt{K2}}{\rightsquigarrow} INTLIST$, $INTLIST \underset{\mathtt{K3}}{\rightsquigarrow} EMPTY$. The justification requires proofs similar to those sketched in Example 1 in Section 6 for the corresponding refinement steps. □

For each parameterised program κ we can (in principle) define a new specification-building operation $\overline{\kappa}$ such that $[\![\overline{\kappa}(SP')]\!] = [\![\kappa]\!]([\![SP']\!])$; then constructor implementations may be viewed as refinements ($SP \underset{\kappa}{\rightsquigarrow} SP'$ is equivalent to $SP \rightsquigarrow \overline{\kappa}(SP')$). Provided that we have means for reasoning about specifications built using these new operations, the correctness of constructor implementations may be established using proof techniques for refinements. Specifically, we need a way of deriving entailments of the form $\overline{\kappa}(SP') \models SP$; this boils down to proving properties of the components of programs built by κ.

The correctness of the final outcome of the stepwise development process may be inferred from the correctness of the individual constructor implementation steps:

$$\frac{SP_0 \underset{\kappa_1}{\rightsquigarrow} SP_1 \underset{\kappa_2}{\rightsquigarrow} \cdots \underset{\kappa_n}{\rightsquigarrow} SP_n = EMPTY}{[\![\kappa_1(\kappa_2(\ldots\kappa_n(empty)\ldots))]\!] \in [\![SP_0]\!]}$$

where $EMPTY$ is the empty specification over the empty signature and $empty$ is its (empty) realization.

Example 1 (continued)
In our example,

```
K0(K1(K2(K3(empty))))
```

yields a non-parameterised program (an SML structure) satisfying $SORT$. Here, `empty` stands for the empty SML structure `struct end`. □

Suppose that parameterised programs compose, that is, for any two parameterised programs κ and κ' such that the signature of the output interface of κ coincides with the signature of the input interface of κ', there is a parameterised program $\kappa;\kappa'$ with $[\![\kappa;\kappa']\!] = [\![\kappa]\!];[\![\kappa']\!]$ (the latter semicolon stands for ordinary function composition, written in diagrammatic order). Then it is easy to see that constructor implementations (vertically) compose:

$$\frac{SP \underset{\kappa}{\leadsto\!\!>} SP' \qquad SP' \underset{\kappa'}{\leadsto\!\!>} SP''}{SP \underset{\kappa';\kappa}{\leadsto\!\!>} SP''}$$

The requirement that parameterised programs can be instantiated is a weaker requirement than that parameterised programs be composable, even though any programming language with decent modularisation facilities should ensure the latter as well. In Standard ML, there is no explicit functor composition operation but the composite of two functors may easily be defined using functor application and abstraction.

As in the case of refinement, vertical composability is not necessary to ensure the correctness of the outcome of the development process. All we need is the condition inherent in the definition of constructor implementation, namely that implementations reflect realizations:

$$\frac{SP \underset{\kappa}{\leadsto\!\!>} SP' \qquad A' \in [\![SP']\!]}{[\![\kappa]\!](A') \in [\![SP]\!]}$$

Many approaches to implementation (see e.g. [EKMP 82, SW 82, Ore 83]) make use of a restrictive kind of constructor defined by a parameterised program having a particular rigid form. Then the vertical composition of two implementations is required to yield an implementation of the same form, which is not always possible. The requirement that the composition of parameterised programs be forced into some given normal form corresponds to requiring programs to be written in a rather restricted programming language.

We have already mentioned that the internal structure of a requirements specification need not be mirrored by programs that realize it. This is why the definitions of refinement and constructor implementation above take no account of the structure of specifications. However, when developing a large program it is crucial to progressively decompose the job into smaller tasks that can be handled separately. Each task is defined by a specification, and solving a task means producing a program component that satisfies this specification. Once all tasks are solved, producing the final system is a simple matter of appropriately assembling these components.

A development step involving the decomposition of a programming task into separate subtasks is modelled using a constructor implementation with a multi-argument parameterised program (see [SST 92]):

$$SP \underset{\kappa}{\leadsto\!\!>} \langle SP_1, \ldots, SP_n \rangle \quad \textbf{iff} \quad [\![\kappa]\!]([\![SP_1]\!] \times \cdots \times [\![SP_n]\!]) \subseteq [\![SP]\!]$$

where $[\![\kappa]\!] : Alg(Sig(SP_1)) \times \cdots \times Alg(Sig(SP_n)) \to Alg(Sig(SP))$ is an n-argument constructor (an n-argument function on algebras) describing a way to put models of $SP_1, \ldots, SP_n$ together to construct a model of SP. Now the development process takes on a tree-like shape. This process is finished once a tree is obtained that has empty sequences (of specifications) as its leaves:

$$SP \underset{\kappa}{\rightsquigarrow} \begin{cases} SP_1 \underset{\kappa_1}{\rightsquigarrow} \langle\,\rangle \\ \vdots \\ SP_n \underset{\kappa_n}{\rightsquigarrow} \begin{cases} SP_{n1} \underset{\kappa_{n1}}{\rightsquigarrow} \left\{ SP_{n11} \underset{\kappa_{n11}}{\rightsquigarrow} \langle\,\rangle \right. \\ \cdots \\ SP_{nm} \underset{\kappa_{nm}}{\rightsquigarrow} \langle\,\rangle \end{cases} \end{cases}$$

Then an appropriate instantiation of the parameterised programs in the tree yields a realization of the original requirements specification. The above development tree yields the program $\kappa(\kappa_1(), \ldots, \kappa_n(\kappa_{n1}(\kappa_{n11}()), \ldots, \kappa_{nm}()))$, with

$$[\![\kappa(\kappa_1(), \ldots, \kappa_n(\kappa_{n1}(\kappa_{n11}()), \ldots, \kappa_{nm}()))]\!] \in [\![SP]\!].$$

(We use an obvious notation $\kappa(P_1, \ldots, P_n)$ for instantiation of n-ary parameterised programs, where $[\![\kappa(P_1, \ldots, P_n)]\!] = [\![\kappa]\!]([\![P_1]\!], \ldots, [\![P_n]\!])$.)

The structure of the final program is determined by the shape of the development tree, which is in turn determined by the decomposition steps. Each such step corresponds to what software engineers call a *design* specification (and what [GHW 82] call an *organizational* specification): it defines the structure of the system by specifying its components and describing how they fit together. This style of development leads to modular programs, built from fully specified, correct and reusable components.

Example 1 (continued)

We show a simple example of a decomposition using a modified version of the sorting specification above:

$$\begin{array}{l} \mathit{SORTonce} = \ \textbf{hide}\ \mathit{all\_once}\ \textbf{in} \\ \qquad \textbf{enrich}\ \mathit{SORT}\ \textbf{by} \\ \qquad\quad \textbf{opns} \quad \mathit{all\_once} : \texttt{int list} \to \texttt{bool} \\ \qquad\quad \textbf{axioms} \quad \mathit{all\_once}(\texttt{nil}) = \texttt{true} \\ \qquad\qquad\qquad \forall x{:}\texttt{int}.\, \forall l{:}\texttt{int list}. \\ \qquad\qquad\qquad\quad \mathit{all\_once}(x::l) = \texttt{true} \iff \\ \qquad\qquad\qquad\qquad \mathit{all\_once}(l) = \texttt{true} \land \mathit{is\_in}(x,l) = \texttt{false} \\ \qquad\qquad\qquad \forall l{:}\texttt{int list}.\ \mathit{all\_once}(\mathit{sort}(l)) = \texttt{true} \end{array}$$

SORTonce just adds to *SORT* the requirement that the result of *sort* does not contain multiple occurrences of elements. Clearly, $\mathit{SORT} \rightsquigarrow \mathit{SORTonce}$.

Consider an additional specification that introduces a function specified to remove adjacent occurrences of the same element in a list:

$$
\begin{array}{lll}
NOSTUTTER = & \mathbf{enrich}\ INTLIST\ \mathbf{by} & \\
& \quad \mathbf{opns} & rem\_stutter : \mathtt{int\ list} \to \mathtt{int\ list} \\
& \quad \mathbf{axioms} & \forall l, l1, l2{:}\mathtt{int\ list}.\forall x, y{:}\mathtt{int}. \\
& & \quad rem\_stutter(l) = l1 \mathtt{@}(x::y::l2) \Longrightarrow x \neq y \\
& & \forall l{:}\mathtt{int\ list}.\forall x{:}\mathtt{int}. \\
& & \quad is\_in(x, l) = is\_in(x, rem\_stutter(l)) \\
& & \forall l, l1, l2{:}\mathtt{int\ list}. \\
& & \quad rem\_stutter(l) = l1 \mathtt{@} l2 \Longrightarrow \\
& & \qquad \exists l3, l4{:}\mathtt{int\ list}. \\
& & \qquad\quad l = l3 \mathtt{@} l4 \wedge rem\_stutter(l3) = l1 \\
& & \qquad\qquad\qquad \wedge\ rem\_stutter(l4) = l2
\end{array}
$$

Now, the problem of implementing the specification *SORTonce* may be decomposed into the problems of implementing *SORTperm* and *NOSTUTTER* as follows:

$$SORTonce \underset{\mathtt{K4}}{\rightsquigarrow} \langle SORTperm, NOSTUTTER \rangle$$

where the parameterised program `K4` is given as follows:

```
functor K4(X:SORTperm,Y:NOSTUTTER):SORTonce =
   struct
      open Y
      fun sort(l) = rem_stutter(X.sort(l))
   end
```

The specification *NOSTUTTER* can easily be implemented using the following functor:

```
functor K5(Z:INTLIST):NOSTUTTER =
   struct
      fun rem_stutter(nil) = nil
        | rem_stutter([x]) = [x]
        | rem_stutter(x::y::l) =
             if x = y then rem_stutter(y::l) else x::rem_stutter(y::l)
   end
```

Since we already have an implementation of *SORTperm* (obtained entirely independently from the development for *NOSTUTTER*) and of *INTLIST*, the development is complete and we can put all these together to obtain the following realization of the specification *SORTonce*:

```
K4(K1(K2(K3(empty))),K5(K3(empty))) : SORTonce
```

□

Horizontal composability for constructor implementations takes the form:

$$\frac{SP_1 \underset{\kappa_1}{\rightsquigarrow} SP'_1 \qquad \cdots \qquad SP_n \underset{\kappa_n}{\rightsquigarrow} SP'_n}{op(SP_1, \ldots, SP_n) \rightsquigarrow op(\overline{\kappa_1}(SP'_1), \ldots, \overline{\kappa_n}(SP'_n))}$$

The problem illustrated by Example 2 still exists, but it cannot arise when *op* corresponds to a parameterised program, as in the decomposition steps via multi-argument parameterised programs above.

8 Specifying and developing parameterised programs

The enterprise of formal specification and development is relevant to parameterised programs, with exactly the same motivation as in the case of ordinary non-parameterised programs. An additional advantage this brings is that it enables the overall shape of a development tree (see above) to be given without the need to supply the parameterised programs involved in each of the steps. The provision of a parameterised program that fits into each step can then be regarded as a separate task, perhaps involving further refinement and decomposition. This is one of the ideas underlying the Extended ML methodology for formal development of Standard ML programs from specifications [ST 89, ST 91, San 91, Kaz 92].

For any two signatures Σ and Σ', we can regard $\Sigma \to \Sigma'$ as a new kind of signature. Then $Alg(\Sigma \to \Sigma')$ is the set of all *parametric* $(\Sigma \to \Sigma')$-*algebras*, that is functions $F : Alg(\Sigma) \to Alg(\Sigma')$. Just as ordinary programs are modelled as algebras, parameterised programs are modelled as parametric algebras. (We generalise this further to multi-argument and higher-order parametric algebras below.)

Note that both "constructor" and "parametric algebra" are names for the same concept: a function mapping algebras to algebras. We use the former when such a function constitutes an implementation step, and the latter when it is itself the goal of the development process. This distinction is blurred below, especially once the extension to higher-order is considered. Another difference is that a constructor is assumed to be defined by a parameterised program, while a parametric algebra is an arbitrary set-theoretic function.

To specify a parameterised program, we give its input and output interfaces. The specification $SP \to SP'$ describes the class of parametric $(Sig(SP) \to Sig(SP'))$-algebras $F : Alg(Sig(SP)) \to Alg(Sig(SP'))$ such that $F(A) \in [\![SP']\!]$ for all $A \in [\![SP]\!]$. Said another way, $Sig(SP \to SP') = Sig(SP) \to Sig(SP')$ and $[\![SP \to SP']\!] = \{F \in Alg(Sig(SP) \to Sig(SP')) \mid F([\![SP]\!]) \subseteq [\![SP']\!]\}$. The statement that κ is a realization of $SP \to SP'$ is thus equivalent to the correctness of the constructor implementation $SP' \underset{\kappa}{\rightsquigarrow} SP$. The specification $SP \to SP'$ is *not* a so-called *parameterised specification*; it is a non-parameterised specification of a parameterised program. See [SST 92] for a discussion of this distinction.

Example 1 (continued)

$INTLIST \to SORT$ specifies a parameterised program which, given an implementation of $INTLIST$, delivers an implementation of $SORT$. Two (equivalent) realizations of this specification are the functors:

```
functor K(X:INTLIST):SORT = K0(K1(K2(X)))
functor K'(X:INTLIST):SORT = K1(K2(X))
```

Another, different realization is:

```
functor K''(X:INTLIST):SORT = K4(K1(K2(X)),K5(X))
```

That is, $[\![\texttt{K}]\!], [\![\texttt{K'}]\!], [\![\texttt{K''}]\!] \in [\![INTLIST \to SORT]\!]$. □

The definition of refinement of specifications applies without modification to specifications of parameterised programs:

$$SP_1 \to SP'_1 \rightsquigarrow SP_2 \to SP'_2 \quad \textbf{iff} \quad [\![SP_2 \to SP'_2]\!] \subseteq [\![SP_1 \to SP'_1]\!]$$

which again presupposes that $Sig(SP_1 \to SP'_1) = Sig(SP_2 \to SP'_2)$, i.e. $Sig(SP_1) = Sig(SP_2)$ and $Sig(SP'_1) = Sig(SP'_2)$. A sufficient condition for this refinement to hold is that $SP_2 \rightsquigarrow SP_1$ and $SP'_1 \rightsquigarrow SP'_2$.

Example 1 (continued)

Simple examples of refinements between specifications of parameterised programs may be built on the examples of refinements given in Example 1 in Section 6: $INTLIST \rightarrow SORT$ refines to $INTLIST \rightarrow SORTperm$ which further refines to $INTLIST \rightarrow SORTins$ which refines to $INTLIST \rightarrow SORTdone$. □

Constructor implementations may be similarly generalized to deal with specifications of parameterised programs. However, the parameterised programs used to define the constructors involved in such implementations are then higher-order, i.e. they take parameterised programs as arguments and return parameterised programs as results. Higher-order functors are not available in Standard ML as defined in [MTH 90], but their semantics and implementation is a topic of current active research [Tof 92, MT 94].

The definitions involved in dealing with parameterised programs and their specifications extend to the higher-order case in a natural way [SST 92]. The set of generalized signatures is defined to be the least set containing ordinary signatures and such that if $\Sigma_1, \ldots, \Sigma_n$ $(n \geq 0)$ and Σ are generalized signatures, then $\langle \Sigma_1, \ldots, \Sigma_n \rangle \rightarrow \Sigma$ is a generalized signature as well; if $n = 1$ we omit the brackets. $Alg(\langle \Sigma_1, \ldots, \Sigma_n \rangle \rightarrow \Sigma)$ is the set of all functions $F : Alg(\Sigma_1) \times \cdots \times Alg(\Sigma_n) \rightarrow Alg(\Sigma)$. As before, to specify a higher-order parameterised program, we give its input interfaces and output interface, which now may themselves be specifications of (higher-order) parameterised programs. The definitions are exactly the same as those given above. (Alternatively, multi-argument parameterised programs could be presented in their "curried" form denoting parametric algebras in $Alg(\Sigma_1 \rightarrow (\Sigma_2 \rightarrow \cdots (\Sigma_n \rightarrow \Sigma) \cdots)))$.)

The concepts of constructor implementation and design specification carry over without modification to the case of higher-order parameterised programs and their specifications.

Example 1 (continued)

The example of constructor implementation with decomposition at the end of the last section may be rephrased using these ideas. Instead of implementing $SORTonce$ in terms of realizations of $SORTperm$ and $NOSTUTTER$, we can build a realization of the specification $INTLIST \rightarrow SORTonce$ in terms of realizations of the specifications $INTLIST \rightarrow SORTperm$ and $INTLIST \rightarrow NOSTUTTER$:

$$INTLIST \rightarrow SORTonce \underset{\mathtt{K6}}{\rightsquigarrow\!\!>} \langle INTLIST \rightarrow SORTperm, INTLIST \rightarrow NOSTUTTER \rangle$$

where `K6` is the following higher-order functor (written using an *ad hoc* but hopefully self-explanatory notation):

```
functor K6( F1:INTLIST->SORTperm, F2:INTLIST->NOSTUTTER )
   : INTLIST->SORTonce =
      ( functor (X:INTLIST):SORTonce =
            struct
               open X
               structure S = F1(X)
               structure N = F2(X)
               fun sort l = N.rem_stutter(S.sort(l))
            end )
```

□

It is possible to restrict attention to constructors of a particularly simple form, since any constructor implementation $SP \underset{\kappa}{\rightsquigarrow\!\!>} \langle SP_1, \ldots, SP_n \rangle$ may be replaced by the decomposition

$SP \underset{app}{\rightsquigarrow} \langle SP_\kappa, SP_1, \ldots, SP_n \rangle$, where $SP_\kappa = \langle SP_1, \ldots, SP_n \rangle \rightarrow SP$ and app is the higher-order parameterised program such that $app(F, A_1, \ldots, A_n) = F(A_1, \ldots, A_n)$ and where the parameterised program κ is then provided as the realization of SP_κ. A decomposition like $SP \underset{app}{\rightsquigarrow} \langle SP_\kappa, SP_1, \ldots, SP_n \rangle$ embodies the decision to implement SP in terms of realizations of $SP_1, \ldots, SP_n$, leaving the decision of how these are used to produce a realization of SP as a *separate* development task, specified by SP_κ. This brings a bottom-up flavour into our principally top-down view of the development process.

Example 1 (continued)

Here is the above example once again:

$$INTLIST \rightarrow SORTonce \underset{\mathtt{APP}}{\rightsquigarrow} \langle HOSP, INTLIST \rightarrow SORTperm, INTLIST \rightarrow NOSTUTTER \rangle$$

where

$$HOSP = \langle INTLIST \rightarrow SORTperm, INTLIST \rightarrow NOSTUTTER \rangle \rightarrow (INTLIST \rightarrow SORTonce)$$

and `APP` is the higher-order functor applying its first argument to its second and third argument:

```
functor APP
      (F:(INTLIST->SORTperm,INTLIST->NOSTUTTER)->(INTLIST->SORTonce),
       F1:INTLIST->SORTperm, F2:INTLIST->NOSTUTTER
      ):INTLIST->SORTonce =
   F(F1,F2)
```

This decomposition embodies a decision that the implementation of $INTLIST \rightarrow SORTonce$ *may* use the implementations of $INTLIST \rightarrow SORTperm$ and $INTLIST \rightarrow NOSTUTTER$, to be provided separately. One way of realizing the specification $HOSP$ is the functor `K6` above; another possibility is to use an entirely different solution, ignoring either or both of the realizations of $INTLIST \rightarrow SORTperm$ and $INTLIST \rightarrow NOSTUTTER$. For example:

```
functor K7( F1:INTLIST->SORTperm, F2:INTLIST->NOSTUTTER )
   : INTLIST->SORTonce =
      ( functor (X:INTLIST):SORTonce =
           struct
              open X
              fun insert(x,nil) = x::l
                | insert(x,y::l) =
                   if x=y then y::l
                   else if X.le(x,y)
                         then insert(x,y::l) = x::y::l
                         else insert(x,y::l) = y::insert(x,l)
              fun sort(nil) = nil
                | sort(x::l) = insert(x,sort(l))
           end )
```

□

The above presentation uses a particularly simple form of specification of parameterised programs, where the output interface does not depend on the particular realization of the input interfaces. This is not sufficient when more complex examples are considered. The necessary extra flexibility is gained by replacing the specification $\langle SP_1, \ldots, SP_n\rangle \rightarrow SP$ by the generalized (dependent) product $\Pi\langle X_1{:}SP_1, \ldots, X_n{:}SP_n\rangle.SP[X_1, \ldots, X_n]$. See [SST 92] for details of this and other technicalities omitted here.

Example 1 (continued)

Specifications like $INTLIST \rightarrow SORTonce$ do not capture the intention that their realizations, when given an argument X realizing $INTLIST$, should produce a realization of $SORTonce$ that *extends* X. In particular, a realization of $INTLIST \rightarrow SORTonce$ might ignore the le component of its argument and produce a realization of $SORTonce$ containing a completely different le function, together with a sorting function that is correct with respect to this new le function rather than the one supplied in the argument. This problem can be solved by use of the following dependent product specification:

$$\begin{aligned}\Pi X{:}INTLIST.\ &\textbf{enrich } SORTonce\\ &\textbf{by axioms } \forall x, y{:}\texttt{int}.\ le(x,y) = X.le(x,y)\end{aligned}$$

In case types are involved, this is the issue of *sharing* in Standard ML and the use of so-called *sharing constraints* as in Standard ML and Extended ML [ST 89, ST 91] is one way of expressing the required dependency. □

9 Behavioural implementations

A specification should be a precise and complete statement of required properties. We should try to avoid including extra requirements, even if they happen to be satisfied by a possible future realization. Such over-specification unnecessarily limits the options left open to the implementer. Ideally, the target is to describe exactly the admissible program behaviours. This suggests that specifications of programming tasks should not distinguish between programs (modelled as algebras) exhibiting the same *behaviour*.

The intuitive idea of behaviour of an algebra has been formalised in a number of ways (see e.g. [Rei 81, GM 82, SW 83, Sch 87, ST 87, NO 88]).[3] In most approaches one distinguishes a certain set OBS of sorts as *observable*. Intuitively, these are the sorts of data directly visible to the user (integers, booleans, characters, etc.) in contrast to sorts of "internal" data structures, which are observable only via the functions provided by the program. The behaviour of an algebra is characterised by the set of *observable computations* taking arguments of sorts in OBS and producing a result of a sort in OBS. In the standard algebraic framework, such computations are modelled as terms of sorts in OBS with variables (representing the inputs) of sorts in OBS only. Two Σ-algebras A and B are *behaviourally equivalent* (w.r.t. OBS), written $A \equiv B$, if they exhibit the same behaviour, that is, if all observable computations yield the same results in A and in B. The motivation is related to that of so-called *testing* equivalences studied in the context of concurrent systems [DH 84]. The role of behavioural equivalence in the context of parametric algebras is a topic of current research and we do not treat this here. Therefore this section deals only with ordinary algebras and development of non-parameterised programs.

Example 4

A hackneyed example that illustrates the idea of behavioural equivalence is that of stacks of integers:

[3] The following paragraph makes sense only in an institution in which signatures have sorts. This is not much of a restriction in practice. In any case, what follows thereafter (apart from the examples) applies to any institution and any equivalence relation on its algebras [ST 88b].

$$
\begin{array}{lll}
STACK = & \textbf{sorts} & stack \\
& \textbf{opns} & empty : stack \rightarrow \texttt{bool} \\
& & push : \texttt{int} \times stack \rightarrow stack \\
& & pop : stack \rightarrow stack \\
& & top : stack \rightarrow \texttt{int} \\
& & is\_empty : stack \rightarrow \texttt{bool} \\
& \textbf{axioms} & is\_empty(empty) = \texttt{true} \\
& & \forall s{:}stack.\, \forall n{:}\texttt{int}.\, is\_empty(push(n,s)) = \texttt{false} \\
& & \forall s{:}stack.\, \forall n{:}\texttt{int}.\, top(push(n,s)) = n \\
& & \forall s{:}stack.\, \forall n{:}\texttt{int}.\, pop(push(n,s)) = s
\end{array}
$$

Suppose that the sorts `int` and `bool` (included implicitly in all the specifications we consider) are taken as observable while the sort *stack* is not. The observable computations are all the terms of the form $is\_empty(s)$ and $top(s)$ where s is a term of sort *stack* with variables of sort `int` only.

Two typical algebras which provide intuitively acceptable realizations of this specification can be coded as SML structures as follows:

```
structure S1 : STACK =
    struct
        type stack = int list
        val empty = nil
        fun push (n,s) = n::s
        fun pop (nil) = nil          | pop (_::s) = s
        fun top (nil) = 0            | top (n::_) = n
        fun is_empty (nil) = true | is_empty (_::_) = false
    end

structure S2 : STACK =
    struct
        type stack = (int -> int) * int
        val empty = ((fn k => 0), 0)
        fun push (n, (f, i)) = ((fn k => if k = i then n else f k), i+1)
        fun pop (f, i) = if i = 0 then (f, 0) else (f, i-1)
        fun top (f, i) = if i = 0 then 0 else f i
        fun is_empty (f, i) = (i=0)
    end
```

`S1` gives the obvious realization of stacks as list of integers and `S2` codes the realization of stacks as (infinite) arrays with pointers to the top of the represented stack (arrays are coded here as functions from integer indices to values).

Now, these two realizations of stacks are behaviourally equivalent, since for each observable computation, like $top(pop(push(n, push(4, push(6, empty)))))$, they both deliver the same result (in this case 4). However, these algebras do not act the same way when non-observable computations are considered: for example, the computations *empty* and $pop(push(6, empty))$ yield the same result in `S1` but they yield different results in `S2`. □

Our earlier discussion would lead us to expect the class of models of a specification to be closed under behavioural equivalence. It is perhaps surprising that this is not easy to achieve directly: the class of models of a set of axioms typically does not have this property. Equational logic may be modified so as to force this to happen (cf. [NO 88]) and a similar idea for other logical systems is discussed in [BHW 94], but it is not clear how this approach can be extended to deal adequately with structured specifications. An alternative is to simply close the class of models of a specification under behavioural equivalence [SW 83, ST 87]. Any specification SP determines the class $[\![SP]\!] \subseteq Sig(SP)$ of models that "literally" satisfy the stated requirements, as discussed in Section 3; the ultimate semantics of SP is taken to be the closure of this under behavioural equivalence:

$$[\![\![SP]\!]\!] = \{A \mid A \equiv B \text{ for some } B \in [\![SP]\!]\}$$

In particular, $[\![\mathbf{sorts}\ S\ \mathbf{opns}\ \Omega\ \mathbf{axioms}\ \Phi]\!]$ contains exactly the $\langle S, \Omega\rangle$-algebras that satisfy the axioms Φ, while $[\![\![\mathbf{sorts}\ S\ \mathbf{opns}\ \Omega\ \mathbf{axioms}\ \Phi]\!]\!]$ contains also the algebras that do not satisfy Φ themselves but are behaviourally equivalent to algebras that do.

Example 4 (continued)

Both the list representation `S1` and the array-with-pointer representation `S2` of stacks are in $[\![\![STACK]\!]\!]$ — and this is what we meant when we declared

```
structure S1 : STACK = ...
structure S2 : STACK = ...
```

That is, $\mathtt{S1}, \mathtt{S2} \in [\![\![STACK]\!]\!]$. This holds even though the latter realization `S2` does not literally satisfy the axiom $\forall s{:}stack.\, \forall n{:}\mathtt{int}.\, pop(push(n, s)) = s$ and so is not in $[\![STACK]\!]$, $\mathtt{S2} \notin [\![STACK]\!]$. □

This approach typically gives extra expressive power: considering the institution of first-order logic with equality, there are classes of algebras that may be finitely characterised in this way, and which cannot be finitely axiomatised directly [Sch 92]. (Of course, this property depends on the logic considered: for example, second order logic allows one to specify behavioural closures directly [HS 95].) Also, *model-oriented specifications* [Jon 86] can be handled: if $[\![SP]\!]$ contains just a single algebra, then $[\![\![SP]\!]\!]$ admits any realization of the exhibited behaviour. In general, $[\![\![SP]\!]\!]$ contains all *reifications* of the algebras in $[\![SP]\!]$ (cf. [Hoa 72]).

The basic intuition for the use of behavioural equivalence in the development process is that it is not necessary to implement a specification SP according to its literal interpretation $[\![SP]\!]$; it is sufficient to implement it up to behavioural equivalence, as captured by its "ultimate" semantics $[\![\![SP]\!]\!]$. The definitions of refinement and constructor implementation are now as follows:

$$SP \rightsquigarrow SP' \quad \textbf{iff} \quad [\![\![SP']\!]\!] \subseteq [\![\![SP]\!]\!],$$

$$SP \underset{\kappa}{\rightsquigarrow} SP' \quad \textbf{iff} \quad [\![\kappa]\!]([\![\![SP']\!]\!]) \subseteq [\![\![SP]\!]\!].$$

Using these definitions, it is possible to develop programs from specifications by means of successive implementation steps exactly as described earlier. But there is a better way. The crucial novelty, due to [Sch 87], is that when *using* a realization of SP, it is convenient (and possible) to pretend that it satisfies the literal interpretation of SP.

Example 4 (continued)

Consider the following trivial specification

$$\begin{array}{lll} TRIV = & \mathbf{opns} & id : \mathtt{int} \times \mathtt{int} \times \mathtt{int} \to \mathtt{int} \\ & \mathbf{axioms} & \forall x, n, z{:}\mathtt{int}.\, n\mathtt{>=0} \Longrightarrow id(x, n, z) = x \end{array}$$

and its perhaps surprising realization in terms of stacks of integers:[4]

```
functor TR (S : STACK) : TRIV =
    struct
        fun multipush (n, z, s:S.stack) =
                if n <= 0 then s else S.push(z, multipush (n-1, z+1, s))
        fun multipop (n, s:S.stack) =
                if n <= 0 then s else multipop (n-1, S.pop(s))
        fun id (x,n,z) =
                S.top( multipop (n, multipush(n, z, S.push(x, S.empty))))
    end
```

Now, given any realization `S` of $STACK$, to verify that `TR(S)` $\in [\![\![TRIV]\!]\!]$, it is convenient to assume that the axiom $\forall s{:}stack.\forall n{:}\mathtt{int}.\, pop(push(n,s)) = s$ holds in `S` literally in spite of the fact that this equation is not valid in $[\![\![STACK]\!]\!]$. Under this assumption, a simple proof by induction (on the second argument of `id`) goes through. The reasoning for the induction step goes as follows:

$$
\begin{aligned}
&\mathtt{id}(x,n\mathtt{+1},z)\\
&\quad = \mathtt{S.top(multipop(}n\mathtt{+1,multipush(}n\mathtt{+1,}z\mathtt{,S.push(}x\mathtt{,S.empty))))}\\
&\quad = \mathtt{S.top(multipop(}n\mathtt{,S.pop(S.push(}z\mathtt{,multipush(}n\mathtt{,}z\mathtt{+1,S.push(}x\mathtt{,S.empty))))))}\\
&\quad = \mathtt{S.top(multipop(}n\mathtt{,multipush(}n\mathtt{,}z\mathtt{+1,S.push(}x\mathtt{,S.empty))))}\\
&\quad = \mathtt{id(}x\mathtt{,}n\mathtt{,}z\mathtt{+1)}\\
&\quad = x
\end{aligned}
$$

where the final step follows by the induction hypothesis. □

These considerations lead to the following definition of *behavioural implementation* [ST 88b]:

$$SP \underset{\kappa}{\overset{\equiv}{\rightsquigarrow}} SP' \quad \textbf{iff} \quad [\![\kappa]\!]([\![\![SP']\!]\!]) \subseteq [\![\![SP]\!]\!]$$

The alert reader will have noticed that there is a problem here: we want to have our cake and eat it. On one hand, we want to allow specifications to be implemented up to behavioural equivalence; on the other hand, we would like to use any realization as if it satisfied its specification literally. Behavioural implementations do not compose, and the following crucial property is lost:

$$\frac{SP \underset{\kappa}{\overset{\equiv}{\rightsquigarrow}} SP' \qquad A' \in [\![\![SP']\!]\!]}{[\![\kappa]\!](A') \in [\![\![SP]\!]\!]}$$

The behavioural implementation $SP \underset{\kappa}{\overset{\equiv}{\rightsquigarrow}} SP'$ ensures only that algebras in $[\![SP']\!]$ give rise to correct realizations of SP; this says nothing about the models in $[\![\![SP']\!]\!]$ that are not in $[\![SP']\!]$.

Example 4 (continued)

As stated above, $TRIV \underset{\mathtt{TR}}{\overset{\equiv}{\rightsquigarrow}} STACK$. Formally, neither the property this embodies (that `TR(S)` $\in [\![\![TRIV]\!]\!]$ for `S` $\in [\![STACK]\!]$) nor its suggested proof tell us anything about the application of `TR` to the algebra `S2` $\notin [\![STACK]\!]$, even though `S2` $\in [\![\![STACK]\!]\!]$ and we have earlier identified `S2` as an acceptable realization of $STACK$. It may be shown that

[4] This is of course an extremely contrived example, but it is easy to come up with realistic programs using stacks where properties like this are to be proved.

TR(S2) $\in [\![[TRIV]\!]]$ but the most obvious proof involves the non-elementary fact that for any natural number n,

$$\texttt{multipop}(n, s) = \underbrace{\texttt{S.pop}(\ldots(\texttt{S.pop}}_{n \text{ times}}(s))\ldots)$$

and similarly for multipush (and then relies on the property that since S2 $\in [\![[STACK]\!]]$, all observable computations in S2 yield the same results on s and on $pop(push(z, s))$, for any stack s and integer z).

Consider, however, another trivial realization of $TRIV$ in terms of $STACK$:

```
functor TR' (S : STACK) : TRIV =
    struct
        fun id (x,n,z) = let val se = S.pop( S.push (z, S.empty))
                         in if se = S.empty then x else z
                         end
    end
```

(TR' cannot be coded in SML since the type S.stack is not ensured to admit equality.)

We can prove now that $TRIV \underset{\texttt{TR'}}{\overset{\equiv}{\rightsquigarrow}} STACK$, hence TR'(S1) $\in [\![[TRIV]\!]]$, but of course TR'(S2) $\notin [\![[TRIV]\!]]$. □

It might seem that all is lost. But there is a way out, originally suggested in [Sch 87]. The above crucial property is recovered if we assume that the constructors used are *stable*, that is, that any constructor $[\![\kappa]\!] : Alg(Sig(SP')) \to Alg(Sig(SP))$ preserves behavioural equivalence:

$$\text{Stability assumption:} \qquad \text{if } A \equiv B \text{ then } [\![\kappa]\!](A) \equiv [\![\kappa]\!](B)$$

(the exact definition of stability of constructors in a formal development framework based on a full-blown programming language is somewhat more complex — see [Sch 87, ST 89]).

Under this assumption, the correctness of the individual implementation steps ensures the correctness of the result:

$$\frac{SP_0 \underset{\kappa_1}{\overset{\equiv}{\rightsquigarrow}} SP_1 \underset{\kappa_2}{\overset{\equiv}{\rightsquigarrow}} \cdots \underset{\kappa_n}{\overset{\equiv}{\rightsquigarrow}} SP_n = EMPTY}{[\![\kappa_1(\kappa_2(\ldots\kappa_n(empty)\ldots))]\!] \in [\![[SP_0]\!]]}$$

Example 4 (continued)

Clearly, the functor TR' as defined above is not stable.

On the other hand, the functor TR is stable (this can be proved along the lines of the argument given above to justify that TR(S2) $\in [\![[TRIV]\!]]$). This shows that TR(S) $\in [\![[TRIV]\!]]$ for all S $\in [\![[STACK]\!]]$, not only for S $\in [\![STACK]\!]$. □

We could repeat here the tree-like development picture of Section 7 — developments involving decomposition steps based on behavioural implementations with multi-argument (stable) constructors yield correct programs as well. We also recover vertical composability, under the assumption that parameterised programs compose as discussed in Section 7:

$$\frac{SP \underset{\kappa}{\overset{\equiv}{\rightsquigarrow}} SP' \qquad SP' \underset{\kappa'}{\overset{\equiv}{\rightsquigarrow}} SP''}{SP \underset{\kappa';\kappa}{\overset{\equiv}{\rightsquigarrow}} SP''}$$

The correctness of a behavioural implementation $SP \underset{\kappa}{\overset{\equiv}{\rightsquigarrow}} SP'$ is easier to verify than the correctness of the corresponding constructor implementation between the same specifications closed under behavioural equivalence: $[\![\kappa]\!]([\![[SP']\!]]) \subseteq [\![[SP]\!]]$ is a weaker condition than $[\![\kappa]\!]([\![\![SP']\!]\!]) \subseteq [\![\![SP]\!]\!]$ (but semantically these two conditions become equivalent if $[\![\kappa]\!]$ is stable). Also, the correctness of $SP \underset{\kappa}{\overset{\equiv}{\leadsto}} SP'$ is in general easier to verify than the correctness of the original constructor implementation $SP \underset{\kappa}{\leadsto} SP'$ (that is, $[\![\kappa]\!]([\![\![SP']\!]\!]) \subseteq [\![\![SP]\!]\!]$ is weaker than $\kappa([\![\![SP']\!]\!]) \subseteq [\![\![SP]\!]\!]$). For instance, viewing the structure S2 of Example 4 as a functor with an empty parameter, we have $STACK \underset{\mathtt{S2}}{\overset{\equiv}{\leadsto}} EMPTY$, while $STACK \underset{\mathtt{S2}}{\not\leadsto} EMPTY$. As we have argued, this extra flexibility reflects our intuitive understanding of what it means for an algebra to realize a specification.

We are still left with the need to establish the stability of constructors, and so one may wonder if it is worthwhile taking advantage of this property. However, recall that constructors are determined by parameterised programs, and these must be expressed in some particular programming language. Thus stability can be checked in advance, for the programming language as a whole (this is simplified somewhat by the fact that the composition of stable constructors is stable) and this frees the programmer from the need to prove it during the program development process.

There is a close connection between the requirement of stability and the security of encapsulation mechanisms in programming languages supporting abstract data types. A programming language ensures stability if the only way to access an encapsulated data type is via the operations explicitly provided in its output interface. This suggests that stability of constructors is an appropriate thing to expect; following [Sch 87] we view the stability requirement as a methodologically justified design criterion for the modularisation facilities of programming languages.

Example 5

The limited notation of Standard ML functors we have used in Examples 1 and 4 throughout this paper ensures stability of constructors.[5] Intuitively, this is because the parameter signature of any SML functor provides sufficient insulation between the functor body and the actual parameters. Within the functor body we can access the parameter only using the "tools" given in the parameter signature. Unfortunately, signatures given for functor results and for structures declared in SML are much more "transparent": they do not provide sufficient insulation between the declaration of a structure and its use. For example, the following code in SML shows a non-stable extension of the structure S2:

```
structure S2 : STACK =
    struct
        type stack = (int -> int) * int
        val empty = ...
        ...
        fun pop (f,i) = if i = 0 then (f,0) else (f,i-1)
        ...
    end
fun id (x,n,z) = let val (f,i) = S2.pop( S2.push (z, S2.empty))
                 in if f(i+1) = 0 then x else z
```

Changing the "internal" implementation details of S2 for example as follows:

```
structure S2 : STACK =
    struct
```

[5]Of course, considerable work would be required to turn this claim into a formal theorem with a precise proof.

```
            type stack = (int -> int) * int
            val empty = ...
            ...
            fun pop (f, i) = if i = 0 then (f,0) else
                                 (fn k => if k = i then 0 else f k, i-1)
            ...
        end
```

changes completely the behaviour of the `id` function as defined in the extension, even though the new realization `S2` of stacks is behaviourally equivalent to the previous one.

This can be viewed as a deficiency in the design of the Standard ML modularisation facilities. This infelicity is not present in the Extended ML formalism [ST 91, KST 94a, KST 94b] where access to a structure or functor result is limited to the use of the tools given in its signature. □

10 Conclusion

We have outlined the main ideas of a framework to support the formal development of correct programs from specifications of their required behaviour. Our purpose has not been to introduce new technicalities, but rather to explain in a careful way the general ideas underlying the algebraic approach and the specific motivation behind the concepts involved in the formalization of the development process. This forced us to clarify some of the finer points of the approach, like the distinction between syntax and semantics in constructor implementations and an abstract formulation of stability in this context.

The main challenge now is to put these ideas into practice in the formal development of non-trivial programs in real programming languages. We are moving in this direction with our work on the Extended ML framework for the formal development of modular Standard ML programs [ST 91, KST 94a, KST 94b], though more effort is required. Subjecting foundational work to the test of practice is sure to bring fascinating new problems and issues to light.

Acknowledgements: Most of the above ideas have been presented elsewhere, and have been discussed with and influenced by many of our colleagues. Thanks especially to Rod Burstall, Jordi Farrés, Joseph Goguen, Fernando Orejas, Oliver Schoett and Martin Wirsing. This research was supported by the EC-funded COMPASS Basic Research working group and MeDiCiS Scientific Cooperation Network (DS, AT), by EPSRC grants GR/H73103 and GR/J07303 and an EPSRC Advanced Fellowship (DS), and by KBN grant 2 P301 007 04 (AT).

References

[Bar 74] J. Barwise. Axioms for abstract model theory. *Ann. Math. Logic* 7:221–265 (1974).

[BF 85] J. Barwise and S. Feferman (eds.) *Model-Theoretic Logics.* Springer (1985).

[BW 82] F. Bauer and H. Wössner. *Algorithmic Language and Program Development.* Springer (1982).

[BV 87] C. Beierle and A. Voß. Viewing implementations as an institution. *Proc. 1987 Conference on Category Theory and Computer Science*, Edinburgh. Springer LNCS 283, 196–218 (1987).

[BHW 94] M. Bidoit, R. Hennicker and M. Wirsing. Characterizing behavioural semantics and abstractor semantics. *Proc. 5th European Symp. on Programming*, Edinburgh. Springer LNCS 788, 105–119 (1994).

[BKLOS 91] M. Bidoit, H.-J. Kreowski, P. Lescanne, F. Orejas and D. Sannella (eds.) *Algebraic System Specification and Development: A Survey and Annotated Bibliography.* Springer LNCS 501 (1991).

[BG 77] R. Burstall and J. Goguen. Putting theories together to make specifications. *Proc. 5th Intl. Joint Conf. on Artificial Intelligence*, Cambridge, Massachusetts, 1045–1058 (1977).

[BG 80] R. Burstall and J. Goguen. The semantics of Clear, a specification language. *Proc. 1979 Copenhagen Winter School on Abstract Software Specification.* Springer LNCS 86, 292–332 (1980).

[Dav 90] A. Davis. *Software Requirements: Analysis and Specification.* Prentice Hall (1990).

[DH 84] R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science* 34:83–133 (1984).

[DGS 93] R. Diaconescu, J. Goguen and P. Stefaneas. Logical support for modularization. In: *Logical Environments* (G. Huet and G. Plotkin, eds.). Cambridge Univ. Press, 83–130 (1993).

[EBO 93] H. Ehrig, M. Baldamus and F. Orejas. New concepts of amalgamation and extension of a general theory of specifications. *Selected Papers from the 8th Workshop on Specification of Abstract Data Types*, Dourdan. Springer LNCS 655, 199–221 (1993).

[EKMP 82] H. Ehrig, H.-J. Kreowski, B. Mahr and P. Padawitz. Algebraic implementation of abstract data types. *Theoretical Computer Science* 20:209–263 (1982).

[EM 85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification I: Equations and Initial Semantics.* Springer (1985).

[Far 92] J. Farrés-Casals. Verification in ASL and Related Specification Languages. Ph.D. thesis, report CST-92-92, Dept. of Computer Science, Univ. of Edinburgh (1992).

[FS 88] J. Fiadeiro and A. Sernadas. Structuring theories on consequence. *Selected Papers from the 5th Workshop on Specification of Abstract Data Types*, Gullane. Springer LNCS 332, 44–72 (1988).

[FJ 90] J. Fitzgerald and C. Jones. Modularizing the formal description of a database system. *Proc. VDM'90 Conference*, Kiel. Springer LNCS 428 (1990).

[Gau 84] M.-C. Gaudel. A first introduction to PLUSS. Technical report, LRI, Université de Paris-Sud, Orsay (1984).

[Gog 84] J. Goguen. Parameterized programming. *IEEE Trans. on Software Engineering* SE-10(5):528–543 (1984).

[GB 80] J. Goguen and R. Burstall. CAT, a system for the structured elaboration of correct programs from structured specifications. Technical report CSL-118, SRI International (1980).

[GB 84] J. Goguen and R. Burstall. Introducing institutions. *Proc. Logics of Programming Workshop*, Carnegie-Mellon. Springer LNCS 164, 221–256 (1984).

[GB 92] J. Goguen and R. Burstall. Institutions: abstract model theory for specification and programming. *Journal of the Assoc. for Computing Machinery* 39:95–146 (1992).

[GM 82] J. Goguen and J. Meseguer. Universal realization, persistent interconnection and implementation of abstract modules. *Proc. Intl. Colloq. on Automata, Languages and Programming*, Aarhus. Springer LNCS 140, 265–281 (1982).

[GM 85] J. Goguen and J. Meseguer. Completeness of many-sorted equational logic. *Houston Journal of Mathematics* 11(3):307–334 (1985).

[GWMFJ 92] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi and J.-P. Jouannaud. Introducing OBJ3. Techical Report SRI-CSL-92-03, SRI International (1992).

[GH 80] J. Guttag and J. Horning. Formal specification as a design tool. *Proc. ACM Symp. on Principles of Programming Languages*, Las Vegas, 251–261 (1980).

[GH 93] J. Guttag and J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer (1993).

[GHW 82] J. Guttag, J. Horning and J. Wing. Some notes on putting formal specifications to productive use. *Science of Computer Programming* 2:53–68 (1982).

[HST 94] R. Harper, D. Sannella and A. Tarlecki. Structured theory presentations and logic representations. *Annals of Pure and Applied Logic* 67:113–160 (1994).

[HJ 89] I.J. Hayes and C.B. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal* 4(6):320–338 (1989).

[Hoa 72] C.A.R. Hoare. Proofs of correctness of data representations. *Acta Informatica* 1:271–281 (1972).

[HS 95] M. Hofmann and D. Sannella. On behavioural abstraction and behavioural satisfaction in higher-order logic. *Proc. 6th Joint Conf. on Theory and Practice of Software Development*, Aarhus. Springer LNCS, to appear (1995).

[Huß 85] H. Hußmann. Rapid prototyping for algebraic specifications: RAP system user's manual. Report MIP-8504, Universität Passau (1985).

[Jon 86] C. Jones. *Systematic Software Development using VDM.* Prentice Hall (1986).

[KST 94a] S. Kahrs, D. Sannella and A. Tarlecki. The semantics of Extended ML: a gentle introduction. *Proc. Intl. Workshop on Semantics of Specification Languages*, Utrecht, 1993. Springer Workshops in Computing, 186–215 (1994).

[KST 94b] S. Kahrs, D. Sannella and A. Tarlecki. The definition of Extended ML. Report ECS-LFCS-94-300, Dept. of Computer Science, Univ. of Edinburgh (1994).

[Kaz 92] E. Kazmierczak. Modularizing the specification of a small database system in Extended ML. *Formal Aspects of Computer Science* 4:100-142 (1992).

[Li 94] W. Li. A logical framework for evolution of specifications. *Proc. 5th European Symp. on Programming*, Edinburgh. Springer LNCS 788, 394–408 (1994).

[MT 94] D. MacQueen and M. Tofte. A semantics for higher-order functors. *Proc. 5th European Symp. on Programming*, Edinburgh. Springer LNCS 788, 409–423 (1994).

[MTH 90] R. Milner, M. Tofte and R. Harper. *The Definition of Standard ML.* MIT Press (1990).

[MA 91] J. Morris and S. Ahmed. Designing and refining specifications with modules. *Proc. 3rd Refinement Workshop*, Hursley Park, 1990. Springer Workshops in Computing, 73–95 (1991).

[NO 88] P. Nivela and F. Orejas. Initial behaviour semantics for algebraic specifications. *Selected Papers from the 5th Workshop on Specification of Abstract Data Types*, Gullane. Springer LNCS 332, 184–207 (1988).

[Ore 83] F. Orejas. Characterizing composability of abstract implementations. *Proc. 1983 Intl. Conf. on Foundations of Computation Theory*, Borgholm. Springer LNCS 158, 335–346 (1983).

[Pau91] L. Paulson. *ML for the Working Programmer.* Cambridge Univ. Press (1991).

[Rei 81] H. Reichel. Behavioural equivalence — a unifying concept for initial and final specification methods. *Proc. 3rd Hungarian Computer Science Conference*, 27–39 (1981).

[RAHL 94] D. Robertson, J. Agusti, J. Hesketh and J. Levy. Expressing program requirements using refinement lattices. *Fundamenta Informaticae* 21:163–183 (1994).

[San 91] D. Sannella. Formal program development in Extended ML for the working programmer. *Proc. 3rd BCS/FACS Workshop on Refinement*, Hursley Park. Springer Workshops in Computing, 99–130 (1991).

[SB 83] D. Sannella and R. Burstall. Structured theories in LCF. *Proc. Colloq. on Trees in Algebra and Programming*, L'Aquila. Springer LNCS 159, 377–391 (1983).

[SST 92] D. Sannella, S. Sokołowski and A. Tarlecki. Toward formal development of programs from algebraic specifications: parameterisation revisited. *Acta Informatica* 29:689–736 (1992).

[ST 86] D. Sannella and A. Tarlecki. Extended ML: an institution-independent framework for formal program development. *Proc. Intl. Workshop on Category Theory and Computer Programming*, Guildford. Springer LNCS 240, 364–389 (1986).

[ST 87] D. Sannella and A. Tarlecki. On observational equivalence and algebraic specification. *J. of Computer and System Sciences* 34:150–178 (1987).

[ST 88a] D. Sannella and A. Tarlecki. Specifications in an arbitrary institution. *Information and Computation* 76:165–210 (1988).

[ST 88b] D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: implementations revisited. *Acta Informatica* 25:233–281 (1988).

[ST 89] D. Sannella and A. Tarlecki. Toward formal development of ML programs: foundations and methodology. *Proc. 3rd Joint Conf. on Theory and Practice of Software Development*, Barcelona. Springer LNCS 352, 375–389 (1989).

[ST 91] D. Sannella and A. Tarlecki. Extended ML: past, present and future. *Proc. 7th Intl. Workshop on Specification of Abstract Data Types*, Wusterhausen. Springer LNCS 534, 297–322 (1991).

[ST 92] D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: model-theoretic foundations. *Proc. Intl. Colloq. on Automata, Languages and Programming*, Vienna. Springer LNCS 623, 656–671 (1992).

[ST 95] D. Sannella and A. Tarlecki. *Foundations of Algebraic Specifications and Formal Program Development.* Cambridge Univ. Press, to appear (1995).

[SW 82] D. Sannella and M. Wirsing. Implementation of parameterised specifications. *Proc. Intl. Colloq. on Automata, Languages and Programming*, Aarhus. Springer LNCS 140, 473–488 (1982).

[SW 83] D. Sannella and M. Wirsing. A kernel language for algebraic specification and implementation. *Proc. 1983 Intl. Conf. on Foundations of Computation Theory*, Borgholm. Springer LNCS 158, 413–427 (1983).

[Sch 87] O. Schoett. Data Abstraction and the Correctness of Modular Programming. Ph.D. thesis, report CST-42-87, Dept. of Computer Science, Univ. of Edinburgh (1987).

[Sch 92] O. Schoett. Two impossibility theorems on behavioural specification of abstract data types. *Acta Informatica* 29:595–621 (1992).

[Tof 92] M. Tofte. Principle signatures for higher-order program modules. *Proc. 19th ACM Symp. on Principles of Programming Languages*, 189–199 (1992).

[Wir 86] M. Wirsing. Structured algebraic specifications: a kernel language. *Theoretical Computer Science* 42:123–249 (1986).

[Wir 93] M. Wirsing. Structured specifications: syntax, semantics and proof calculus. *Logic and Algebra of Specification* (F. Bauer, W. Brauer and H. Schwichtenberg, eds.). Springer, 411–442 (1993).