Math. Struct. in Comp. Science (1993), *vol.* 3, *pp.* 333–363 Copyright © 1993 Cambridge University Press

Program specification and data refinement in type theory[†]

ZHAOHUI LUO

Department of Computer Science, University of Edinburgh, Mayfield Road, Edinburgh EH9 3JZ

Received 14 June 1991; revised 30 January 1993

The study of type theory may offer a uniform language for modular programming, structured specification and logical reasoning. We develop an approach to program specification and data refinement in a type theory with a strong logical power and nice structural mechanisms to show that it provides an adequate formalism for modular development of programs and specifications. Specification of abstract data types is considered, and a notion of abstract implementation between specifications is defined in the type theory and studied as a basis for correct and modular development of programs by stepwise refinement. The higher-order structural mechanisms in the type theory provide useful and flexible tools (specification operations and parameterized specifications) for modular design and structured specification. Refinement maps (programs and design decisions) and proofs of implementation correctness can be developed by means of the existing proof development systems based on type theories.

1. Introduction

Program specification and modular program development by stepwise refinement has been an interesting research area in computer science (see, for example, Hoare (1972), Liskov and Zilles (1975), Guttag *et al.* (1976), Goguen *et al.* (1978), Burstall and Goguen (1980), Ehrig *et al.* (1983), Ehrig and Mahr (1985 and 1990), Futatsugi *et al.* (1985), Wirsing (1986), Jones (1986), Sanella and Tarlecki (1987 and 1988a), and Wirsing and Broy (1989) among the enormous literature). Various formal abstraction mechanisms (*e.g.*, algebraic specifications) have been studied to provide good methodologies and tools that can be used to apply useful principles for software development (such as the separation of concerns and divide-and-conquer), and to guarantee the correctness of programs with respect to their specifications. These investigations have also contributed to some extent to the development of programming languages.

Type theories (*e.g.*, Martin-Löf's type theory (Martin-Löf 1975 and 1984), the Automath type theory (de Bruijn 1980), Nuprl's type theory (Constable *et al.* 1986), and Coquand and Huet's calculus of constructions (Coquand and Huet 1988) have been developed mainly for the foundation and formalization of mathematics. Since the work by Martin-Löf, it has

[†] An earlier version of this paper appears in Proc. of TAPSOFT'91 (Luo 1991b).

become known that type theories can also provide basic mechanisms for programming and program specification (Martin-Löf 1982; Nordström *et al.* 1990). For instance, program derivation has been studied in various type theories (Nordström *et al.* 1990; Backhouse *et al.* 1989; Paulin-Mohring 1989). However, although it is known that a type theory can be used as a programming and specification language, some important topics concerning modular design and structured specification (*e.g.*, abstract implementation and modular refinement) have not been paid enough attention in type-theoretic settings, and the potential of type theory has not been well developed in this area.

It is expected that the study of type theory may offer an *adequate* computational and logical language for computer science. There are several compelling reasons supporting such a claim of adequacy. First, type theory offers a coherent treatment of two related but different fundamental notions in computer science: computation and logical inference. This makes it possible to program and understand programs, either operationally or by logical reasoning, in a single formalism. Second, type theory can provide nice abstraction mechanisms that support a conceptually clear (*e.g.*, modular and structured) development of programs, specifications and proofs. This makes it a promising candidate for a uniform language for programming, specification and reasoning in the large, as well as in the small. Finally, although type theory provides powerful and sophisticated tools, it is simple. There are two aspects to this simplicity: one is that it allows a direct understanding of the meanings of the constructions in the language, which gives a solid basis for its use in applications; the other is that it is manageable, in the sense that there is a good way to implement it on computers.

It is not our task in this paper to discuss all of the above aspects. (A more thorough discussion can be found in Luo (1993).) Our aim in this paper is to develop a type-theoretic approach to program specification and data refinement, and to show that a type theory with nice structural mechanisms provides an adequate language for both modular design by data refinement and structured specification of (functional) programs in the type theory. This, in particular, addresses the second aspect of the use of type theory mentioned above, and is a part of our work aimed at developing type theory as a rich and uniform language for programming, logical reasoning, structured specification and modular program development.

The type theory that we work with in this paper is the Extended Calculus of Constructions (**ECC**) (Luo 1989, 1990a and 1993), which is developed as an initial step towards a rich computational language with a powerful internal logic for modular development of programs, specifications and proofs. As a formal system, **ECC** extends the calculus of constructions (Coquand and Huet 1988) with predicative type universes and Σ-types (strong sum); it may also be seen as an extension of Martin-Löf's type theory with universes (Martin-Löf 1975) by an impredicative universe (higher-order logic). However, unlike Martin-Löf's type theory and the calculus of constructions, the incorporation of both an impredicative universe and predicative universes enhances a conceptual distinction between the notion of logical formulae (propositions) and that of sets (data types); this basic idea leads to a unifying theory of dependent types, which provides not only strong logical power but also adequate abstraction mechanisms for pragmatic applications. One of the pragmatic motivations for the development of the theory **ECC** was to consider applications to program specification and abstract reasoning. The higher-order structural and logical mechanisms (Σ-types and type universes, in particular) prove to be very useful for abstract reasoning (Luo 1991a) and program specification, the latter being discussed in this paper (also see Burstall and McKinna (1991) and McKinna (1992)). It is our hope that such an investigation of program specification and data refinement in type theory will enhance the use of type theory as a uniform language for programming, specification and reasoning.

A *specification* in the type theory consists of (a pair of) a type, whose objects are the possible structures (programs and program modules) that may realize the specification, and a predicate over the structure type, which specifies the properties that any realization should satisfy. In particular, the structure type of a specification of an abstract data type (say of stacks) can be defined as a Σ-type, each of whose objects has as its components a type (of stacks) associated with an explicit congruence relation (between stacks) and certain operations (corresponding to the empty stack, push operation, *etc.*); the predicate over the structure type would specify the required properties, including that the associated binary relation (between stacks) is a congruence. (Using an explicit congruence rather than a built-in equality is both adequate concerning the semantics and important for stepwise abstract refinement. See Section 3.) The semantics is straightforward and 'model-theoretic' in the sense that a *realization* of a specification is simply a structure (an object of the structure type) satisfying the required properties.

In order to discuss program development by stepwise refinement, we formalize a notion of *abstract implementation* between specifications, which is similar to the notion of theory morphism for abstract reasoning (Luo *et al.* 1989; Luo 1991a) and the notion of 'deliverables' by Burstall and McKinna (Burstall and McKinna 1991; McKinna 1992). A specification SP *refines to* (or *is implemented by*) another specification SP' through a refinement map ρ (a function from the structure type of SP' to that of SP) if the images of ρ over the realizations of SP' are realizations of SP. In such a case, the refinement map is an incomplete program expressing the design decisions made in the refinement step. This implementation relation composes vertically (*cf.*, Burstall and Goguen (1980)) and hence satisfies the basic requirement for stepwise development of programs.

Using the notion of abstract implementation, we further discuss methodological issues in software development and show that the higher-order structural mechanisms in the type theory nicely support modular design and structured specification. Σ-types support *decomposition* of specifications into independent specifications (which may possibly share some common parts). We also identify two general classes of specification operations, called *constructors* and *selectors*, which are monotone with respect to the implementation relation and can be used both in structured design by modular refinement and in structuring requirements specifications.

The higher-order facilities in the type theory naturally support *parameterized specifications*. A notion of implementation between parameterized specifications is defined and is shown to compose vertically. When a parameterized specification is monotone with respect to the implementation relation between specifications, the property of horizontal composition (*cf.*, Burstall and Goguen (1980)) holds, and the design principle of divide-and-conquer can also be applied.

The type-theoretic approach to specification and data refinement is simple, and the higher-order mechanisms in the type theory provide powerful and useful supports in various aspects of modular development of programs and specifications. Note that the type theory provides a single formal system in which programs, specifications and their implementation relationships can be uniformly formalized and discussed. Such an 'internalization' has the immediate benefit that refinement maps (programs and design decisions) as well as proofs of implementation correctness can be developed (interactively) in a proof development system such as Lego (Pollack 1989; Luo and Pollack 1992), in which the type theory **ECC** is implemented. (In fact, all of the examples and propositions in this paper have been checked in the Lego system.) We shall relate and compare our type-theoretic approach to that of algebraic specifications, in particular by relating the notion of implementation to that of constructor implementation developed by Sannella and Tarlecki (Sannella and Tarlecki 1988b) and discussing the differences between the two approaches.

In Section 2, we give a brief introduction to the type theory used in this paper. Section 3 discusses specifications, specification of abstract data types, and the notion of abstract implementation. The issues of modular design and specification operations are dealt with in Section 4, and parameterized specifications and their implementations are discussed in Section 5. Some discussion of the type-theoretic approach in comparison with other approaches to specifications and further research topics is givèn in the conclusion.

2. The Extended Calculus of Constructions

The type theory **ECC** (Luo 1989 and 1990a) is a natural combination of Martin-Löf's type theory (Martin-Löf 1975) and the calculus of constructions (Coquand and Huet 1988), developed on the idea that there should be a clear distinction between the notions of data types and logical propositions. The conceptual universe of types for the type theory is shown in Figure 1. The type system has good proof-theoretic properties (Church-Rosser, strong normalization, decidability and others, see Luo (1989 and 1990a) for details), which provide the proof-theoretic basis for the operational meaning theory of the type theory and for its computer implementation. A set-theoretic (realizability) model can be found in Luo (1991a). **ECC** is implemented in the proof development system Lego (Pollack 1989; Luo and Pollack 1992), which supports resolution-style interactive proof development. In this section, we give a brief and informal explanation of the type theory and introduce some notational conventions used in this paper. To make the paper more accessible to non-experts in type theory, some informal explanations of basic concepts in type theory are also included.

We start by introducing the basic concepts and the general rules. A *context* is a list of assumptions written in the form $x_1{:}A_1, \ldots, x_n{:}A_n$ $(n \geq 0)$, whose validity (well-formedness) is asserted by judgements of the form 'Γ **valid**', as given by the following rules:

$$\frac{}{\langle\rangle\ \textbf{valid}} \qquad \frac{\Gamma \vdash A : Type_i \quad x \notin FV(\Gamma)}{\Gamma, x{:}A\ \textbf{valid}},$$

where $\langle\rangle$ is the empty context and $FV(\Gamma)$ is the set of free variables occurring in context Γ.

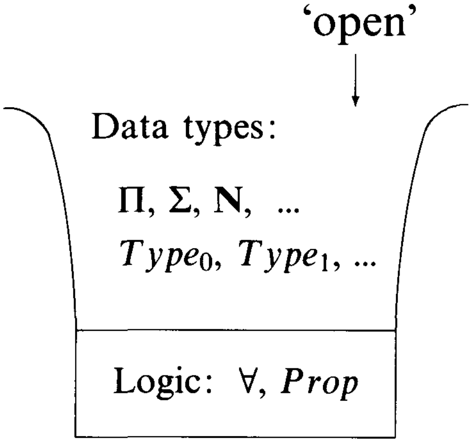

Fig. 1. The conceptual universe of types.

The most important form of judgements in type theory is $\Gamma \vdash a : A$ (or simply $a : A$ when the context is clear or irrelevant[†]), which asserts that *object a is of type A (in context* Γ*)*. For example, having assumed $x{:}A$ in a valid context, we can derive that x is an object of type A in the context, as expressed by the following rule:

$$\frac{\Gamma, x{:}A, \Gamma' \ \textbf{valid}}{\Gamma, x{:}A, \Gamma' \vdash x : A} .$$

A *universe* is a type with (names of) types as its objects. Conversely, the *types* are the objects with universes as their types. The universes $Prop$ and $Type_i$ ($i \in \omega$) are introduced by the following rules:

$$\frac{\Gamma \ \textbf{valid}}{\Gamma \vdash Prop : Type_0} \qquad \frac{\Gamma \ \textbf{valid}}{\Gamma \vdash Type_i : Type_{i+1}} .$$

That is, we have (intuitively): $Prop \in Type_0 \in Type_1 \in \ldots$. Furthermore, any object of type $Prop$ is an object of $Type_0$ and any object of type $Type_i$ is an object of $Type_{i+1}$; namely, $Prop \subseteq Type_0 \subseteq Type_1 \subseteq \ldots$. This type inclusion between universes, together with the computational equality $=$ (conversion, see below), generates a subtyping relation $\preceq$ between the types,[‡] which is formally expressed by the following rule:

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash B : Type_i}{\Gamma \vdash a : B} \quad (A \preceq B) .$$

As a special case, the above rule implies that two computationally equal types have the

[†] In our discussion of specifications and program development in this paper, the reader may always assume that we are working in the empty context, unless we have made the context explicit.

[‡] Since this is not important for an understanding of the rest of this paper, we have intentionally omitted the technical definition of the subtyping relation. See Luo (1989 and 1990a) for details.

same objects. The universes in the type theory give a strong notion of type polymorphism and, together with the other type constructors, provide nice structural mechanisms.

Dependent product types (Π-types) in the theory provide both dependent function spaces and the logical formulae of the internal logic. The formation rules for Π-types are

$$\frac{\Gamma, x{:}A \vdash B : Prop}{\Gamma \vdash \Pi x{:}A.B : Prop} \qquad \frac{\Gamma \vdash A : Type_i \quad \Gamma, x{:}A \vdash B : Type_i}{\Gamma \vdash \Pi x{:}A.B : Type_i}\ .$$

They have the following introduction and elimination rules

$$\frac{\Gamma, x{:}A \vdash b : B}{\Gamma \vdash \lambda x{:}A.b : \Pi x{:}A.B} \qquad \frac{\Gamma \vdash f : \Pi x{:}A.B \quad \Gamma \vdash a : A}{\Gamma \vdash f(a) : [a/x]B}$$

(where $[a/x]$ is the usual substitution operator), and the computation rule (β-conversion)

$$(\lambda x{:}A.b)(a) = [a/x]b,$$

which gives meaning to the application operator. Intuitively, $\Pi x{:}A.B[x]$ represents the set of (dependent) functions or functional programs from A to $B[x]$:

$$\{f \mid f(a) \in B[a] \ \ for\ all\ a \in A\}.$$

Notation We may write $\Pi x{:}A.B$ as $A \to B$, when B is not dependent on x ($x \notin FV(B)$). We also often write $f(a_1, ..., a_n)$ for $f(a_1)...(a_n)$. If $f : A \to B$ and $g : B \to C$, we use $g \circ f =_{\mathrm{df}} \lambda x{:}A.g(f(x))$ to denote the functional composition of f and g.

Note that the universes $Type_i$ are *predicatively* closed over products. For example, $\Pi X{:}Type_i.X$ is of type $Type_{i+1}$ but not of type $Type_i$. In contrast, the universe *Prop* is *impredicative* in the sense that it is closed for arbitrary products. For example, $\Pi X{:}Prop.X$ is of type *Prop*. By the principle of propositions-as-types (Curry and Feys 1958; Howard 1980), there is an internal (intuitionistic) higher-order logic in the type theory, whose formulas are the objects of type *Prop*, called *propositions*. A proposition P is *provable* if there is an object of type P. The logical constants and operators can be defined as shown in Figure 2, where A is an arbitrary type, P_1 and P_2 are arbitrary propositions and $P[x]$ is an arbitrary family of propositions indexed by objects of type A.

Unlike Martin-Löf's type theory (Martin-Löf 1973 and 1984), in **ECC**, types and propositions are *not* identified and there is a distinguishable internal logic. Every proposition is (lifted as) a type, but *not* vice versa. This gives a conceptual distinction between data types, which reside in the predicative universes, and (logical) propositions, which reside in the impredicative universe.[†] In our language, the logical propositions constitute a totality – the universe *Prop*. Likewise, there are *internal* notions of *predicates* and *relations* that form totalities represented by types of propositional functions. For example, the types of the (internal) predicates and binary relations over type A are

$$A \to Prop \quad and \quad A \to A \to Prop,$$

respectively. (In contrast, for a predicative type theory where there is no totality of

[†] We can give a slightly different formulation of **ECC** by distinguishing the types with their names, which will make this point clearer. This is out of the range of this paper. See, for example, Luo (1992).

$\forall x{:}A.P[x]$	$=_{\mathrm{df}}$	$\Pi x{:}A.P[x]$
$P_1 \supset P_2$	$=_{\mathrm{df}}$	$\forall x{:}P_1.P_2$
true	$=_{\mathrm{df}}$	$\forall X{:}Prop.\ X \supset X$
false	$=_{\mathrm{df}}$	$\forall X{:}Prop.X$
$P_1\ \&\ P_2$	$=_{\mathrm{df}}$	$\forall X{:}Prop.\ (P_1 \supset P_2 \supset X) \supset X$
$P_1 \vee P_2$	$=_{\mathrm{df}}$	$\forall X{:}Prop.\ (P_1 \supset X) \supset (P_2 \supset X) \supset X$
$\neg P_1$	$=_{\mathrm{df}}$	$P_1 \supset$ **false**
$\exists x{:}A.P[x]$	$=_{\mathrm{df}}$	$\forall X{:}Prop.\ (\forall x{:}A.(P[x] \supset X)) \supset X$
$a =_A b$	$=_{\mathrm{df}}$	$\forall P{:}A \to Prop.\ P(a) \supset P(b)$

Fig. 2. Definitions of logical operators in **ECC**

propositions, the notion of predicate can only be properly considered at the meta level outside the type theory.) With the internal notion of predicate, it is possible to define a propositional equality (so-called *Leibniz's equality*, $=_A$, as in Figure 2), which says that two objects of the same type are equal if and only if they cannot be distinguished by any predicate over their type. An important property of the Leibniz equality is that it reflects the basic computational equality (conversion), in the sense that *two closed objects are Leibniz equal if and only if they are computationally equal*, that is, if $\vdash a : A$ and $\vdash b : A$, then

$$a = b \quad \textit{if and only if} \quad (a =_A b) \textit{ is provable.}$$

This *equality reflection* result, a proof of which can be found in Luo (1990a), shows that the Leibniz equality can be used to give adequate specifications of programs for concrete data types (see Section 3.1). Finally, we remark that the normalization theorem of the type system entails the consistency of the internal logic (Luo 1990a); furthermore, as discussed in Luo (1990b), the internal logic adequately reflects the intuitionistic higher-order predicate logic (Church 1940) in the type-theoretic setting.

A strong sum type (Σ-type) $\Sigma x{:}A.B[x]$ intuitively represents the set of pairs (a, b), where a is an object of type A and b an object of type $B[a]$:

$$\{\,(a,b) \mid a \in A \textit{ and } b \in B[a]\,\}.$$

The predicative universes are (predicatively) closed for strong sum:

$$\frac{\Gamma \vdash A : Type_i \quad \Gamma, x{:}A \vdash B : Type_i}{\Gamma \vdash \Sigma x{:}A.B : Type_i}\ .$$

Σ-types in **ECC** have the following introduction and elimination rules

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : [a/x]B \quad \Gamma, x{:}A \vdash B : Type_i}{\Gamma \vdash \mathbf{pair}_{\Sigma x:A.B}(a,b) : \Sigma x{:}A.B}$$

$$\frac{\Gamma \vdash c : \Sigma x{:}A.B}{\Gamma \vdash \pi_1(c) : A} \qquad \frac{\Gamma \vdash c : \Sigma x{:}A.B}{\Gamma \vdash \pi_2(c) : [\pi_1(c)/x]B}\ ,$$

and the computation rules

$$\pi_i(\mathbf{pair}_A(a_1,a_2)) = a_i \quad (i = 1,2),$$

which give meanings to the projection operators. The basic constructions that Σ-types provide are tuples, which can be analyzed by extracting their components. Based on this functionality, Σ-types provide a nice structuring mechanism in various pragmatic applications. In the presence of other type constructors (in particular, type universes), Σ-types can be used as types of program modules or abstract structures (see MacQueen (1986) and Luo (1991a) and Section 3.2).

Notation We may write $\Sigma x{:}A.B$ as $A \times B$, when B is not dependent on x ($x \notin FV(B)$). Although pairs in the type system are heavily typed (for decidability reasons), we often use (a,b) rather than $\mathbf{pair}_A(a,b)$ to denote pairs when no confusion may occur. In this paper, we shall also use the following notational convention:

$$\sum [x_1{:}A_1,\ x_2{:}A_2,\ ...,\ x_n{:}A_n]$$

will denote the Σ-type

$$\Sigma x_1{:}A_1 \Sigma x_2{:}A_2 ... \Sigma x_{n-1}{:}A_{n-1}.A_n,$$

and for any object a of such a type, we use $x_i[a]$ as (the name of) the obvious projections over object a, *e.g.*, $x_2[a]$ stands for $\pi_1(\pi_2(a))$ and $x_n[a]$ for $\pi_2(...\pi_2(\pi_2(a))...)$ with π_2 occuring $n-1$ times.

The predicative universes $Type_i$ are viewed as universes of data types. (This is in contrast with the view of coding of data types (Bohm and Berarducci 1985) in an impredicative type system like Girard–Reynold's polymorphic λ-calculus (Girard 1972; Reynolds 1974) or the calculus of constructions (Coquand and Huet 1988).) These predicative universes are supposed to be *open* in the same sense as Martin-Löf explains for his type theory (Martin-Löf 1973 and 1984). In particular, various inductive data types can be added to the predicative universes. In the context of programming and program specification, these inductive types are regarded as *concrete* data types of the type theory being viewed as a programming language (see Section 3.2 for a discussion of the difference between concrete and abstract data types). For example, the type of natural numbers can be introduced by adding constants

$$\mathbf{N} : Type_0, \quad 0 : \mathbf{N}, \quad \mathbf{succ} : \mathbf{N} \to \mathbf{N},$$

and a recursion operator $\mathbf{rec_N}$ with the elimination rule

$$\frac{\Gamma \vdash C : \mathbf{N} \to Type_i \quad \Gamma \vdash c : C(0) \quad \Gamma \vdash f : \Pi x{:}\mathbf{N}.\ C(x) \to C(\mathbf{succ}(x))}{\Gamma \vdash \mathbf{rec_N}(c,f) : \Pi x{:}\mathbf{N}.C(x)},$$

and the computation rules

$$\mathbf{rec_N}(c,f)(0) = c \quad \text{and} \quad \mathbf{rec_N}(c,f)(\mathbf{succ}(x)) = f(x, \mathbf{rec_N}(c,f)(x)).$$

Similarly, the type of (finite) lists of objects of a type A can be introduced by adding the following rules:

$$\frac{\Gamma \vdash A : Type_i}{\Gamma \vdash \mathbf{List}(A) : Type_i} \qquad \frac{\Gamma \vdash A : Type_i}{\Gamma \vdash \mathbf{nil}_A : \mathbf{List}(A)} \qquad \frac{\Gamma \vdash a : A \quad \Gamma \vdash l : \mathbf{List}(A)}{\Gamma \vdash \mathbf{cons}_A(a,l) : \mathbf{List}(A)}$$

$$\frac{\begin{array}{c}\Gamma \vdash C : \mathbf{List}(A) \to Type_i \quad \Gamma \vdash c : C(\mathbf{nil}_A) \\ \Gamma \vdash f : \Pi a{:}A \Pi l{:}\mathbf{List}(A).\ C(l) \to C(\mathbf{cons}_A(a,l))\end{array}}{\Gamma \vdash \mathbf{rec}_{\mathbf{List}(A)}(c,f) : \Pi l{:}\mathbf{List}(A).C(l)},$$

together with the computation rules

$$\begin{array}{rcl}\mathbf{rec}_{\mathbf{List}(A)}(c,f)(\mathbf{nil}_A) & = & c, \\ \mathbf{rec}_{\mathbf{List}(A)}(c,f)(\mathbf{cons}_A(a,l)) & = & f(a,l,\mathbf{rec}_{\mathbf{List}(A)}(c,f)(l)).\end{array}$$

Other data types or type constructors, such as that of trees, may also be introduced in a similar way. In Coquand and Paulin-Mohring (1990), Ore (1992) and Luo (1992), general approaches to introducing inductive types are considered. Using recursion operators such as $\mathbf{rec}_{\mathbf{N}}$ and $\mathbf{rec}_{\mathbf{List}(A)}$, one can define other functional programs; for example, a function computing the head of a list of natural numbers can be defined as: $hd_{\mathbf{N}} = \mathbf{rec}_{\mathbf{List}(\mathbf{N})}(0, \lambda x{:}\mathbf{N}\lambda l{:}\mathbf{List}(\mathbf{N})\lambda y{:}\mathbf{N}.x)$, which returns 0 when applied to the empty list and n when applied to a non-empty list $\mathbf{cons}_{\mathbf{N}}(n,l)$.

When dealing with predicative universes, it is often tedious to write the level subscripts. A technique has been developed (Huet 1987; Harper and Pollack 1991; Pollack 1990) to ease the tension of worrying about universe levels so that, in practice, one can omit the universe levels to write *Type* instead of $Type_i$. This is nicely implemented in the proof development system Lego (Pollack 1989; Luo and Pollack 1992). With such a facility, to assume $X{:}Type$ in a context is in some sense equivalent to assuming that X be an arbitrary type. One can also quantify over *Type* to talk about 'all types', bearing in mind that it is the machine that does the work of avoiding universe circularity (by giving an error message when it occurs). In this paper, we shall adopt such a principle of 'typical ambiguity' to omit the universe subscripts.

3. Specifications and Data Refinement

In this section, we introduce the notion of specification, show how abstract data types can be specified and show how a notion of abstract implementation between specifications can be defined and used for stepwise program development by refinement.

3.1. *Program specifications and their realizations*

Type theory can be seen as a functional programming language in which basic programs are functions in function spaces such as $\mathbf{N} \to \mathbf{N}$. It is well known that in type theories one can specify programs (Martin-Löf 1982; Nordström and Petersson 1983; Nordström *et al.* 1990). For example, in Martin-Löf's type theory, a specification of sorting programs for lists of natural numbers may be defined as the following Σ-type:

$$\Sigma f{:}\mathbf{List}(\mathbf{N}) \to \mathbf{List}(\mathbf{N}).\ \forall l{:}\mathbf{List}(\mathbf{N}).\ Sorted(l, f(l)),$$

whose objects are pairs of a program and a proof that the program is indeed a sorting program. This more traditional idea follows Martin-Löf's consideration of identifying specifications with types (Martin-Löf 1982 and 1984), which does not separate the computational content (programs) from the axiomatic requirements (correctness proofs). Considering such a separation as beneficial, Burstall (1989) considered the following notion of program specification in the context of **ECC**, where there is an internal notion of predicate.

Informally, a specification in the type theory consists of (a pair of) a type, whose objects are the possible programs (and program modules) that may realize the specification, and a predicate over the type, which specifies the properties that the realizations of the specification should satisfy.

Definition 3.1. (specifications) A *specification* SP consists of a type $\mathbf{Str}[SP]$ of *SP-structures*, called the *structure type of* SP, and a predicate $\mathbf{Ax}[SP]$ over $\mathbf{Str}[SP]$. The set of specifications is described by the following type:

$$\mathbf{SPEC} =_{\mathrm{df}} \sum [\mathbf{Str} : Type,\ \mathbf{Ax} : \mathbf{Str} \rightarrow Prop]\,.$$

For any type S, we also write $\mathbf{Spec}(S)$ for the class of specifications whose structure type is S.[†]

Remark Note that a specification is *not* just a type, but a pair. The pragmatic significance of this is that we can separate computational contents (expressed by the structure type of a specification) from the axiomatic requirements for the programs (also see a remark in Section 3.3). Such a separation is also the idea for the notion of mathematical theories as considered in Luo (1991a) and Luo *et al.* (1989). Note that the internal notion of predicate in the type theory is important for the formalization of such a notion of specification so that specifications can be manipulated *inside* the type theory.

The semantics of specifications is determined by the type theory, as given by the following notion of realization.

Definition 3.2. (realizations) Let SP be a specification. A *realization* (or *concrete implementation*) of SP is an SP-structure r such that $\mathbf{Ax}[SP](r)$ is provable, that is, an object r of type $\mathbf{Str}[SP]$ such that there is an object of type $\mathbf{Ax}[SP](r)$. SP is called *realizable* (or *consistent*) if there exists a realization.

Note that a realization is just a program (or program module), but *not* a pair of a program and the proof of its correctness.[‡] For example, using our notion of specification and realization, a specification *Sorting* of the sorting programs for lists of natural numbers

[†] Formally, $\mathbf{Spec}(S)$ is defined as $S \rightarrow Prop$, that is, the type of predicates over S. In this paper, for readability, we ignore the details of formal transformations between $\mathbf{Spec}(S)$ and **SPEC**, and use $\mathbf{Spec}(S)$ *informally* as the 'type' of specifications of the form (S, P), where P is of type $S \rightarrow Prop$.

[‡] One may use Σ-types to put together programs and their correctness proofs, *e.g.*, by defining a *model* of a specification SP as a pair of an SP-structure and a proof that the SP-structure satisfies $\mathbf{Ax}[SP]$. Then, the set of SP-models are given by the type $\mathbf{Mod}(SP) =_{\mathrm{df}} \Sigma s{:}\mathbf{Str}[SP].\mathbf{Ax}[SP](s)$.

is a pair consisting of the structure type $\mathbf{Str}[\mathit{Sorting}] = \mathbf{List(N)} \to \mathbf{List(N)}$ and the predicate $\mathbf{Ax}[\mathit{Sorting}] = \lambda f{:}\mathbf{List(N)} \to \mathbf{List(N)}.\ \forall l{:}\mathbf{List(N)}.\ \mathit{Sorted}(l, f(l))$, and a realization of *Sorting* is just a sorting function from **List(N)** to **List(N)**.

Specifications such as *Sorting* are concerned with the programs of *concrete* data types such as the types of natural numbers and lists of natural numbers. (See McKinna (1992) and Burstall and Mckinna (1992) for a more detailed account and examples of program development for concrete data types.) These data types are built-in as inductive data types whose meanings are determined by the associated rules, in particular, by the computation rules characterizing the computational equality that we take as a basic part of the understanding of those inductive data types. Therefore, the *adequacy* of the specifications concerned with concrete data types, for instance, that *Sorting* does specify the sorting programs, is in particular based on the fact that the propositional equality used in them (for the *Sorting* example, the use of propositional equality becomes clear when a full description of the relation *Sorted* is given) reflects or faithfully describes the computational equality. In **ECC**, the Leibniz equality can be used to describe the computational equality based on the property of equality reflection (see Section 2).[†] This issue of adequacy is best explained by the following simple example. Consider the following specification *ID* of the identity functions over natural numbers:

$$\mathbf{Str}[\mathit{ID}] =_{\mathrm{df}} \mathbf{N} \to \mathbf{N}, \quad \mathbf{Ax}[\mathit{ID}](f) =_{\mathrm{df}} \forall x{:}\mathbf{N}.(f(x) =_{\mathbf{N}} x).$$

The use of the Leibniz equality ($=_{\mathbf{N}}$) in the definition of $\mathbf{Ax}[\mathit{ID}]$ is adequate, because it reflects the computational equality (and hence our intention). For any realization *id* of *ID* and any natural number n (of type **N** in the empty context), we have $\mathit{id}(n) =_{\mathbf{N}} n$ (Leibniz-equal), which implies $\mathit{id}(n) = n$ (computationally equal), by the property of equality reflection.

3.2. *Specifications of abstract data types*

The examples of specifications considered above directly specify programs for concrete data types in the type theory. For (large) program development by stepwise refinement, we are more interested in specifications of abstract data types, *i.e.*, specifications of program modules with loose semantics, in the sense that a variety of data representations may be used to implement the abstract types and the corresponding functional operations. Σ-types, together with type universes, provide a good mechanism in the type theory for describing abstract structures, and can be used to express (types of) program modules. (See Nordström *et al.* (1990) for an example.) This gives us an important basis for specifying abstract data types.

However, careful consideration of the purpose of introducing abstract data types and the methodology of data refinement for program development shows that specification of abstract data types has different requirements from specification of programs for concrete data types. A particular point is that, in general, the equality over the abstract type should

[†] In Martin-Löf's type theory with weak intensional equality (see Nordström *et al.* (1990)), the weak equality can be used to describe computation, since it reflects the computational equality.

not be expressed by a propositional equality that reflects the computational equality (such as the Leibniz equality in our language or the intensional/extensional equality in Martin-Löf's type theories). The reason is that such a propositional equality is in fact the finest equality relation in the type theory;[†] to use it as the equality for an abstract type would not faithfully express the intention that the abstract type may further be implemented by a variety of rather different type structures (or data representations), and would prevent us from refining the specification of the abstract data type to another as expected. For example, consider an abstract type *Stack* of stacks (of natural numbers). It is natural to use array-pointer pairs to implement (or represent) a stack. However, if the equality over stacks is specified as the Leibniz equality, it would be impossible to choose such a data representation (see Example 3.6 of data refinement in Section 3.3). To emphasize, using a propositional equality that reflects the computational equality as the equality over an abstract type does not faithfully express our intention that the abstract type to be specified allows a variety of implementations via different data representations.

Therefore, instead of using a built-in equality, such as the Leibniz equality, as the equality for an abstract type, we associate the abstract type with an explicitly specified congruence relation to represent the intended equality.[‡] We now give an example to explain how abstract data types can be specified using Σ-types, while taking the above consideration into account. We shall use the following definition in our examples:

$$\mathbf{Setoid} =_{\mathrm{df}} \sum[Dom : Type,\ Eq : Dom \to Dom \to Prop].$$

A 'setoid' is a type together with a binary relation over the type; in a specification of abstract data type, the fact that the binary relation is a congruence is specified in the axiom part of the specification.

Example 3.3. (Stack) A specification of stacks (of natural numbers) can be given as follows:

$$\mathbf{Str[Stack(N)]} =_{\mathrm{df}} \sum \begin{bmatrix} Stack : \mathbf{Setoid} \\ empty : Dom[Stack] \\ push : \mathbf{N} \to Dom[Stack] \to Dom[Stack] \\ pop : Dom[Stack] \to Dom[Stack] \\ top : Dom[Stack] \to \mathbf{N} \end{bmatrix},$$

and for any structure S of type **Str[Stack(N)]**,

$$\begin{aligned} \mathbf{Ax[Stack(N)]}(S) =_{\mathrm{df}} & \quad \mathbf{Cong}_{\mathbf{Stack(N)}}(Eq) \\ & \& \quad Eq(pop(empty), empty) \\ & \& \quad top(empty) =_{\mathbf{N}} 0 \end{aligned}$$

[†] The leibniz equality is the finest equivalence relation in the type theory. In fact, for any binary reflexive relation R over type A, it is provable that $\forall x, y{:}A.\ (x =_A y) \supset R(x, y)$.

[‡] The condition that the equality is a congruence seems to be the weakest reasonable requirement. For design specifications, one may start with stronger equalities (even the Leibniz equality!); but the moral here is that one should be aware of the restrictions of those equalities on further refinement. For requirements specifications, it is not good to have such design decisions made too early.

$$\& \quad \forall n{:}\mathbf{N}\forall s{:}Dom[Stack].\ Eq(pop(push(n,s)),s)$$
$$\& \quad \forall n{:}\mathbf{N}\forall s{:}Dom[Stack].\ top(push(n,s)) =_{\mathbf{N}} n,$$

where *Stack* abbreviates *Stack*[*S*], *Eq* abbreviates *Eq*[*Stack*[*S*]], and so on for the others; $\mathbf{Cong}_{\mathbf{Stack(N)}}(Eq)$ stands for the following proposition expressing the fact that the binary relation between stacks is a congruence, that is, an equivalence relation respecting the operations over stacks:

$$\begin{aligned}\mathbf{Cong}_{\mathbf{Stack(N)}}(Eq) \ =_{\mathrm{df}}\ & Equiv(Eq)\ \&\\ & \forall s,s'{:}Dom[Stack].\ \ Eq(s,s') \supset\\ & \quad top(s) =_{\mathbf{N}} top(s')\ \&\ Eq(pop(s),pop(s'))\ \&\\ & \quad \forall m,n{:}\mathbf{N}.\ m =_{\mathbf{N}} n \supset Eq(push(m,s),push(n,s')).\end{aligned}$$

Remark Note that in the above example, we have used the Leibniz equality for the concrete data type **N** of natural numbers, and the associated consequence *Eq* for the abstract type of stacks. Such a distinction reflects our discussion above, and can be compared to the distinction between initial semantics and loose semantics in algebraic specifications (see Section 6 for a further discussion).

Given such a specification of an abstract data type, there are various possible realizations (concrete implementations), which are tuples of the structure type of the specification that satisfy the axiomatic requirements. For instance, one can realize the abstract type *Dom*[*Stack*] by the concrete data type **List**(**N**), and the stack operations such as *push* and *top* by functions such as $\mathbf{cons}_{\mathbf{N}}$ and $hd_{\mathbf{N}}$.

3.3. *Data refinement and implementation*

As well as considering concrete implementations (realizations) of specifications, a notion of (abstract) implementation is most important if we want to develop programs in a stepwise way by refinement between specifications. That is, we want to know what we mean by saying that a specification refines to (or is implemented by) another specification. Various notions have been proposed in the literature, starting from the early notion of abstraction function and representation function (Hoare 1972; Goguen *et al.* 1978) to the more recent considerations (see, for example, Sanella and Tarlecki (1988b), Ehrig and Mahr (1990), and Wirsing and Broy (1989)). We formalize in the type theory a simple notion of refinement (implementation). This notion comes from the consideration of theory morphisms in abstract reasoning (Luo *et al.* 1989; Luo 1991a) and is similar to the notion of deliverables (Burstall and McKinna 1991).

Definition 3.4. (refinement map and implementation) Let *SP* and *SP′* be specifications. A *refinement map* from *SP′* to *SP* is a function ρ from **Str**[*SP′*] to **Str**[*SP*],

$$\rho : \mathbf{Str}[SP'] \to \mathbf{Str}[SP],$$

such that the following proposition, called the *satisfaction condition*, is provable:

$$\mathbf{Sat}(\rho) =_{\mathrm{df}} \forall s'{:}\mathbf{Str}[SP'].\ \mathbf{Ax}[SP'](s') \supset \mathbf{Ax}[SP](\rho(s')).$$

If ρ is a refinement map from SP' to SP, we say that SP *refines to* (or *is implemented by*) *SP' through ρ*, written $SP \Longrightarrow_\rho SP'$.

Notation If $\mathbf{Str}[SP] = \mathbf{Str}[SP']$ and $id : \mathbf{Str}[SP'] \to \mathbf{Str}[SP]$ is the identity function, we may write $SP \Longrightarrow SP'$ to abbreviate $SP \Longrightarrow_{id} SP'$.

Remark Refinement maps are incomplete programs incorporating various design decisions made during the process of refinement implementation. Note that a refinement map is *not* a pair but just a map between the structure types that satisfies the satisfaction condition. (This is slightly different from the notion of theory morphism (Luo *et al.* 1989; Luo 1991a) and that of deliverable (Burstall and McKinna 1991).) Again, the separation of computational contents (programs) and their correctness proofs is emphasized here. Such a separation is very important when considering development of programs, since refinement maps (programs) should not contain unnecessary components concerned with (proofs of) implementation correctness.

Informally, a refinement map from SP' to SP determines the following subset of realizations of the original specification SP:

$$\{\, \rho(m') \in \mathbf{Str}[SP] \mid m' \in \mathbf{Str}[SP'] \text{ and } \mathbf{Ax}[SP'](m') \text{ is provable} \,\},$$

i.e., the images of the refinement map over the realizations of SP'. The stepwise development of programs from a specification SP_0 would be a sequence of refinement implementation steps:

$$SP_0 \Longrightarrow_{\rho_1} SP_1 \Longrightarrow_{\rho_2} \dots \Longrightarrow_{\rho_n} SP_n.$$

Any realization r_n of SP_n gives a realization r_0 of SP_0 as

$$r_0 =_{\mathrm{df}} \rho_1(\dots(\rho_n(r_n))),$$

and the composition of the correctness proofs for each step will give the proof that r_0 is a realization of SP_0. This justifies the fact that the implementation relation *composes vertically* (*cf.*, Burstall and Goguen (1980)), as expressed by the following proposition.

Proposition 3.5. (vertical composition) If $SP \Longrightarrow_\rho SP'$ and $SP' \Longrightarrow_{\rho'} SP''$, then $SP \Longrightarrow_{\rho\circ\rho'} SP''$. As a special case, we have $SP \Longrightarrow SP'$ and $SP' \Longrightarrow SP''$ imply $SP \Longrightarrow SP''$.

Remark As is well known, starting from a realizable specification, stepwise refinement does *not* necessarily lead to realizable specifications. It is the developer's responsibility to choose 'reasonable' and good design decisions that may lead to realizable implementing specifications and good programs that realize the requirements specifications.

Let us now see a traditional example of refinement – implementing stacks by array-pointer pairs. (This is an example for explanation and is not necessarily a good design for real software development.)

Example 3.6. We consider a specification **Array(N)** for arrays (of natural numbers), define a refinement map ρ from **Array(N)** to the specification **Stack(N)**, defined in Example 3.3, such that **Stack(N)** is implemented by **Array(N)** through ρ. The specification of **Array(N)**

is given by

$$\mathbf{Str[Array(N)]} =_{\mathrm{df}} \sum \begin{bmatrix} Array : \mathbf{Setoid} \\ newarray : Dom[Array] \\ assign : \mathbf{N} \to Dom[Array] \to \mathbf{N} \to Dom[Array] \\ access : Dom[Array] \to \mathbf{N} \to \mathbf{N} \end{bmatrix},$$

and for any structure A of type **Str[Array(N)]**,

$$\begin{aligned} \mathbf{Ax[Array(N)]}(A) =_{\mathrm{df}} \quad & \mathbf{Cong}_{\mathbf{Array(N)}}(Eq) \\ \& \quad & \forall i{:}\mathbf{N}.\ access(newarray, i) =_{\mathbf{N}} 0 \\ \& \quad & \forall n, i, j{:}\mathbf{N}\forall a{:}Dom[Array]. \\ & \qquad (i =_{\mathbf{N}} j \supset access(assign(n, a, i), j) =_{\mathbf{N}} n) \\ & \qquad \& \ (i \neq_{\mathbf{N}} j \supset access(assign(n, a, i), j) =_{\mathbf{N}} access(a, j)), \end{aligned}$$

where $\mathbf{Cong}_{\mathbf{Array(N)}}(Eq)$ is the proposition expressing the fact that the relation Eq between arrays is a congruence, similarly defined as $\mathbf{Cong}_{\mathbf{Stack(N)}}(Eq)$ in Example 3.3.
The refinement map ρ, given any **Array**(N)-structure A, generates the following **Stack**(N)-structure:

$$\begin{aligned} Dom[Stack[\rho(A)]] &=_{\mathrm{df}} \textstyle\sum[arr : Dom[Array[A]],\ ptr : \mathbf{N}] \\ Eq[Stack[\rho(A)]] &=_{\mathrm{df}} \lambda s, s'{:}Dom[Stack[\rho(A)]]. \\ &\qquad ptr[s] =_{\mathbf{N}} ptr[s'] \ \& \\ &\qquad \forall i{:}\mathbf{N}.\ i < ptr[s] \supset access(arr[s], i) =_{\mathbf{N}} access(arr[s'], i) \\ empty[\rho(A)] &=_{\mathrm{df}} (newarray[A],\ 0) \\ push[\rho(A)] &=_{\mathrm{df}} \lambda n{:}\mathbf{N}\lambda s{:}Dom[Stack[\rho(A)]]. \\ &\qquad (assign(n, arr[s], ptr[s]), ptr[s] + 1) \\ pop[\rho(A)] &=_{\mathrm{df}} \lambda s{:}Dom[Stack[\rho(A)]].\ (arr[s], ptr[s] - 1) \\ top[\rho(A)] &=_{\mathrm{df}} \lambda s{:}Dom[Stack[\rho(A)]]. \\ &\qquad \mathbf{rec}_{\mathbf{N}}(0,\ \lambda x, y{:}\mathbf{N}.access(arr[s], ptr[s] - 1))(ptr[s]). \end{aligned}$$

The chosen data representation is to use array-pointer pairs to represent stacks. Two such stack representations are equal if and only if the pointers (represented by natural numbers) are the same, and accessing the representing arrays at any position lower than the pointers gives the same result.
ρ defined above is indeed a refinement map from **Array**(N) to **Stack**(N), that is,

$$\mathbf{Stack(N)} \Longrightarrow_{\rho} \mathbf{Array(N)}.$$

Therefore, any realization r of **Array**(N) (which is realizable, see Wand (1992) for example) will give a realization $\rho(r)$ of **Stack**(N).
Note that the 'equality representation' (or 'representation invariants') in the traditional approaches to proofs of abstract implementation is directly reflected in the refinement map (its $Eq[Stack[\rho(A)]]$ part). The choice of the above data representation is possible because we have associated an explicit congruence rather than used a fixed equality such as the Leibniz equality for the abstract type of stacks.

It is obviously important in a refinement development of programs that the correctness of the (abstract) implementations must be verified. In other words, the satisfaction condition must be proved. Proof development systems based on type theories like Lego (Pollack 1989; Luo and Pollack 1992) can be used to verify the correctness of implementation. The above fact of implementation correctness has been formally checked in Lego; in fact, we have used Lego to develop the refinement map interactively on a machine.

Finally, we remark that, because of the separation of computational contents and the axiomatic part in the notions of specification and implementation (in particular, because our notion of refinement map is not a pair but a structure mapping that satisfies the satisfaction condition), step by step verification of the correctness in a multi-step development of programs by refinement is not necessary in practice. One can do several steps of refinement without worrying about the development of the correctness proofs and then verify the correctness by viewing the composition of the developed maps as one refinement map developed in a single step. Such flexibility is very useful in the practical development of software.

4. Modular design and structured specification

Modular design and structured specification have been generally accepted as two related useful methodologies for software development. There are two issues here. First, given a requirements specification to be implemented, programmers use principles such as divide-and-conquer and stepwise refinement to decompose and refine the specification until they reach suitable low-level specifications, which can be concretely implemented by, say efficient enough, software modules. This is the process of modular design, which may involve many intermediate design specifications, probably proposed by chief programmers and implemented separately by others. Second, to get a good requirements specification for a large software system, people must structure the specification in a modular way so that it is understandable and may suggest some possible design decisions.

Having given a notion of implementation in the last section, in this section we discuss modular design and structured specification in our type-theoretic approach (parameterized specifications are discussed in Section 5). In particular, we consider various specification operations[†] that can be used either in modular design by refinement or in structuring requirements specifications. An important property of such specification operations is the monotonicity with respect to the implementation relation (see below), which will ensure independent further refinements of the argument specifications (the so-called *horizontal composition* property (Burstall and Goguen 1980)).

4.1. *Decomposition and sharing*

Using the principle of divide-and-conquer in a design process, developers often decompose a specification into several independent ones (with clear interfaces) so that they can be

[†] A *specification operation* is a function that takes specifications (and possibly some other kinds of objects) as arguments and returns a specification as the result of application. Specification operations can also be seen as parameterized specifications. See Section 5.

implemented separately and then their realizations put together to get a realization of the original specification. Considering this, one might think that the notion of implementation given in the last section is over-simplified, since at first appearance it seemed to cover only the situations where a single line of refinement is pursued. In fact, this is not the case. Using the notion of implementation and Σ-types in our type theory, one can do specification decomposition by considering the following specification operation. (This also gives us a simple example to explain why the monotonicity of specification operations offers independence for further refinements.)

Definition 4.1. Let SP and SP' be specifications. Then, specification $SP \otimes SP'$ is defined as follows:

$$\mathbf{Str}[SP \otimes SP'] =_{\mathrm{df}} \mathbf{Str}[SP] \times \mathbf{Str}[SP']$$

and, for any s of type $\mathbf{Str}[SP \otimes SP']$,

$$\mathbf{Ax}[SP \otimes SP'](s) =_{\mathrm{df}} \mathbf{Ax}[SP](\pi_1(s)) \ \& \ \mathbf{Ax}[SP'](\pi_2(s)).$$

The (infix) specification operation $\otimes$ is of type $\mathbf{SPEC} \to \mathbf{SPEC} \to \mathbf{SPEC}$.

The way to use the above specification operation to decompose a specification SP into several (say two) independent specifications (SP_1 and SP_2) by a certain design strategy is to consider a refinement step of the following form:

$$SP \Longrightarrow_{\rho} SP_1 \otimes SP_2,$$

where the refinement map $\rho : \mathbf{Str}[SP_1 \otimes SP_2] \to \mathbf{Str}[SP]$ is the incomplete program expressing the design strategy at this decomposition step. The intention here is to implement the specifications SP_1 and SP_2 independently by further refinements. The soundness for such independent further refinements is guaranteed by the monotonicity of $\otimes$ with respect to the implementation relation.

Proposition 4.2. (monotonicity of $\otimes$) If $SP_1 \Longrightarrow_{\rho_1} SP_1'$ and $SP_2 \Longrightarrow_{\rho_2} SP_2'$, then

$$SP_1 \otimes SP_2 \Longrightarrow_{\rho_1 \otimes \rho_2} SP_1' \otimes SP_2',$$

where $\rho_1 \otimes \rho_2 =_{\mathrm{df}} \lambda s' {:} \mathbf{Str}[SP_1' \otimes SP_2'].(\rho_1(\pi_1(s')), \rho_2(\pi_2(s')))$.

By the monotonicity of $\otimes$ and the vertical composition property of the implementation relation (Proposition 3.5), we have that $SP \Longrightarrow_{\rho} SP_1 \otimes SP_2$ and $SP_i \Longrightarrow_{\rho_i} SP_i'$ ($i = 1,2$) imply $SP \Longrightarrow_{\rho \circ (\rho_1 \otimes \rho_2)} SP_1' \otimes SP_2'$ and, for any realizations r_i' of SP_i' ($i = 1,2$), $\rho \circ (\rho_1 \otimes \rho_2)(r_1', r_2')$ is a realization of SP.

In our discussion so far, we have only considered decomposition of a specification into several completely independent specifications. In practice, it is often the case that the sub-specifications are not completely independent but share some common structure.† For example, one may decompose a specification of a parser into several specifications

† Much attention has been paid to such structure sharing in the design of both programming languages (*e.g.*, Standard ML (MacQueen 1981; Milner *et al.* 1990) and Pebble (Burstall and Lampson 1984)) and specification languages (*e.g.*, Clear (Burstall and Goguen 1980) and Extended ML (Sanella and Tarlecki 1987)).

including *AbsSyn* for an abstract syntax tree generator and *SymTab* for management of the symbol table; these latter two specifications both use symbols and symbol management functions specified by another specification *Symbol*. Note that a parser can only work correctly when *AbsSyn* and *SymTab* use the same realization of *Symbol*. Such a structure sharing can be dealt with using Σ-types by considering the following specification operation $\sum$, which has ⊗ above as a special case.

Definition 4.3. Let SP be a specification and $P : \mathbf{Str}[SP] \rightarrow \mathbf{SPEC}$. Then, $\sum(SP,P)$ is the specification defined as follows:

$$\mathbf{Str}[\textstyle\sum(SP,P)] =_{\mathrm{df}} \Sigma s{:}\mathbf{Str}[SP].\mathbf{Str}[P(s)]$$

and, for any s' of type $\mathbf{Str}[\sum(SP,P)]$,

$$\mathbf{Ax}[\textstyle\sum(SP,P)](s') =_{\mathrm{df}} \mathbf{Ax}[SP](\pi_1(s')) \;\&\; \mathbf{Ax}[P(\pi_1(s'))](\pi_2(s')).$$

$\sum$ is of type $\Pi SP{:}\mathbf{SPEC}.\ (\mathbf{Str}[SP] \rightarrow \mathbf{SPEC}) \rightarrow \mathbf{SPEC}$.

Proposition 4.4. (monotonicity of $\sum$) Let SP and SP' be specifications, $P : \mathbf{Str}[SP] \rightarrow \mathbf{SPEC}$ and $P' : \mathbf{Str}[SP'] \rightarrow \mathbf{SPEC}$. If

1 $SP \Longrightarrow_{\rho} SP'$, and
2 δ is a function of type $\Pi s'{:}\mathbf{Str}[SP'].\ \mathbf{Str}[P'(s')] \rightarrow \mathbf{Str}[P(\rho(s'))]$ such that $\forall s'{:}\mathbf{Str}[SP'].\ P(\rho(s')) \Longrightarrow_{\delta(s')} P'(s')$ is provable,[†]

then,

$$\textstyle\sum(SP,P) \Longrightarrow_{\sum(\rho,\delta)} \sum(SP',P'),$$

where $\sum(\rho,\delta) =_{\mathrm{df}} \lambda s'{:}\mathbf{Str}[\sum(SP',P')].\ (\rho(\pi_1(s')),\delta(\pi_1(s'),\pi_2(s')))$.

Suppose P is of the form $\lambda s_0{:}\mathbf{Str}[SP_0].\ P_1(s_0) \otimes P_2(s_0)$. Then, a refinement step

$$SP \Longrightarrow_{\rho} \textstyle\sum(SP_0,P)$$

decomposes SP into three specifications SP_0, $P_1(s_0)$ and $P_2(s_0)$; the latter two share a common structure specified by SP_0. The above monotonicity result suggests one should decompose SP into SP_0 and (parameterized specification) P, which can then be further refined independently (see Section 5 for refinement of parameterized specifications). Another way to look at the further refinement of $\sum(SP_0,P)$ is to consider SP_0 and the following specification SP':

$$\begin{aligned}\mathbf{Str}[SP'] &=_{\mathrm{df}} \Pi s_0{:}\mathbf{Str}[SP_0].\ \mathbf{Str}[P(s_0)],\\ \mathbf{Ax}[SP'](f) &=_{\mathrm{df}} \forall s_0{:}\mathbf{Str}[SP_0].\ \mathbf{Ax}[SP_0](s_0) \supset \mathbf{Ax}[P(s_0)](f(s_0)).\end{aligned}$$

In other words, we proceed to implement $P_1(s_0)$ and $P_2(s_0)$ independently, assuming that s_0 is an arbitrary realization of SP_0. Note that SP_0 and SP' are independent of each other and have a clear interface specified by Π. To get a realization of $\sum(SP_0,P)$, we simply put together any realization r_0 of SP_0 and the result of applying any realization of

[†] P and P' are parameterized specifications and, when SP and SP' have the same structure type and ρ is the identity function, the condition for δ here is to say that the parameterized specification P refines to P' through δ. See Definition 5.2 in Section 5.

SP' to r_0. (The reader may have noticed that the above refinement has suggested another specification operation corresponding to the type constructor Π. We do not elaborate this here.)

Warning As remarked after Proposition 3.5, not all such decompositions can lead to solutions; in other words, one may go into a blind alley – some of the sub-specifications are not realizable. Here is a trivial example: Decomposing a (realizable) specification whose structure type is $\Sigma X{:}Type_0.X$ into two specifications with $Type_0$ and $\Pi X{:}Type_0.X$ as structure types, respectively, produces an inconsistent specification (the second one), since $\Pi X{:}Type_0.X$ has no object in any consistent context in the type theory. In any stage of refinement development, the programmer must be careful about such a consistency argument. If necessary, one may verify that certain intermediate design specifications are realizable. For the above situation concerning sharing, if independent decomposition is not feasible, we have to first refine SP_0 to SP_0' (with the same structure type) so that such a decomposition for $\sum(SP_0', P)$ is possible, or we simply find an (intended) realization r_0 of SP_0 and then implement $P(r_0)$.

4.2. *Constructors and selectors*

Since the work by Burstall and Goguen on the specification language Clear (Burstall and Goguen 1980), it has been generally accepted that specification operations play important roles both in modular design by refinement and in structuring specifications. For example, we can define the following simple specification operations that can often be used in structuring specifications:

Join$_S$: 'puts together' the axiomatic parts of two specifications over the same structure type S. If $\mathbf{Str}[SP] = \mathbf{Str}[SP'] = S$, then

$$\begin{aligned}\mathbf{Str}[\mathbf{Join}_S(SP, SP')] &=_{\mathrm{df}} S,\\ \mathbf{Ax}[\mathbf{Join}_S(SP, SP')](s) &=_{\mathrm{df}} \mathbf{Ax}[SP](s)\ \&\ \mathbf{Ax}[SP'](s).\end{aligned}$$

Join is of type

$$\Pi S{:}Type.\ \mathbf{Spec}(S) \to \mathbf{Spec}(S) \to \mathbf{Spec}(S).$$

Similarly, one may define **Meet**$_S$ with $\mathbf{Ax}[\mathbf{Meet}_S(SP, SP')](s) =_{\mathrm{df}} \mathbf{Ax}[SP](s) \vee \mathbf{Ax}[SP'](s)$, and other possibly useful operators by means of logical operators.

Extend: extends a specification by some extra structure-components and/or some axioms. Given a specification SP, an extension Ext_Str of $\mathbf{Str}[SP]$, which is a function of type $\mathbf{Str}[SP] \to Type$, and some axioms (a predicate) Ext_Ax over the extended structure type (*i.e.*, $\Sigma s{:}\mathbf{Str}[SP].\ Ext\_Str(s)$), define

$$\begin{aligned}\mathbf{Str}[\mathbf{Extend}(SP, Ext\_Str, Ext\_Ax)] &=_{\mathrm{df}} \Sigma s{:}\mathbf{Str}[SP].\ Ext\_Str(s),\\ \mathbf{Ax}[\mathbf{Extend}(SP, Ext\_Str, Ext\_Ax)](s') &=_{\mathrm{df}} \mathbf{Ax}[SP](\pi_1(s'))\ \&\ Ext\_Ax(s').\end{aligned}$$

Extend is of type

$$\Pi SP{:}\mathbf{SPEC}\Pi f{:}\mathbf{Str}[SP] \to Type\Pi g{:}(\Sigma s{:}\mathbf{Str}[SP].f(s)) \to Prop.\ \mathbf{SPEC}.$$

There are various specification operations that can be defined. Instead of studying them one by one (*e.g.*, considering whether they are monotone), we define two general classes of specification operations called *constructors*[†] and *selectors*, which are determined by functions between structure types.

Definition 4.5. (constructors and selectors) Let S and S' be types and $\rho : S' \to S$. The specification operations $\mathbf{Con}_\rho$ and $\mathbf{Sel}_\rho$ are defined as follows:

1 $\mathbf{Con}_\rho$, *the constructor determined by* ρ, is a specification operation of type $\mathbf{Spec}(S') \to \mathbf{Spec}(S)$ defined as: for any SP' with $\mathbf{Str}[SP'] = S'$,

$$\begin{aligned} \mathbf{Str}[\mathbf{Con}_\rho(SP')] &=_{\mathrm{df}} S, \\ \mathbf{Ax}[\mathbf{Con}_\rho(SP')](s) &=_{\mathrm{df}} \exists s'{:}S'.\ \mathbf{Ax}[SP'](s') \ \&\ \rho(s') =_S s. \end{aligned}$$

2 $\mathbf{Sel}_\rho$, *the selector determined by* ρ, is a specification operation of type $\mathbf{Spec}(S) \to \mathbf{Spec}(S')$ defined as: for any specification SP with $\mathbf{Str}[SP] = S$,

$$\begin{aligned} \mathbf{Str}[\mathbf{Sel}_\rho(SP)] &=_{\mathrm{df}} S', \\ \mathbf{Ax}[\mathbf{Sel}_\rho(SP)](s') &=_{\mathrm{df}} \mathbf{Ax}[SP](\rho(s')). \end{aligned}$$

Intuitively, the constructor $\mathbf{Con}_\rho$ applied to specification SP' constructs as its realizations the images of ρ over the SP'-realizations, while the selector $\mathbf{Sel}_\rho$ applied to SP selects the inverse images of the SP-realizations by ρ. Interesting specification operations can be defined by using selectors and constructors. For example, **Join** and **Extend** discussed above can be defined in the following way:

1 The operation $\mathbf{Join}_S$ can be defined by

$$\mathbf{Join}_S(SP, SP') =_{\mathrm{df}} \mathbf{Sel}_d(SP \otimes SP'),$$

where $d =_{\mathrm{df}} \lambda s{:}S.(s,s) : S \to S \times S$ is the diagonal function over S.

2 The operation **Extend** can be defined by

$$\mathbf{Extend}(SP, Ext\_Str, Ext\_Ax) =_{\mathrm{df}} \mathbf{Join}_S(\mathbf{Sel}_{\pi_1}(SP), (S, Ext\_Ax)),$$

where $S = \Sigma s{:}\mathbf{Str}[SP].\ Ext\_Str(s)$ and $\pi_1 : S \to \mathbf{Str}[SP]$ is the first projection function.

The constructors can be used to play a role of 'renaming' and information hiding similar to the operation **derive** in the specification language ASL (Sanella and Wirsing 1983; Wirsing 1986). **derive** in ASL is based on a signature morphism from the signature of the resulting specification to that of the argument specification. Such a signature morphism σ, when it is a signature inclusion, corresponds to a (forgetful) map ρ from the structure type of the argument specification to that of the resulting specification; and in such a case, $\mathbf{Con}_\rho(SP)$ corresponds to **derive** SP **from** σ. Similarly, the operation **translate** (Sanella and Tarlecki 1988a) can be simulated as selectors.

It is easy to verify the following basic properties of constructors and selectors.

Proposition 4.6. Let $\rho : S' \to S$.

[†] The name 'constructor' comes from the similarity of this class of specification operations to Sannella and Tarlecki's notion of constructor. See Section 4.3.

Realizability:

1 If $SP' : \mathbf{Spec}(S')$ is realizable, so is $\mathbf{Con}_\rho(SP')$.

2 If $\mathbf{Sel}_\rho(SP)$ is realizable, so is SP.

Monotonicity:

1 For $SP_1', SP_2' : \mathbf{Spec}(S')$, $SP_1' \Longrightarrow SP_2'$ implies $\mathbf{Con}_\rho(SP_1') \Longrightarrow \mathbf{Con}_\rho(SP_2')$.

2 For $SP_1, SP_2 : \mathbf{Spec}(S)$, $SP_1 \Longrightarrow SP_2$ implies $\mathbf{Sel}_\rho(SP_1) \Longrightarrow \mathbf{Sel}_\rho(SP_2)$.

4.3. *Constructor/selector implementation*

The constructor operations are very similar in spirit to the notion of constructors (functions between algebra classes) introduced in Sanella and Tarlecki (1988b), although they are semantically different. Sannella and Tarlecki have proposed the idea to suggest the following refinement methodology: starting from an initial specification SP to be implemented, one uses constructors to specify some specification implementing SP and then goes on to implement the argument specifications of the constructors used in this step. Such a method applies in our setting as well. In fact, we can define a similar notion of constructor/selector implementation, which turns out to be equivalent to the notion of implementation we have defined. This enables us to relate our approach to that in algebraic specifications and gives a better understanding of the notion of implementation.

Definition 4.7. (constructor/selector implementation) Let SP and SP' be specifications, and ρ be of type $\mathbf{Str}[SP'] \to \mathbf{Str}[SP]$.

1 *SP is implemented by SP' through constructor ρ* (notation $SP \overset{c}{\Longrightarrow}_\rho SP'$) if $SP \Longrightarrow \mathbf{Con}_\rho(SP')$.

2 *SP is implemented by SP' through selector ρ* (notation $SP \overset{s}{\Longrightarrow}_\rho SP'$) if $\mathbf{Sel}_\rho(SP) \Longrightarrow SP'$.

Proposition 4.8. Let SP and SP' be specifications and $\rho : \mathbf{Str}[SP'] \to \mathbf{Str}[SP]$. Then, the following are equivalent:

$$SP \Longrightarrow_\rho SP', \quad SP \overset{c}{\Longrightarrow}_\rho SP', \quad SP \overset{s}{\Longrightarrow}_\rho SP'.$$

Proof. The first and the last statements are computationally equal, and they are logically equivalent to the second. □

5. Parameterized specification

Parameterization is a powerful abstraction tool, both for modular design and for structured specification. It may enhance the reusability of program modules and their specifications. A type theory with good structural facilities can provide powerful higher-order parameterization mechanisms for parameterized specifications as well as parameterized program modules.

In fact, we have seen an example of the use of parameterized specifications in Section 4.1, where we considered implementation of a specification of the form $\sum(SP_0, P)$. We pointed out there that there are at least two design decisions that such a form of specifications may suggest: one is to decompose it into two independent specifications SP_0 and SP', where SP' is a specification of parameterized program modules (using the type constructor Π); another is to make direct use of the monotonicity property of the specification operation $\sum$ to consider further refinements of the specification SP_0 and the *parameterized specification* P. Taking this latter view, we must consider parameterized specifications and their implementations.

Of course, the need for and usefulness of parameterized specifications in modular design and structured specification cannot be explained completely by a simple example. We will not elaborate this in this paper. Among the large amount of literature on this are Burstall and Goguen (1980), Sanella and Wirsing (1983), Ehrig and Mahr (1985) and in particular Sanella *et al.* (1990), where a recent account of this issue in algebraic specification can be found.

5.1. *Parameterized specifications*

Parameterized specifications are functions in the type theory, which applied to its arguments return specifications as results. In other words, parameterized specifications have types of the following forms:

$$\Pi x_1{:}A_1 ... \Pi x_n{:}A_n.\ \mathbf{SPEC} \quad \text{or} \quad \Pi x_1{:}A_1 ... \Pi x_n{:}A_n.\ \mathbf{Spec}(S),$$

where $n \geq 1$. Note that the forms of arguments to which a parameterized specification can apply are not restricted here; they can be any kinds of objects including structures (program modules), specifications and any kinds of parameterized objects.

For example, we may parameterize the specification of stacks (see Example 3.3) in two different ways. First, given any (non-empty) concrete data type A with a congruence relation, the parameterized specification returns a specification of stacks for that concrete data type. This can be done in the obvious way in our type-theoretic setting; for example, the stack parameterized over concrete data types would look like

$$\mathbf{Stack} = \lambda X{:}Type\lambda x{:}X\lambda R{:}X \to X \to Prop.\ \mathbf{Stack}(X,x,R),$$

where $\mathbf{Stack}(X,x,R)$ is the same as $\mathbf{Stack}(\mathbf{N})$, except that $\mathbf{N}$, 0 and $=_{\mathbf{N}}$ are replaced by X, x and R, respectively. (Depending on the way **Stack** is to be used, one may require that R be a congruence by adding another argument to **Stack** or remove the argument R by using Leibniz's equality over X.) Such a parameterization is over concrete program modules, and the parameterized specification P involved in specification of the form $\sum(SP_0, P)$ is of such a kind.

Considering structured specifications and modular design, we may parameterize a specification over specifications. This is what is normally meant by parameterized specification in the algebraic approach to specifications (*cf.*, Clear (Burstall and Goguen 1980) and other specification languages). For example, given any specification of an abstract data type (*e.g.*, of sets, stacks or arrays), we may want to extend them by a specification of stacks to get a specification of stacks of sets, stacks or arrays, *etc.* Instead of doing them one by one, we want to parameterize the specification of stacks over such specifications. The example below explains how this can be done.

There is a further point to make before we give the example. A parameterized specification **STACK** extending specifications by stacks cannot take an arbitrary specification as its argument; the structure type of an eligible argument specification must have a distinguished type with some object. In the algebraic approach to specifications, this is usually done by considering a special specification (usually called **Elem**) as the parameter specification. This has only one sort and one constant of the sort (see Burstall and Goguen (1980) for example). Satisfaction (or matching) of an argument specification to a parameter specification is through a signature morphism from the parameter specification to the argument. In other words, we need to talk about the 'components' of the structure type of specifications. A way to do this in the type theory is to use functions to indicate the components of a structure type. For example, for any type S, a function of type

$$\mathbf{Elem}(S) =_{\mathrm{df}} S \to \sum[X{:}\mathbf{Setoid},\ x{:}Dom[X]]$$

can be used as a *component indicator*, which, given any structure of type S, identifies a type (with a binary relation) and an object of the type. For instance, the following function (*cf.*, Example 3.6)

$$ind\_array =_{\mathrm{df}} \lambda A{:}\mathbf{Str}[\mathbf{Array}(\mathbf{N})].\ (Array[A], newarray[A])$$

is of type $\mathbf{Elem}(\mathbf{Str}[\mathbf{Array}(\mathbf{N})])$ and may be used in the application of parameterized specification **STACK** below to generate a specification of stacks of arrays of natural numbers.

Example 5.1. (STACK) We define a parameterized specification **STACK**, which, when applied to a specification whose structure type has a distinguished non-empty setoid, returns as result a specification that extends the argument specification by a stack specification over the indicated setoid. We shall use the specification operation **Extend** to define **STACK**. First, we define two preliminary functions for extensions of structure type and axioms, respectively.

1 Ext_Str_Stack is a function of type $\Pi S{:}Type.\ \mathbf{Elem}(S) \to S \to Type$.
Given a type S, a function $Elem$ of type $\mathbf{Elem}(S)$ and an object s of type S, $Ext\_Str\_Stack(S, Elem, s)$ is defined to be the same as the structure type of $\mathbf{Stack}(\mathbf{N})$ in Example 3.3, except that we replace $\mathbf{N}$ by $Dom[X[Elem(s)]]$.
2 Ext_Ax_Stack is the function of type

$$\Pi S{:}Type\Pi Elem{:}\mathbf{Elem}(S).\ (\Sigma s{:}S.Ext\_Str\_Stack(S, Elem, s)) \to Prop.$$

Given a type S, $Elem$ of type $\mathbf{Elem}(S)$ and s' of type $\Sigma s{:}S.Ext\_Str\_Stack(S, Elem, s))$, $Ext\_Ax\_Stack(S, Elem, s')$ is the proposition defined in the same way as the axiom part of $\mathbf{Stack}(\mathbf{N})$ in Example 3.3, except that we replace $\mathbf{N}$, 0 and $=_{\mathbf{N}}$ by $Dom[X[Elem(\pi_1(s'))]]$, $x[Elem(\pi_1(s'))]$ and $Eq[X[Elem(\pi_1(s'))]]$, respectively.

Now, we define parameterized specification **STACK** as follows:

$$\begin{aligned}\mathbf{STACK}\ =_{\mathrm{df}}\ & \lambda S{:}Type\lambda Elem{:}\mathbf{Elem}(S)\lambda SP{:}\mathbf{Spec}(S).\\ & \mathbf{Extend}(SP, Ext\_Str\_Stack(S, Elem), Ext\_Ax\_Stack(S, Elem)),\end{aligned}$$

which is of type

$$\Pi S{:}Type\Pi Elem{:}\mathbf{Elem}(S).\ \mathbf{Spec}(S) \rightarrow \mathbf{Spec}(\Sigma s{:}S.Ext\_Str\_Stack(S, Elem, s)).$$

Applying **STACK** to, for example, the specification **Array**(**N**) (see Example 3.6) with the component indicator *ind_array* defined above will result in the specification of stacks of arrays of natural numbers:

$$\mathbf{STACK}(\mathbf{Str}[\mathbf{Array}(\mathbf{N})], ind\_array, \mathbf{Array}(\mathbf{N})).$$

5.2. *Implementation of parameterized specifications*

In a design process, it is often natural to decompose a specification into several specifications, some of which are parameterized specifications. For example, when a specification is of the form $P(SP)$ or $\sum(SP, P)$, a decomposition into SP and the parameterized specification P may be desirable. Such a need calls for a notion of implementation between parameterized specifications.

Definition 5.2. (implementation of parameterized specifications) Let P and P' be parameterized specifications over the same parameter type Par. A *refinement map* from P' to P is a function

$$\delta : \Pi s{:}Par.\ \mathbf{Str}[P'(s)] \rightarrow \mathbf{Str}[P(s)]$$

such that the following *satisfaction condition* is provable:

$$\mathbf{Sat}(\delta) =_{\mathrm{df}} \forall s{:}Par.\ P(s) \Longrightarrow_{\delta(s)} P'(s).$$

If δ is a refinement map from P' to P, we say that P *refines to* (or *is implemented by*) P' *through* δ, written $P \Longrightarrow_{\delta} P'$.

Remark A parameterized specification P is implemented by P' if P' implements P pointwisely through a *uniform* refinement map. The essential idea of pointwise implementation comes from Sanella and Wirsing (1983). Note that the polymorphism and type dependency in type theory gives a nice way of expressing a (uniform) family of refinement maps as a single polymorphic function. You may have noticed the similarity between this definition and the notion of natural transformation between functors in category theory. Although the above definition is already rather general, one may further consider an implementation between two parameterized specifications with parameter types that may be different. We will not expand on this here.

The above notion of implementation composes vertically.

Proposition 5.3. (vertical composition) Let P, P' and P'' be parameterized specifications with the same parameter type Par. If $P \Longrightarrow_{\delta} P'$ and $P' \Longrightarrow_{\delta'} P''$, then $P \Longrightarrow_{\delta \bullet \delta'} P''$, where $\delta \bullet \delta' =_{\mathrm{df}} \lambda s{:}Par.\ \delta(s) \circ \delta'(s)$.

Example 5.4. Following Example 5.1, we can similarly define a parameterized specification (*cf.*, Example 3.6) :

$$\begin{aligned}\mathbf{ARRAY} \ =_{\mathrm{df}}\ & \lambda S{:}Type\lambda Elem{:}\mathbf{Elem}(S)\lambda SP{:}\mathbf{Spec}(S).\\ & \mathbf{Extend}(SP, Ext\_Str\_Array(S,Elem), Ext\_Ax\_Array(S,Elem)).\end{aligned}$$

$\mathbf{ARRAY}(S, Elem, SP)$ extends the argument specification SP by a specification of arrays. For any type S and any component indicator $Elem$ of type $\mathbf{Elem}(S)$, we can find a refinement map δ (*cf.*, Example 3.6) such that $\mathbf{STACK}(S, Elem) \Longrightarrow_\delta \mathbf{ARRAY}(S, Elem)$.

There are two kinds of parameter types that often occur: the structure type of some specification, which we have seen in a specification of the form $\sum(SP, P)$, or a type of specifications (*e.g.*, $\mathbf{Spec}(S)$ or **SPEC**). In the latter case, it is important to consider the property of *horizontal composition* of the implementation relation (Burstall and Goguen 1980), since it guarantees that we can implement a specification of the form $P(SP)$ by implementing SP and the parameterized specification P separately. The above notion of implementation also enjoys the property of horizontal composition when a parameterized specification is monotone with respect to the implementation relation between specifications.

Definition 5.5. (monotonicity) Let Par be **SPEC** or $\mathbf{Spec}(S)$. A parameterized specification P of type $Par \to \mathbf{SPEC}$ is *monotone* if and only if there is a function

$$f : \Pi A, B{:}Par.\ (\mathbf{Str}[B] \to \mathbf{Str}[A]) \to (\mathbf{Str}[P(B)] \to \mathbf{Str}[P(A)])$$

such that $\forall A, B{:}Par\forall\rho{:}\mathbf{Str}[B] \to \mathbf{Str}[A].\ (A \Longrightarrow_\rho B) \supset (P(A) \Longrightarrow_{f(\rho)} P(B))$ is provable. If so, we say P is *monotone via* f.

Proposition 5.6. (horizontal composition) Let Par be **SPEC** or $\mathbf{Spec}(S)$, $SP, SP' : Par$ and $P, P' : Par \to \mathbf{SPEC}$. Then, $P(SP) \Longrightarrow_\rho P'(SP')$ for some ρ if the following conditions hold:

$$SP \Longrightarrow_{\rho_0} SP', \quad P \Longrightarrow_\delta P', \quad \text{and} \quad P \text{ or } P' \text{ is monotone.}$$

Proof. Define

$$\rho =_{\mathrm{df}} \begin{cases} f(SP, SP', \rho_0) \circ \delta(SP') & \text{if } P \text{ is monotone via } f,\\ \delta(SP) \circ g(SP, SP', \rho_0) & \text{if } P' \text{ is monotone via } g.\end{cases}$$

Then, we have $P(SP) \Longrightarrow_\rho P'(SP')$. □

The property of horizontal composition shows that we can implement a specification of the form $P_0(SP_0)$ by independent refinements $P_0 \Longrightarrow_{\delta_1} \dots \Longrightarrow_{\delta_m} P_m$ and $SP_0 \Longrightarrow_{\rho_1} \dots \Longrightarrow_{\rho_n} SP_n$ when the parameterized specifications involved are monotone.

Finally, note that the above definition of monotonicity and Proposition 5.6 can easily be generalized to the case where parameterized specifications have more than one specification as arguments (Par is of the form $Par_1 \times \dots \times Par_n$, where Par_i is either **SPEC** or $\mathbf{Spec}(S_i)$).

6. Conclusion and discussion

We have considered a type-theoretic approach to program specification and data refinement in a type theory with a strong logical power and good structural mechanisms. The higher-order facilities in the type theory provide useful mechanisms for modular design and structured specification. A notable advantage of this approach is that it provides a uniform language for modular programming, structured specification, logical reasoning, and modular development of programs. We have been able to formalize internally notions such as implementation in the type theory, and this enables us to use a computer implementation of the type theory (Lego) to develop the refinement maps (programs) and the correctness proofs of implementations.

From the above, it is clear that our work has been highly influenced by the existing work on specifications, including algebraic specification, and especially by the methodological developments made so far. Comparative studies of our type-theoretic approach with other approaches are beyond the scope of this paper and need further research. Some work along this line has been in progress since the publication of Burstall and Mckinna's work on deliverables (Burstall and Mckinna 1991) and an earlier version of this paper (Luo 1991b). For example, Reus and Streicher (1992) has shown that the laws of module algebra in algebraic specifications can be translated into the type-theoretic approach and proved to be sound. In the following, before discussing related work and further research topics, we give a brief discussion of several aspects of the type-theoretic approach, both to highlight some of the differences as well as the similarities to other approaches such as algebraic specification. Such a discussion, we hope, may make our motivations and the technical development in this paper clearer, and would be useful for readers who are familiar with other approaches to specification and development of programs.

The most important aspect to be emphasized is that we have used type theory as a *uniform* language – it is a functional programming language, a specification language and a language for logical reasoning. What has been presented above is the specification and modular development of functional programs *in* the type theory. Therefore, we are *not* looking for a general semantic understanding of specifications for its own sake, but rather to produce a single language that incorporates both programs and their specifications. This is different from the traditional style of specifications using, *e.g.*, Hoare logic, which is a logical language built on top of a programming language such as Pascal. It is also different from the algebraic approach to specifications, where a general semantical study of specifications is considered, but the operational (or computational) notion of program and program modules seems to be ignored, and programs are semantically modelled by the notion of algebra (based on set theory). As algebras are *not* computational programs, there is a need in the algebraic approach to fill in the gap between specifications and the programs in ordinary programming languages (*cf.*, the development of languages such as Extended ML (Sanella and Tarlecki 1987)), and between the algebraic semantics of specifications and the operational semantics of programs (*e.g.*, the semantics given by term-rewriting). Our study, using type theory for program specification and development, offers a single language (the type theory), which has a simple operational meaning theory (*cf.*, Martin-Löf (1984) and Luo (1993)), and on which basis, the programs and specifications in the language are understood.

Compared with set theory, type theory is a more manageable formalism, as shown by its proof-theoretic properties and the success in computer implementation of various proof development systems based on type theory. However, although they are different, types have fundamental similarities with sets, which allow us to formalize many pragmatic applications nicely in the type theory. Our internal formalization of the various notions for modular refinement, such as abstract implementation, is such an example. Therefore, with respect to formalization of specifications and their refinement, we have gained a good compromise between the model-theoretic approach (*cf.*, for example, Sanella and Tarlecki (1988b)), which is simple and powerful (but usually defined on a set theory basis, which is essentially non-computational and, hence, for which it is difficult to find suitable computer supports for proof development), and simplicity from the computational point of view, which allows a good implementation of the uniform language – type theory. This is why we said in the introduction that the semantics of specifications and their refinement is 'model-theoretic', in the sense that types (Σ-types) are used to represent sets of program modules, and functions (functional programs in the type theory) are used to represent refinement maps, which are programs in the type theory. This allows us to *internalize* those notions, which are usually defined at the meta level (*e.g.*, in set theory).

For readers familiar with algebraic specifications, the following remarks on some technical points may be helpful. In type theory, we have a distinction between concrete data types and abstract data types, as discussed in Section 3. This is comparable to the distinction between the initial semantics and the loose semantics in algebraic specifications, where specifications are the only kind of entities to be considered. The types such as $\mathbf{N}$ of natural numbers, are data types in the type theory, and in algebraic specifications are specified with initial semantics; the specifications of abstract data types have a variety of possible realizations, and in algebraic specifications are considered as having loose semantics. Based on such a view, a Σ-type is regarded as representing a class of program modules (which in algebraic specification are semantically considered as a class of algebras), and structure mappings between Σ-types correspond to functions between classes of algebras. Note that in our setting, the notions of signature and signature morphisms are not necessary, as we are working in type theory, which is comparable to working in a 'syntactically manageable set theory'. However, there is an essential difference here; that is, the refinement maps are themselves (computational) programs rather than non-computational functions in set theory.

Among the related work, that by Taylor, Pollack and the present author on abstract reasoning (Luo *et al.* 1989; Luo 1991a) and Burstall's idea of deliverables (Burstall 1989; Burstall and Mckinna 1992) were most influential on this work (in particular, the basic notion of implementation between specifications). MacQueen (1986) and Nordström *et al.* (1990) were the primary sources for the author's use of Σ-types to represent sets of program modules. Sannella, Sokolowski and Tarlecki (Sannella *et al.* 1992) have recently proposed ideas of higher-order parameterization and are working on a specification formalism based on ASL and incorporating some type-theoretic constructions. The author has appreciated very much their argument on the difference between parameterized specifications and specifications of program modules, and this has influenced the careful distinction of these two in this paper.

Discussions in this paper have omitted an important aspect of specification and implementation, that is, observational equivalence. In our setting, we can consider observational specification as well. For example, Sannella and Tarlecki's notion of abstractor implementation (with observational equivalence as a special case, see Sannella and Tarlecki

(1988b)) can be similarly dealt with here by introducing the following specification operation:

Abstractor: for an equivalence relation R over some structure type S, $\mathbf{Abs}[R] : \mathbf{Spec}(S) \rightarrow \mathbf{Spec}(S)$ is defined as follows:

$$\begin{aligned} \mathbf{Str}[\mathbf{Abs}[R](SP)] &=_{\mathrm{df}} S, \\ \mathbf{Ax}[\mathbf{Abs}[R](SP)](s) &=_{\mathrm{df}} \exists s'{:}S.\ \mathbf{Ax}[SP](s') \ \&\ R(s,s'); \end{aligned}$$

and the notion of abstractor implementation would be given as:

Abstractor implementation: SP is implemented by SP' via R (an equivalence relation over $\mathbf{Str}[SP]$) through refinement map $\rho : \mathbf{Str}[SP'] \rightarrow \mathbf{Str}[SP]$, written $SP \overset{a}{\Longrightarrow}{}^{R}_{\rho} SP'$, if and only if,

$$\mathbf{Abs}[R](SP) \Longrightarrow_{\rho} SP'.$$

Further research is needed to consider this topic in more detail.

Finally, there is a very interesting relationship between the researches in program development and in proof development systems that are developed for theorem proving. For example, the refinement proof development style in Lego has many features in common with program development. The technical machinery developed in this paper for structured specifications and modular refinement is closely related to, and can be applied to, theorem-proving (in the large) (Luo 1991a). Based on the experience of developing the existing proof development systems, it would be interesting for people to develop special environments for program specification and development based on type theory.

Acknowledgements I am in debt to Paul Taylor with whom I started to consider the notion of theory morphism in abstract reasoning. I am grateful to Rod Burstall, Don Sannella and many members of the Lego club in Edinburgh for many interesting discussions, especially at the early stage of this work. Special thanks to Kieran Clenaghan and two anonymous referees who have carefully read an earlier version of this paper and suggested many improvements.

References

Backhouse, R., Chisholm, P., and Malcolm, G. (1989) Do-it-youself type theory. *Formal Aspects of Computing* **1** (1).

Böhm, C. and Berarducci, A. (1985) Automatic synthesis of typed λ-programs on term algebras. *Theoretical Computer Science* **39**.

Burstall, R. (1989) *An approach to program specification and development in constructions.* Talk given in Workshop on Programming Logic, Bastad, Sweden.

Burstall, R. and Goguen, J. (1980) The semantics of Clear, a specification language. *Springer-Verlag Lecture Notes in Computer Science* **86**.

Burstall, R. and Lampson, B. (1984) Pebble, a kernel language for modules and abstract data types. *Springer-Verlag Lecture Notes in Computer Science* **173**.

Burstall, R. and McKinna, J. (1991) *Deliverables: an approach to program development in the calculus of constructions.* LFCS report ECS-LFCS-91-133, Dept of Computer Science, University of Edinburgh.

Burstall, R. and McKinna, J. (1992) *Deliverables: a categorical approach to program development in type theory.* LFCS report ECS-LFCS-92-242, Dept of Computer Science, University of Edinburgh.

Church, A. (1940) A formulation of the simple theory of types. *J. Symbolic Logic* **5** (1).

Constable, R. *et al.* (1986) *Implementing Mathematics with the NuPRL Proof Development System,* Prentice Hall.

Coquand, T. and Huet, G. (1988) The calculus of constructions. *Information and Computation* **76** (2/3).

Coquand, T. and Paulin-Mohring, C. (1990) Inductively defined types. *Springer-Verlag Lecture Notes in Computer Science* **417**.

Curry, H. and Feys, R. (1958) *Combinatory Logic*, volume 1, North-Holland Publishing Company.

de Bruijn, N. (1980) A survey of the project AUTOMATH. In: Hindley, J. and Seldin, J. (eds.) *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press.

Ehrig, H., Fey, W. and Hansen, H. (1983) *ACT ONE: an algebraic specification language with two levels of semantics.* Technical Report 83-03, Technical University of Berlin, Fachbereich Informatik.

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

Ehrig, H. and Mahr, B. (1990) *Fundamentals of Algebraic Specification 2: Module specifications and Constraints*, Springer.

Futatsugi, K., Goguen, J., Jouannaud, J.-P. and Meseguer, J. (1985) Principles of OBJ2. *Proc. POPL 85.*

Girard, J.-Y. (1972) *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*, PhD thesis, Université Paris VII.

Goguen, J., Thatcher, J. and Wagner, E. (1978) Abstract data types as initial algebras and the correctness of data representation. In: Yeh, R., (ed.) *Current Trends in Programming Methodology, Vol. 4*, Prentice Hall.

Guttag, J., Horowitz, E. and Musser, D. (1976) Abstract data types and software validation. *Comm. ACM* **21** (12).

Harper, R. and Pollack, R. (1991) Type checking, universe polymorphism, and typical ambiguity in the calculus of constructions. *Theoretical Computer Science* **89** (1).

Hoare, C. (1972) Proofs of correctness of data representation. *Acta Informatica* **1** (1).

Howard, W. A. (1980) The formulae-as-types notion of construction. In: Hindley, J. and Seldin, J. (eds.) *To H. B. Curry: Essays on Combinatory Logic*, Academic Press.

Huet, G. (1987) A calculus with type:type. (Unpublished manuscript)

Jones, C. (1986) *Systematic Software Development using VDM*, Prentice Hall.

Liskov, B. and Zilles, S. (1975) Specification techniques for data abstraction. *IEEE Trans. on Software Engineering* **SE-1** (1)

Luo, Z. (1989) **ECC**, an extended calculus of constructions. In: *Proc. of the Fourth Ann. Symp. on Logic in Computer Science*, Asilomar, California, U.S.A.

Luo, Z. (1990a) *An Extended Calculus of Constructions*, PhD thesis, University of Edinburgh. Also as Report CST-65-90/ECS-LFCS-90-118, Department of Computer Science, University of Edinburgh.

Luo, Z. (1990b) *A problem of adequacy: conservativity of calculus of constructions over higher-order logic.* Technical report, LFCS report series ECS-LFCS-90-121, Department of Computer Science, University of Edinburgh.

Luo, Z. (1991a) A higher-order calculus and theory abstraction. *Information and Computation* **90** (1) 107–137.

Luo, Z. (1991b) Program specification and data refinement in type theory. Proc. of the Fourth Inter. Joint Conf. on the Theory and Practice of Software Development (TAPSOFT). *Springer-Verlag Lecture Notes in Computer Science* **493**. Also as LFCS report ECS-LFCS-91-131, Dept. of Computer Science, Edinburgh University.

Luo, Z. (1992) A unifying theory of dependent types: the schematic approach. Proc. of Symp. on Logical Foundations of Computer Science (Logic at Tver'92). *Springer-Verlag Lecture Notes in Computer Science* **620**. Also as LFCS Report ECS-LFCS-92-202, Dept. of Computer Science, University of Edinburgh.

Luo, Z. (1993) *A Theory of Dependent Types and Computer Science*, The Oxford University Press. (To appear)

Luo, Z. and Pollack, R. (1992) *LEGO Proof Development System: User's Manual.* LFCS Report ECS-LFCS-92-211, Department of Computer Science, University of Edinburgh.

Luo, Z., Pollack, R. and Taylor, P. (1989) *How to Use LEGO: a preliminary user's manual.* LFCS Technical Notes LFCS-TN-27, Dept. of Computer Science, Edinburgh University.

MacQueen, D. (1981) Structures and parameterization in a typed functional language. *Proc. Symp. on Functional Programming and Computer Architecture.*

MacQueen, D. (1986) Using dependent types to express modular structure. *Proc. 13th Principles of Programming Languages.*

Martin-Löf, P. (1975) An intuitionistic theory of types: predicative part. In: Rose, H. and Shepherdson, J. C. (eds.) *Logic Colloquium'73.*

Martin-Löf, P. (1982) Constructive mathematics and computer programming. In: Cohen, L.C. *et al.* (eds.) *Logic, Methodology and Philosophy of Science VI*, Amsterdam, North-Holland.

Martin-Löf, P. (1984) *Intuitionistic Type Theory*, Bibliopolis.

McKinna, J. (1992) *Deliverables: a categorical approach to program development in type theory*, PhD thesis, Department of Computer Science, University of Edinburgh.

Milner, R., Tofte, M. and Harper, R. (1990) *The Definition of Standard ML*, MIT.

Nordström, B. and Petersson, K. (1983) Types and specifications. *Proceedings of IFIP'83* 915–920.

Nordström, B., Petersson, K. and Smith, J. (1990) *Programming in Martin-Löf's Type Theory: an introduction*, Oxford University Press.

Ore, C.-E. (1992) The Extended Calculus of Constructions (ECC) with inductive types. *Information and Computation* **99** (2) 231–264.

Paulin-Mohring, C. (1989) Extracting F^{ω} programs from proofs in the calculus of constructions. *Proc. POPL 89.*

Pollack, R. (1989) *The theory of LEGO.* (Manuscript)

Pollack, R. (1990) Implicit syntax. In: *The Preliminary Proceedings of the 1st Workshop on Logical Frameworks.*

Reus, B. and Streicher, T. (1992) *Lifting the laws of module algebra to deliverables.* Technical report, Ludwig-Maximilians-Universität Münichen, Institut für Informatik.

Reynolds, J. (1974) Towards a theory of type structure. *Springer-Verlag Lecture Notes in Computer Science* **19**.

Sannella, D., Sokolowski, S. and Tarlecki, A. (1990) *Toward formal development of programs from algebraic specifications: Parameterization revisited.* (Draft)

Sannella, D., Sokolowski, S. and Tarlecki, A. (1992) *Toward formal development of programs from algebraic specifications: Parameterization revisited.* LFCS Report ECS-LFCS-92-222, Department of Computer Science, University of Edinburgh.

Sannella, D. and Tarlecki, A. (1987) Extended ML: an institution-independent framework for formal program development. Proc. Workshop on Category Theory and Computer Programming. *Springer-Verlag Lecture Notes in Computer Science* **240** 364–389.

Sannella, D. and Tarlecki, A. (1988a) Specifications in arbitrary institutions. *Information and Computation* **76**.

Sannella, D. and Tarlecki, A. (1988b) Toward formal development of programs from algebraic specifications: implementation revisited. *Acta Informatica* **25**.

Sannella, D. and Wirsing, M. (1983) *A kernal language for algebraic specification and implementation.* Technical Report CSR-155-83, Dept of Computer Science, University of Edinburgh.

Wand, P. (1992) *Functional programming and verification with lego*, Master's thesis, Department of Computer Science, University of Edinburgh.

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

Wirsing, M. and Broy, M. (1989) A modular framework for specification and implementation. TAPSOFT'89. *Springer-Verlag Lecture Notes in Computer Science* **351**.