Program Specification and Data Refinement in Type Theory

Zhaohui Luo
Department of Computer Science, University of Edinburgh
JCMB, KB, Mayfield Rd., Edinburgh EH9 3JZ, U.K.

January 21, 1991

Abstract

We develop a type-theoretic approach to program specification and data refinement and show that a type theory with a strong logical power and nice structural mechanisms 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, [Hoa72, LZ75, GHM76, GTW78, BG80, EFH83, MSV83, EM85, FGJM85, Wir86, Jon86, ST87, ST88a, WB89] among the enormous literature). Various formal abstraction mechanisms (*e.g.*, algebraic specifications) have been studied to provide good methodology and tools which can be used to apply the useful principles of software development like separation of concerns and divide-and-conquer and to guarantee the correctness of programs with respect to their specifications.

Type theories (*e.g.*, Martin-Löf's type theory [ML75, ML84], the Automath type theory [dB80], Nuprl's type theory [C$^+$86], and Coquand-Huet's calculus of constructions [CH88]) were mainly developed for foundation and formalization of mathematics. Since the work by Martin-Löf, it has become known that type theories can also provide basic mechanisms for programming and program specification (*c.f.*, [ML82, NPS90]). For instance, program derivation has been studied in various type theories [NPS90, BCM89, PM89]. However, although it is known that a type theory may be used as a programming and specification language, some important topics concerning modular design and structured specification (for example, 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.

In this paper, we develop a type-theoretic approach to program specification and data refinement and show that a type theory with nice structural mechanisms provides an adequate formalism for both modular design by data refinement and structured specification.

The type theory that we work with in this paper is the Extended Calculus of Constructions (**ECC**) [Luo89, Luo90a]. As a formal system, **ECC** extends the calculus of constructions [CH88] 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 [ML75] by an impredicative universe (higher-order logic). However, different from 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 of 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 [Luo91] and program specification, the latter of which is discussed in this paper (also see [BM90]).

A *specification* in the type theory consists of (a pair of) a type, whose objects are the possible structures (program modules) which 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 about 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) which satisfies 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 (see [TL88, Luo91]) and the notion of 'deliverables' by Burstall (see [BM90]). 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 which expresses the design decisions made in the refinement step. This implementation relation composes vertically (*c.f.*, [BG80]) and hence satisfies the basic requirement for stepwise development of programs.

Based on 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 supports *parameterized specifications*. A notion of implementation between parameterized specifications is defined and it is shown to compose vertically. When a parameterized specification is monotone with respect to the implementation relation between specifications, the property of horizontal composition (*c.f.*, [BG80]) holds and the design principle of divide-and-conquer can be applied as well.

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 one formal system in which specifications, programs and their implementation relationships can be uniformly formalized and discussed. Such an 'internalization' has an 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 like Lego [Pol89, LPT89], 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.) By this, we seem to have gained a good compromise between traditional model-theoretic approaches to specification semantics and the need to implement set theory in order to verify implementation correctness. We relate 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 [ST88b].

In section 2, we briefly introduce the type theory used in this paper. Section 3 discusses specifications, specification of abstract data types, and the notion of abstract implementation. The issues about modular design and specification operations are dealt with in section 4, and parameterized specifications and their implementations are discussed in section 5. The work reported here is in progress; several further research topics are discussed in the conclusion.

2 The Extended Calculus of Constructions

As we have mentioned in the introduction, the type theory **ECC** [Luo89, Luo90a] is a natural combination of Martin-Löf's type theory [ML75] and the calculus of constructions [CH88], based on the idea that there should be clear distinction between the notion of sets (data types) and that of logical formulae. The type system has good proof-theoretic properties (Church-Rosser, strong normalization, decidability and the others, see [Luo89, Luo90a] for details) and a set-theoretic (realizability) model can be found in [Luo91]. **ECC** is implemented in the proof development system Lego [Pol89, LPT89], 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.

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

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

where $\langle\rangle$ is the empty context and $FV(\Gamma)$ is the set of free variables in context Γ. A judgement of the form $\Gamma \vdash a : A$ (or simply $a : A$ when the context is clear or irrelevant[1]) means 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

[1]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.

by the following rule:

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

A *universe* is a type which has (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:

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

That is, we have (intuitively): $Prop \in Type_0 \in Type_1 \in ....$ 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, $Prop \subseteq Type_0 \subseteq Type_1 \subseteq ....$ This type inclusion between universes, together with the intensional equality $\simeq$ (conversion, see below), generates a subtyping relation $\preceq$ between the types,[2] 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 convertible types have the 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 embedded 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

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

and the β-conversion rule $(\lambda x{:}A.b)(a) \simeq [a/x]b$, where $[a/x]$ is the usual substitution operator.

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 $f(a_1)...(a_n)$. □

[2] Since this is not important to understand the rest of this paper, we have intentionally omitted the technical definition of the subtyping relation. See [Luo89, Luo90a] for details.

$$
\begin{array}{rcl}
P_1 \supset P_2 & =_{df} & P_1 \rightarrow P_2 \\
\forall x{:}A.P(x) & =_{df} & \Pi x{:}A.P(x) \\
false & =_{df} & \forall X{:}Prop.X \\
P_1 \ \& \ P_2 & =_{df} & \forall X{:}Prop.\ (P_1 \supset P_2 \supset X) \supset X \\
P_1 \vee P_2 & =_{df} & \forall X{:}Prop.\ (P_1 \supset X) \supset (P_2 \supset X) \supset X \\
\neg P_1 & =_{df} & P_1 \supset false \\
\exists x{:}A.P(x) & =_{df} & \forall X{:}Prop.\ (\forall x{:}A.(P(x) \supset X)) \supset X \\
a =_A b & =_{df} & \forall P{:}A \rightarrow Prop.\ P(a) \supset P(b)
\end{array}
$$

Figure 1: Logical operators

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, this gives an embedded (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 1, where A is an arbitrary type, P_1 and P_2 are arbitrary propositions and P is an arbitrary predicate over A (a *predicate over type* A is a propositional function of type $A \rightarrow Prop$). $=_A$ as defined in Figure 1 is the so-called *Leibniz's equality* (between objects of type A). The Leibniz's equality reflects the basic computational equality (conversion) (that is, two closed terms are Leibniz's equal if and only if they are convertible [Luo90a]) and hence can be used to give adequate specifications of programs for concrete data types. The normalization theorem of the type system entails the consistency of the embedded logic [Luo90a]; furthermore, it is shown in [Luo90b] that this embedded logic is a conservative extension of the intuitionistic higher-order predicate logic (*c.f.*, [Chu40]).

Remark Different from Martin-Löf's type theory [ML75, ML84], in **ECC** types and propositions are *not* identified and there is a distinguishable embedded logic. Every proposition is (lifted as) a type, but *not* vice versa. This gives a conceptual distinction between sets (data types), which reside in the predicative universes, and propositions (logical formulas), which reside in the impredicative universe.[3] This distinction is im-

[3]We can give a slightly different formulation of **ECC** by distinguishing the types with their names,

portant. Propositions should not be understood as arbitrary sets; to represent sets one should use predicative types. (See [Luo90b] for a discussion on this topic.) □

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 $B(a)$. When B is a predicate over A, it expresses the (intuitionistic) subset of the objects of type A which satisfy the property B. 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 have the following introduction and elimination rules

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

and the conversion rules $\pi_i(pair_A(a_1,a_2)) \simeq a_i$ $(i = 1,2)$.

Notations 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 is heavily typed (for decidability reasons), we often use (a,b) rather than $pair_A(a,b)$ 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 $\pi_2(...\pi_2(\pi_2(a))...)$. □

The predicative universes $Type_i$ can be viewed as set universes; the non-propositional types in predicative universes represent sets (or data types). (This is in contrast with the view of coding of data types [BB85] in an impredicative type system like Girard-Reynold's polymorphic λ-calculus [Gir72, Rey74] or the calculus of constructions [CH88].) These predicative universes are supposed to be *open* in the sense as Martin-Löf explains for his type theory [ML75, ML84]. For example, the type of natural numbers can be introduced by adding constants $N : Type_0$, $0 : N$ and $succ : N \to N$, and a recursion operator rec_N with the elimination rule

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

---

which will make this point clearer. This is out of the range of this paper.

and conversion rules $rec_N(c,f)(0) \simeq c$ and $rec_N(c,f)(succ(x)) \simeq f(c, rec_N(c,f)(x))$. Other data types like that of lists or type constructors may also be introduced in a similar way. An important topic which requires further research is how to introduce inductive types in a general way so that the good proof-theoretic properties still hold (see [CPM90, Ore90] where extensions of **ECC** by inductive types are discussed; also see [Luo90c] for a recent proposal using Martin-Löf's W-types).

Concerning about the predicative universes, it is often tedious to always write the level subscripts. A technique has been developed [Hue87, HP89, Pol90] 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 [Pol89, LPT89]. With such a facility, to assume $X{:}Type$ in a context is in some sense equivalent to assume 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 who does the work to avoid 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 Type Theory

It is well-known that in type theories one can specify programs (*c.f.*, [ML82, NP83, NPS90]). For example, in Martin-Löf's type theory, a specification of sorting programs for lists of natural numbers may be defined as follows:

$$Sorting =_{df} \Sigma f{:}List(N) \to List(N).\ \forall l{:}List(N).\ sorted(l, f(l))$$

The adequacy of such specifications (*i.e.*, *Sorting* does specify the sorting programs) is in particular based on the fact that the propositional equality used in them reflects the computational equality, *i.e.*, two closed programs are equal under the propositional equality if and only if they are computationally equal. (The need for this will become clear if one gives a full description of the predicate *sorted* in the above example). In **ECC**, Leibniz's equality (see section 2) can be used to describe the computational equality (conversion) based on an equality reflection result (see [Luo90a]). In Martin-Löf's type theory with weak intensional equality (see [NPS90]), the weak equality can be used to describe computation since it reflects the definitional equality.[4]

[4]In Martin-Löf's type theory with strong extensional equality [ML84], the strong equality is essentially equivalent to the judgemental equality and the latter could be viewed as computational equality. Note that in the discussion here, we have taken a simple view without considering issues like observational equivalence.

Note that specifications like the above directly specify programs for *concrete* data types (built-in in a type theory). Considering (large) program development by stepwise refinement, we are more interested in specifications of abstract data types with loose semantics. Σ-types provide a good mechanism to describe abstract structures and can be used to describe abstract data types. (See [NPS90] for an example of this, where the weak equality in Martin-Löf's type theory is used as the equality of the abstract data type.) However, carefully considering the (loose) semantics of abstract data types and the methodology of data refinement for program development, one will find out that specification of abstract data types has different requirements from those of programs for concrete data types discussed above. In particular, the equality over the abstract type should *not* be specified by a built-in equality like Leibniz's equality in our system (or the weak/strong equality in Martin-Löf's type theories); otherwise, we would not be able to refine a specification of an abstract data type to another in a stepwise way as expected and the best we could do would be to give a concrete realization of the specification.

Therefore, instead of using a built-in equality as the equality for the carrier of an abstract data type, we associate the carrier type with an explicitly specified congruence relation to represent the intended equality. In what follows, after briefly explaining a general setting for specifications and semantics, we show by example how abstract data types can be specified, and then, we define a notion of abstract implementation between specifications and show how abstract refinement works.

3.1 Specifications and abstract data types

A specification in the type theory consists of (a pair of) a type, whose objects are the possible structures (program modules) which 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 $Str[SP]$ of SP-structures, called the structure type of SP, and a predicate $Ax[SP]$ over $Str[SP]$. The set of specifications is described by the following type:*

$$SPEC =_{df} \sum [Str : Type,\ Ax : Str \to Prop]$$

For any type S, we also write $Spec(S)$ for the type 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 (see also a remark in section 3.2). Such a separation is also the idea by Burstall on deliverables [BM90] and that for mathematical theories [Luo91]. $\square$

The semantics of specifications is determined by the type theory, as the following notions of realization and model give.

Definition 3.2 (realizations and models) *Let SP be a specification. A realization (or concrete implementation) of SP is an SP-structure r (i.e., an object of type $Str[SP]$) such that $Ax[SP](r)$ is provable. A model of SP is an SP-structure together with the evidence that it satisfies $Ax[SP]$; the set of SP-models are given by the following type:*

$$Mod(SP) =_{df} \Sigma s{:}Str[SP].Ax[SP](s)$$

SP is called realizable (or consistent) if there exists a realization, i.e., $Mod(SP)$ is non-empty. $\square$

For example, a specification of the sorting programs for lists of natural numbers consists of the structure type $List(N) \rightarrow List(N)$ and the predicate $\lambda f{:}List(N) \rightarrow List(N).\ \forall l{:}List(N).\ sorted(l, f(l))$. A realization is just a sorting program.

We now give an example to explain how abstract data types can be specified. We shall use the following definition in examples:

$$Setoid =_{df} \sum[Dom : Type,\ Eq : Dom \rightarrow Dom \rightarrow Prop]$$

A 'setoid' is a type together with a binary relation over the type. As we already mentioned above, an abstract type is always associated with a binary relation, which is a congruence relation as specified in the axiom part of specifications and is used to represent the intended equality over the abstract type.

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

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

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

$$\begin{array}{rll} Ax[Stack(N)](S) =_{df} & & Cong_{Stack(N)}(Eq) \\ & \& & Eq(pop(empty), empty) \\ & \& & top(empty) =_N 0 \\ & \& & \forall n{:}N \forall s{:}Dom[Stack].\ Eq(pop(push(n,s)), s) \\ & \& & \forall n{:}N \forall s{:}Dom[Stack].\ top(push(n,s)) =_N n \end{array}$$

where $Stack$ abbreviates $Stack[S]$, Eq abbreviates $Eq[Stack[S]]$, and similar for the others; $Cong_{Stack(N)}(Eq)$ stands for the following proposition expressing that the binary relation between stacks is a congruence:

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

□

Note that, in the above example, we only require that the equality Eq between stacks be a congruence. This enables further refinement of the specification by another (for example, arrays of natural numbers, see below). If we had instead used Leibniz's equality, such an abstract refinement would not be possible.

3.2 Data refinement and implementation

A notion of (abstract) implementation is the 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 (see [Hoa72, GTW78]) to the more recent considerations (see, for example, [ST88b, WB89]). We formalize in the type theory a simple notion of refinement (implementation). This notion comes from the consideration of theory morphisms in abstract reasoning [TL88, Luo91] and is similar to the notion of deliverables [BM90].

Definition 3.4 (refinement map and implementation) *Let SP and SP' be specifications. A refinement map from SP' to SP is a functional operation ρ from $Str[SP']$*

to $Str[SP]$,

$$\rho : Str[SP'] \to Str[SP]$$

such that the following satisfaction condition is provable:

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

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

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

Remark Note that refinement maps are incomplete programs and incorporate various design decisions during the process of refinement implementation. One might consider a seemingly simpler and more general notion by defining a refinement map simply as a function of type $Mod(SP') \to Mod(SP)$ with certain proof-irrelevance condition. However, such a notion would not distinguish the computational part of an abstract implementation from its non-computational contents (correctness proofs) as ours does. Such a separation is very important when considering development of programs, since refinement maps (programs) should not contain unnecessary components concerning about (proofs of) implementation correctness. We also believe, at least in practice, that (in an intuitionistic setting) every function comes from an extensional operation and our notion of refinement is general enough and adequate. (See [Bee82] for a relevant discussion on the relationship between functions and extensional operations.) □

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

$$\{\, \rho(m') \mid \exists p.\ (m', p) \in Mod(SP') \,\}$$

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:

$$SP_0 \Longrightarrow_{\rho_1} SP_1 \Longrightarrow_{\rho_2} \ldots \Longrightarrow_{\rho_n} SP_n$$

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

$$r_0 = \rho_1(\ldots(\rho_n(r_n)))$$

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* (*c.f.*, [BG80]).

Proposition 3.5 (vertical composition) *If* $SP \Longrightarrow_{\rho} SP'$ *and* $SP' \Longrightarrow_{\rho'} SP''$*, then* $SP \Longrightarrow_{\rho\circ\rho'} SP''$*, where* $\rho \circ \rho' = \lambda s'':Str[SP'']. \rho(\rho'(s''))$ *is the functional composition of* ρ *and* ρ'*.* □

Remark As a special case of the above proposition, $SP \Longrightarrow SP'$ and $SP' \Longrightarrow SP''$ imply $SP \Longrightarrow SP''$. □

Let us now see a traditional example of refinement — implementing stacks by array-pointer pairs.

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*

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

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

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

where $Cong_{Array(N)}(Eq)$ *is the proposition expressing that the relation* Eq *between arrays is a congruence, similarly defined as* $Cong_{Stack(N)}(Eq)$ *in Example 3.3.*

The refinement map ρ, given any $Array(N)$-structure A, generates the following $Stack(N)$-structure:

$$
\begin{array}{lll}
Dom[Stack[\rho(A)]] & =_{df} & \sum[arr : Dom[Array[A]],\ ptr : N] \\
Eq[Stack[\rho(A)]] & =_{df} & \lambda s, s'{:}Dom[Stack[\rho(A)]]. \\
 & & \quad ptr[s] =_N ptr[s'] \ \& \\
 & & \quad \forall i{:}N.\ i < ptr[s] \supset access(arr[s], i) =_N access(arr[s'], i) \\
empty[\rho(A)] & =_{df} & (newarray[A],\ 0) \\
push[\rho(A)] & =_{df} & \lambda n{:}N \lambda s{:}Dom[Stack[\rho(A)]].\ (assign(n, arr[s], ptr[s]), ptr[s] + 1) \\
pop[\rho(A)] & =_{df} & \lambda s{:}Dom[Stack[\rho(A)]].\ (arr[s], ptr[s] - 1) \\
top[\rho(A)] & =_{df} & \lambda s{:}Dom[Stack[\rho(A)]].\ access(arr[s], ptr[s] - 1)
\end{array}
$$

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). This is possible because we have associated the explicit congruences in the specifications rather than using a fixed equality for the abstract type.

ρ defined above is indeed a refinement map from $Array(N)$ to $Stack(N)$.

Fact 3.7 *$Stack(N) \Longrightarrow_{\rho} Array(N)$.* □

An obviously important issue in a refinement development of programs is 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 [Pol89, LPT89] 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 the machine.

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 like divide-and-conquer and stepwise refinement to decompose and refine the specification until they reach desirable 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 some chief programmers and further implemented separately by others. Secondly, to get a good requirements specification of 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, we discuss in this section modular design and structured specification in our type-theoretic approach (parameterized specifications are discussed in section 5). In particular, we consider various specification operations[5] which can be used either in structured 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 [BG80]).

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 implemented separately and their realizations can be 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 the independency for further refinements.)

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

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

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

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

The (infix) specification operation $\otimes$ is of type $SPEC \to SPEC \to SPEC$. □

The idea to use the above specification operation to decompose a specification SP into several (say two) independent specifications (SP_1 and SP_2) by certain design strategy is

[5] A *specification operation* is a function which 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.

to consider a refinement step of the following form:

$$SP \Longrightarrow_\rho SP_1 \otimes SP_2$$

where the refinement map $\rho : Str[SP_1 \otimes SP_2] \rightarrow Str[SP]$ 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 $SP_i \Longrightarrow_{\rho_i} SP_i'$ $(i = 1, 2)$, then $SP_1 \otimes SP_2 \Longrightarrow_{\rho_1 \otimes \rho_2} SP_1' \otimes SP_2'$, where $\rho_1 \otimes \rho_2 = \lambda s'{:}Str[SP_1' \otimes SP_2'].(\rho_1(\pi_1(s')), \rho_2(\pi_2(s')))$.* □

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.[6] For example, one may decompose a specification of a parser into several specifications including $AbsSyn$ for an abstract syntax tree generator and $SymTab$ for management of the symbol table; these later two specifications both use symbols and symbol management functions which are 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 by using Σ-types by considering the following specification operation $\boldsymbol{\Sigma}$, which has $\otimes$ above as a special case.

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

$$Str[\boldsymbol{\Sigma}(SP, P)] =_{df} \Sigma s{:}Str[SP].Str[P(s)]$$

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

$$Ax[\boldsymbol{\Sigma}(SP, P)](s') =_{df} Ax[SP](\pi_1(s')) \ \& \ Ax[P(\pi_1(s'))](\pi_2(s'))$$

$\boldsymbol{\Sigma}$ is of type $\Pi SP{:}SPEC\,\Pi P{:}Str[SP] \rightarrow SPEC.\ SPEC$. □

[6] Much attention has been paid to such structure sharing in the design of both programming languages (*e.g.*, Standard ML [Mac81, MTH90] and Pebble [BL84]) and specification languages (*e.g.*, Clear [BG80] and Extended ML [ST87]).

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

1. $SP \Longrightarrow_\rho SP'$, *and*

2. δ *is a function of type* $\Pi s'{:}Str[SP'].\ Str[P'(s')] \to Str[P(\rho(s'))]$ *such that* $\forall s'{:}Str[SP'].\ P(\rho(s')) \Longrightarrow_{\delta(s')} P'(s')$ *is provable,*[7]

then,

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

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

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

$$SP \Longrightarrow_\rho \sum(SP_0, P)$$

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 to decompose SP into SP_0 and (parameterized specification) P which are independent of each other (see section 5). Another way to look at the further refinement of $\sum(SP_0, P)$ is to consider SP_0 and the following specification SP':

$$Str[SP'] =_{df} \Pi s_0{:}Str[SP_0].\ Str[P(s_0)]$$

$$Ax[SP'](f) =_{df} \forall m{:}Mod(SP_0).Ax[P(f(\pi_1(m)))](f(\pi_1(m)))$$

In other words, we proceed to implement $SP_1(s_0)$ and $SP_1(s_0)$ independently, assuming that s_0 is an arbitrary realization of SP_0. Note that SP_0 and SP' are independent of each other and they 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 SP' to r_0.

Warning: 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 specification whose structure type is $\Sigma x{:}Prop.x$ into two specifications with $Prop$ and $\Pi x{:}Prop.x$ as structure types, respectively, produces an inconsistent specification (the second one) since $\Pi x{:}Prop.x$ has no object in any consistent context in the type theory. In any stage of refinement development, the programmer

[7] 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.

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 about 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 $P(r_0)$. $\square$

4.2 Constructors and selectors

Since the work by Burstall and Goguen on specification language Clear [BG80], 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 which may often be used in structuring specifications:

- $Join_S$: it 'puts together' the axiomatic parts of two specifications over the same structure type S: if $Str[SP] = Str[SP'] = S$, then

  $$\begin{aligned} Str[Join_S(SP, SP')] &=_{df} S \\ Ax[Join_S(SP, SP')](s) &=_{df} Ax[SP](s) \ \&\ Ax[SP'](s) \end{aligned}$$

  $Join$ is of type

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

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

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

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

  $Extend$ is of type

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

There are various specification operations which 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*[8] 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 Con_ρ and Sel_ρ are defined as follows:*

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

$$\begin{array}{rcl} Str[Con_\rho(SP')] & =_{df} & S \\ Ax[Con_\rho(SP')](s) & =_{df} & \exists s'{:}S'.\ Ax[SP'](s')\ \&\ \rho(s') =_S s \end{array}$$

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

$$\begin{array}{rcl} Str[Sel_\rho(SP)] & =_{df} & S' \\ Ax[Sel_\rho(SP)](s') & =_{df} & Ax[SP](\rho(s')) \end{array}$$

□

Intuitively, the constructor Con_ρ applied to specification SP' constructs as its realizations the images of ρ over the SP'-realizations, while the selector Sel_ρ 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:

- The operation $Join_S$ can be defined as

  $$Join_S(SP, SP') =_{df} Sel_d(SP \otimes SP')$$

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

- The operation $Extend$ can be defined as

  $$Extend(SP, Ext\_Str, Ext\_Ax) =_{df} Join_S(Sel_{\pi_1}(SP), (S, Ext\_Ax))$$

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

[8] The name 'constructor' comes from the similarity of this class of specification operations with Sannella and Tarlecki's notion of constructor. See section 4.3.

Another example is that constructors can be used to play a role of 'renaming' and information hiding similar to the operation *derive* in specification language ASL [SW83, Wir86]. *derive* in ASL is based on a signature morphism from the signature of the resulting specification to that of the argument specification. Such an 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, $Con_\rho(SP)$ corresponds to *derive* SP *from* σ. Similarly, the operation **translate** (see [ST88a]) can be simulated as selectors.

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

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

- *Realizability:*
  1. *if* $SP' : Spec(S')$ *is realizable, so is* $Con_\rho(SP')$*;*
  2. *if* $Sel_\rho(SP)$ *is realizable, so is* SP*.*
- *Monotonicity:*
  1. *for* $SP_1', SP_2' : Spec(S')$*,* $SP_1' \Longrightarrow SP_2'$ *implies* $Con_\rho(SP_1') \Longrightarrow Con_\rho(SP_2')$*;*
  2. *for* $SP_1, SP_2 : Spec(S)$*,* $SP_1 \Longrightarrow SP_2$ *implies* $Sel_\rho(SP_1) \Longrightarrow Sel_\rho(SP_2)$*.* □

4.3 Constructor/selector implementation

The constructor operations are very similar in spirit to the notion of constructors (functions between algebra classes) introduced in [ST88b], 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 which implements SP and goes on to implement the argument specifications of the constructors used in this step. Such a method applies to 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* $Str[SP'] \to Str[SP]$.

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

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

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

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

Proof The first and the last statements are intensionally equal (convertible) and they are logically equivalent to the second. □

Constructors and selectors are essentially dual operations. They suggest different methodologies of refinement development of programs. The notion of selector implementation suggests a bottom-up approach to refinement, while that of constructor implementation suggests a top-down approach, which is usually taken.

5 Parameterized Specification

Parameterization is a powerful abstraction tool both for modular design and for structured specification. 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 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 such a specification form may suggest: one is to decompose it into two independent specifications SP_0 and SP', where SP' is a specification of parameterized program modules; another is to directly use 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 and usefulness of parameterized specifications in modular design and structuring specifications can not only be explained by a simple example. We would not elaborate this in this paper. Among the large amount of literature on this are [BG80, SW83, EM85] and in particular [SST90] where a recent nice account on this issue 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:

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

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 knids 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

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

where $Stack(X, x, R)$ is the same as $Stack(N)$ except that N, 0 and $=_N$ are replaced by X, x and R, respectively. (Depending on different intentions of how **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 the 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 it normally means by parameterized specification in the algebraic approach to specifications (*c.f.*, Clear [BG80] 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 something to say before we give the example. A parameterized specification $STACK$ extending specifications by stacks can not 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, which has only one sort and one constant of the sort (see [BG80] 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.[9] In other words, we need to talk about the 'components' of the structure type of specifications. A way to do this is 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

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

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

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

is of type $Elem(Array(N))$ and may be used in 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 which extends the 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.*

- *Ext_Str_Stack is a function of type $\Pi S{:}Type.\ Elem(S) \to S \to Type$.*

  Given a type S, a function Elem of type $Elem(S)$ and an object s of type S, $Ext\_Str\_Stack(S, Elem, s)$ is defined to be the same as the structure type of $Stack(N)$ in Example 3.3 except that we replace N and $=_N$ by $Dom[X[Elem(s)]]$ and $Eq[X[Elem(s)]]$, respectively.

- *Ext_Str_Stack is function of type*

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

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

[9] Note that there is an important difference between the algebraic approach and a type-theoretic approach. In the former, parameter specifications, which play a role of 'type', are at the same level with the actual specification arguments. In a type theory, this can not be the case.

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

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

which is of type

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

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

$$STACK(Str[Array(N)], ind\_array, Array(N))$$

□

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 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*

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

such that the following satisfaction condition is provable:

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

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

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 [SW83]. Note that the polymorphism and type dependency in type theory gives a nice way to express a (uniform) family of refinement maps as a single polymorphic function. One may have noticed the similarity of this definition with the notion of natural transformation between functors in category theory. Although the above definition is already rather general, one may further consider implementation between two parameterized specifications with possibly different parameter types. We do not expand this discussion 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 $\delta \bullet \delta' = \lambda s{:}Par.\ \delta(s) \circ \delta'(s)$.* □

Example 5.4 *Following example 5.1, we can similarly define a parameterized specification (c.f., Example 3.6)*

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

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

There are two kinds of parameter types which 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.*, $Spec(S)$ or $SPEC$). In the latter case, it is important to consider the property of *horizontal composition* of the implementation relation [BG80], since it guarantees 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 $Spec(S)$. A parameterized specification P of type $Par \to SPEC$ is monotone if and only if there is a function*

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

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

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

1. *$SP \Longrightarrow_{\rho_0} SP'$,*
2. *$P \Longrightarrow_{\delta} P'$, and*
3. *P or P' is monotone.*

Proof Define

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

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 $P_0 \Longrightarrow_{\delta_1} ... \Longrightarrow_{\delta_n} P_n$ and $SP_0 \Longrightarrow_{\rho_1} ... \Longrightarrow_{\rho_n} SP_n$. Doing so, we do not need to verify the monotonicity of more than one of the parameterized specifications, since monotone parameterized specifications can only be implemented by monotone ones.

Proposition 5.7 *Let P and P' be parameterized specifications over Par. Suppose $P \Longrightarrow_{\delta} P'$. Then, if the parameterized specification P is monotone, so is P', and vice versa.*

Proof Suppose P is monotone via. f. Then P' is monotone via. the following function:

$$g =_{df} \lambda A, B{:}Par \lambda \rho{:}Str[B] \to Str[A].\ \delta(A) \circ f(A, B, \rho) \circ \delta(B)$$

The other direction is similar. □

Finally, we remark that the above definition of monotonicity and proposition for horizontal composition can easily be generalized to the case where parameterized specifications have more than one specification as arguments (Par is of the form $Par_1 \times ... \times Par_n$ where Par_i is either $SPEC$ or $Spec(S_i)$).

6 Conclusions

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 is that we have been able to formalize internally the notions like implementation in the type theory, and this enables us to use an implementation of the type theory (Lego) to develop the refinement maps (programs) and the correctness proofs of implementations. Thus, by means of type theory – a more restricted formalism compared with set theory, we have gained a good compromise between the model-theoretic approaches to specification semantics and implementation, which usually causes difficulties in finding suitable proof development supports, and pure axiomatic approaches which could not talk directly about model sets (and may suffer from a complicated notion of implementation, for example).

Discussions in this paper have omitted an important aspect about 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[ST88b]) can be similarly dealt with here by introducing the following specification operation:

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

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

and the notion of abstractor implementation would be given as:

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

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

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

Among the related work, the work by Taylor, Pollack and the author on abstract reasoning [TL88, Luo91] and Burstall's idea on deliverables [BM90] are most influential on this work (in particular, the basic notion of implementation between specifications). Sannella, Sokolowski and Tarlecki [SST90] have recently proposed their ideas of higher-order parameterization and are working on a specification formalism [ST90] which is based on ASL and a type system involving semantical inference. The author has appreciated very much their argument on the difference between parameterized specifications and specifications of program modules and this influences the careful distinction of these two in the final versoin of this paper. Our idea of using explicit congruence relations in specifications of abstract data types seems to be related to some similar ideas which appeared in algebraic specifications (see [WB89] and [MSV83] for example). The relationship of our approach to that in algebraic specifications is not clear yet.

Finally, there is a very interesting relationship between the researches in program development and in proof development systems which are developed for theorem proving. For example, the refinement proof development style in Lego has many common features compared with program development. A full-scale discussion on this is out of the range of this paper.

Acknowledgements Thanks to Rod Burstall for many suggestions, Don Sannella and Ed Kazmierczak for interesting discussions, and the members in the Lego club in Edinburgh for their comments on this work.

References

[BB85] C. Böhm and A. Berarducci. Automatic synthesis of typed λ-programs on term algebras. *Theoretical Computer Science*, 39, 1985.

[BCM89] R. Backhouse, P. Chisholm, and G. Malcolm. Do-it-youself type theory. *Formal Aspects of Computing*, 1(1), 1989.

[Bee82] M.J. Beeson. Problematic principles in constructive mathematics. *Logic Colloquium'80*, 1982.

[BG80] R. Burstall and J. Goguen. The semantics of Clear, a specification language. *Lecture Notes in Computer Science*, 86, 1980.

[BL84] R. Burstall and B. Lampson. Pebble, a kernel language for modules and abstract data types. *Lecture Notes in Computer Science*, 173, 1984.

[BM90] R. Burstall and J. McKinna. Deliverables: an approach to program development in the calculus of constructions. In the preliminary Proceedings of the 1st Workshop on Logical Frameworks, 1990.

[C$^+$86] R.L. Constable et al. *Implementing Mathematics with the NuPRL Proof Development System*. Pretice-Hall, 1986.

[CH88] Th. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2/3), 1988.

[Chu40] A. Church. A formulation of the simple theory of types. *J. Symbolic Logic*, 5(1), 1940.

[CPM90] Th. Coquand and Ch. Paulin-Mohring. Inductively defined types. *Lecture Notes in Computer Science*, 417, 1990.

[dB80] N.G. de Bruijn. A survey of the project AUTOMATH. In J. Hindley and J. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, 1980.

[EFH83] H. Ehrig, W. Fey, and H. Hansen. ACT ONE: an algebraic specification language with two levels of semantics. Technical Report 83-03, Technical University of Berlin, Fachbereich Informatik, 1983.

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

[FGJM85] K. Futatsugi, J. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. *Proc. POPL 85*, 1985.

[GHM76] J.V. Guttag, E. Horowitz, and D.R. Musser. Abstract data types and software validation. *Comm. ACM*, 21(12), 1976.

[Gir72] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[GTW78] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. Abstract data types as initial algebras and the correctness of data representation. In R. Yeh, editor, *Current Trends in Programming Methodology, Vol. 4*. Prentice Hall, 1978.

[Hoa72] C.A.R. Hoare. Proofs of correctness of data representation. *Acta Informatica*, 1(1), 1972.

[HP89] R. Harper and R. Pollack. Type checking, universe polymorphism, and typical ambiguity in the calculus of constructions. *Theoretical Computer Science*, 1989. to appear.

[Hue87] G. Huet. A calculus with type:type. unpublished manuscript, 1987.

[Jon86] C.B. Jones. *Systematic Software Development using VDM*. Prentice-Hall, 1986.

[LPT89] Z. Luo, R. Pollack, and P. Taylor. *How to Use LEGO: a preliminary user's manual.* LFCS Technical Notes LFCS-TN-27, Dept. of Computer Science, Edinburgh University, 1989.

[Luo89] Z. Luo. **ECC**, an extended calculus of constructions. In *Proc. of the Fourth Ann. Symp. on Logic in Computer Science*, Asilomar, California, U.S.A., June 1989.

[Luo90a] Z. Luo. *An Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1990. Also as Report CST-65-90/ECS-LFCS-90-118, Department of Computer Science, University of Edinburgh.

[Luo90b] Z. Luo. 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, 1990.

[Luo90c] Z. Luo. A unifying theory of dependent types. manuscript., 1990.

[Luo91] Z. Luo. A higher-order calculus and theory abstraction. *Information and Computation*, 90(1):107–137, 1991.

[LZ75] B. Liskov and S. Zilles. Specification techniques for data abstraction. *IEEE Trans. on Software Engineering*, SE-1(1), 1975.

[Mac81] D.D. MacQueen. Structures and parameterization in a typed functional language. *Proc. Symp. on Functional Programming and Computer Architecture*, 1981.

[ML75] P. Martin-Löf. An intuitionistic theory of types: predicative part. In H.Rose and J.C.Shepherdson, editors, *Logic Colloquium'73*, 1975.

[ML82] P. Martin-Löf. Constructive mathematics and computer programming. In L.J. Cohen et al, editor, *Logic, Methodology and Philosophy of Science VI*, Amsterdam, 1982. North-Holland.

[ML84] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.

[MSV83] T.S.E. Maibaum, M.R. Sadler, and P.A.S. Veloso. Logical implementation, 1983.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT, 1990.

[NP83] B. Nordström and K. Petersson. Types and specifications. *Proceedings of IFIP'83*, pages 915–920, 1983.

[NPS90] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory: an introduction.* Oxford University Press, 1990.

[Ore90] C.-E. Ore. The Extended Calculus of Constructions (**ECC**) with inductive types. To appear in Information and Computation, 1990.

[PM89] Ch. Paulin-Mohring. Extracting F^ω programs from proofs in the calculus of constructions. *Proc. POPL 89*, 1989.

[Pol89] R. Pollack. The theory of LEGO. manuscript, 1989.

[Pol90] R. Pollack. Implicit syntax. In the preliminary Proceedings of the 1st Workshop on Logical Frameworks, 1990.

[Rey74] J.C. Reynolds. Towards a theory of type structure. *Lecture Notes in Computer Science*, 19, 1974.

[SST90] D. Sannella, S. Sokolowski, and A. Tarlecki. Toward formal development of programs from algebraic specifications: Parameterization revisited. draft, 1990.

[ST87] D. Sannella and A. Tarlecki. Extended ML: an institution-independent framework for formal program development. *Proc. Workshop on Category Theory and Computer Programming, LNCS 240*, pages 364–389, 1987.

[ST88a] D. Sannella and A. Tarlecki. Specifications in arbitrary institutions. *Information and Computation*, 76, 1988.

[ST88b] D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: implementation revisited. *Acta Informatica*, 25, 1988.

[ST90] D. Sannella and A. Tarlecki. A kernel specification formalism with higher-order parameterization. Draft, 1990.

[SW83] D.T. Sannella and M. Wirsing. A kernal language for algebraic specification and implementation. Technical Report CSR-155-83, Dept of Computer Science, University of Edinburgh, 1983.

[TL88] P. Taylor and Z. Luo. Theories, mathematical structures and strong sums. manuscript, December 1988.

[WB89] M. Wirsing and M. Broy. A modular framework for specification and implementation. *TAPSOFT'89, LNCS*, 351, 1989.

[Wir86] M. Wirsing. Structured algebraic specifications: a kernel language. *Theoretical Computer Science*, 42:123–249, 1986.