To appear in: *Proc. Joint Conf. on Theory and Practice of Software Development (TAPSOFT)*, Brighton, April 1991 (Springer Lecture Notes in Computer Science).

Structuring Specifications in-the-Large and in-the-Small: Higher-Order Functions, Dependent Types and Inheritance in SPECTRAL[1]

Bernd Krieg-Brückner, Donald Sannella [2]
FB3 Mathematik und Informatik, Universität Bremen
{bkb, dts}@Informatik.Uni-Bremen.DE

SPECTRAL is an experiment in specification language design that tries to maintain compactness in spite of a number of orthogonal concepts. The design is based on Extended ML and PROSPECTRA, generalising and extending both approaches. Of particular concern are the means for structuring specifications and programs in-the-large and in-the-small. The language includes constructs for defining general higher-order functions yielding specifications or program modules. Concepts of subtyping and (object-oriented) inheritance are included to support the specification development process and to enhance re-usability. Much of the power of the language comes from the use of dependent types.

1. Introduction

1.1. Need for Structuring Concepts

Algebraic methods for specifying data types and functions are by now relatively well developed (see e.g. [COMPASS 91] for a comprehensive list of references). It has long been recognised (see e.g. [BG 77]) that large specifications are unmanageable unless they are built in a structured fashion from smaller specifications using specification-building operations of various kinds and mechanisms for parameterisation and modularisation of specifications. All algebraic specification languages developed in the past ten years have provided such structuring mechanisms; differences between languages concern mainly the particular mechanisms chosen. The structure chosen by a specifier incorporates intangible aspects of the specifier's knowledge of the problem being specified and of the relationships between parts of the problem that is essential to the understanding and use of the specification. For this reason, it is vital for a specification language to provide flexible, powerful and natural mechanisms for structuring specifications.

[1] The work reported in this paper has been partially supported by the Commission of the European Communities in the ESPRIT Basic Research Working Group COMPASS, and a grant from the UK Science and Engineering Research Council

[2] current address: LFCS, University of Edinburgh, Edinburgh EH9 3JZ, Scotland (dts@lfcs.ed.ac.uk)

The perceived need for structure in specifications is related to (but subtly different from) the need for modular structure in programs. In the Extended ML wide-spectrum specification / programming language [ST 85, 86], [San 90], the modularisation mechanisms of the Standard ML programming language [MacQ 86a] (cf. [MTH 90]) are used to structure both finished programs, their specifications, and unfinished programs under development. There has also been some use of inheritance [CO 88] such as that used in object-oriented programming languages to structure specifications. The tendency in both programming language and specification language design seems to be towards providing structuring mechanisms with increasingly more power. This tendency is counterbalanced by the desire for languages (like Standard ML) in which static well-formedness checks can be carried out automatically.

1.2. Design Goals of SPECTRAL

The design of a specification language is presented that combines a number of desirable language concepts for program development in-the-small and in-the-large. A distinction is made between "small" objects (*values*) and "large" objects (*structures,* corresponding to many-sorted algebras). Uniformity is achieved by having one notion of typing and subtyping for small and large objects (*types* of values and *classes* of structures) and a general notion of *function* between all four kinds of items: values, types, structures and classes; most notably, these functions can be of *higher order*. Making a distinction between algebras and specifications as sets of algebras (structures and classes, resp.), and providing higher-order functions over (combinations of) these, gives a very general framework to express structuring of specifications on the one hand and structuring of parameterised implementation schemata on the other [SST 90]; subtyping allows the expression of inheritance as a further way of structuring. This satisfies the needs for different ways of *structuring* generalised *specifications*, in particular at the requirement stage, and generic *implementations*, ultimately a particular software system. The structuring of individual phases of the development process is also supported this way, and there is hope that the necessary proof obligations associated with the development process can also be broken down into more manageable pieces. Potentially the most important consequence of the resulting structure is enhanced re-usability and adaptability of individual specifications, implementations, proofs, etc.

One motivation for the design of a language with a rather rich combination of concepts (such as subtyping, constraints and dependent types, higher-order functions etc.) is the help these concepts give during the processes of devising a requirement specification and of formal development of programs from such specifications; they aid the detailed representation of program properties and their relationships, and they structure and facilitate proofs of them. It is worth emphasising that the needs for structuring mechanisms are different at the requirement specification and at the design specification (or implementation) level: in short, structuring specifications is different from structuring programs. Furthermore, the structure of a requirement specification is often not reflected in the structure of the ultimate implementation; see [FJ 90] where this point is convincingly made with the help of a realistic concrete example.

A starting point for the design was to combine the mutually complementary experiences of Extended ML (for aspects of development in-the-large, see [ST 85, 86], [San 90]) and PROSPECTRA (for aspects of development in-the-small, and transformational development with associated support tools, see e.g. [Krie 89, 90, 91]) from a language design, semantics, development methodology, proof methodology, and support system point of view. Both approaches are extended and generalised; the Extended ML theory of development of modular programs [ST 89] should be adaptable to this language with Standard ML (SML) (see e.g. [Har 86], [Wik 87], [Reade 89], [MTH 90]) as a natural target language for development, and the present PROSPECTRA methodology and system should be adaptable to this design.

Ideally, simplicity should not be compromised in order to include a host of potentially interesting concepts. It is not clear whether this design goal has been (or can be) met. The syntax is quite compact and orthogonal, but all semantic consequences have not been investigated yet. It is foreseen that, although there is (and must be) a uniform semantic framework (the whole "spectrum"), the user may wish to deal with one (or a combination) of the "spectral lines" only, i.e. a particular sublanguage (e.g. without subtypes, higher-order or partial functions), with simplified semantics for easier learning, but also for the use of specialised tools. This calls for a structuring of the semantic description (e.g. using institutions [GB 84], [ST 86]; in-the-large versus in-the-small as in [MTH 90]) and even derived semantics for sublanguages that are simpler.

1.3. Organisation of the Paper

The purpose of this paper is not to present a finished solution but to spawn discussion. The major elements of the design are presented by means of illustrative examples in §2 to §5, focussing, in this paper, on the less usual aspects of structuring in-the-large; §6 gives a short summary of the syntax, predefined operators and some other notational issues.

2. Two Levels of Typing, Uniform Concept of Function

2.1. Four Kinds of Items, Two Levels of Typing

There are four kinds of items ("first-class objects") in the language, with a uniform syntax for definitions (id: kind or id: kind = exp, resp.), cf. §6.1:

- values
- types (or sorts) — *sets of values*
- structures (or algebras)
- classes (or specifications) — *sets of structures*

Typing is analogous at the level of types and classes, re-using inference mechanisms and typing algorithms at both levels; the conceptual uniformity makes the powerful concepts easier to grasp. There is a strict separation between the levels of values/types and structures/classes as in SML and Extended ML: structures are not values and classes are not types, although values and types can be viewed for some purposes as degenerate structures. This separation between levels is necessary to enable static type checking. Although other features rule out static type checking for the full language (see e.g. §5.1), it is nevertheless desirable to retain the possibility of static "rough type" checking in the sense of [SST 91] to enable specifications to be checked easily for structural well-formedness.

2.2. Specifications and Program Modules: Classes and Structures

Consider some simple examples of classes:

```
ELEM:   class =
  ⌈  T:  type ⌋

PO:      class =
  ⌈  T:      type
     ≤:      T × T —> bool
  for all x, y, z: T =>
     x ≤ x,    x ≤ y ∧ y ≤ x → x = y,    x ≤ y ∧ y ≤ z → x ≤ z ⌋

MONOID: class =
  ⌈  M:     type
     n:     M
     ⊗:     M × M —> M
  for all x, y, z: M =>
     n ⊗ x = x,  x ⊗ n = x,   (x ⊗ y) ⊗ z  =  x ⊗ (y ⊗ z) ⌋
```

The class ELEM just gives a signature with one type definition; it is loose in the sense of admitting all structures with at least one appropriately-named type component as models. The signature of the class PO contains two components: a type definition for T and a binary relation on T. In addition, axioms are given to characterise the relation as a partial order. All structures having a signature with a type named T conform to ELEM. In particular, any structure of class PO conforms to the class ELEM (see also §3.1); this will, for example, be used in parameter passing below. Similar to PO, the class MONOID specifies properties of operations to be used in other specifications.

A class denotes a set of algebras, namely all those algebras over the given signature (not just the term-generated ones) satisfying the given axioms (actually, it denotes a proper class; we will ignore such foundational niceties here). Thus a class is a loose requirement specification describing a set of structures; see also §3.2 below.

The class NATS below defines the natural numbers (in a loose fashion). Zero and Pos are subtypes of Nat. Assuming that int is a pre-defined visible type with the usual operations, then NATSasInts defines a structure (i.e. an algebra) that is a model of the given class NATS (the notation for constrained domains in Zero and Pos is described in §5.1). Note that the body of a structure contains *definitions* whose semantics defines objects uniquely (for example giving the least fixpoint for a recursively defined function).

Strictly speaking, structures do not always denote "complete" algebras, in the sense that structures may contain functions over types which do not belong to the structure but are taken from the context (e.g. in §4.1: if X: PO and L: LISTS2 X, then Sort X L contains a function sort over a type List that belongs to L but not to Sort X L). Furthermore, structures may themselves have structures as components (but the resulting hierarchy must of course be acyclic). Corresponding comments also apply to classes. The familar word "algebra" will continue to be used in the rest of the paper, in spite of this inaccuracy.

```
NATS:   class =
  ⌈ Nat:          type
    Zero, Pos:    type  ≤  Nat
    zero:         Zero
    succ:         Nat —>  Pos
    pred:         Pos —>  Nat
    for all x: Nat =>
       pred (succ  x)  =  x              ⌋

NATSasInts:   NATS =
  ⌈ Nat       =  int
    Zero      =  (n: int || n = 0)
    Pos       =  (n: int || n > 0)
    zero      =  0
    for all n: Nat =>
       succ (n)   =  n + 1
       pred (n)  =  n – 1    ⌋
```

2.3. Uniform Concept of Function

The syntax allows functions in all combinations of the four kinds of items above, notably also higher-order functions. Some of these are well-known such as functions from values to values and from structures to structures (SML functors, parameterised programs [Gog 84] or, in a slightly different setting, generics in Ada); some are desirable extensions w.r.t. SML (*higher-order* functions from structures to structures or from structures to *classes*); some are a bit unusual at first (e.g. from structures to values or types); and some are intentionally not user-definable, such as functions from classes to classes, although pre-defined operations of this kind are provided in the language (—>, +). The examples in the sequel show that higher-order functions from structures to classes are particularly useful to act as "parameterised specifications".

2.4. Parameterised Classes

The simplest example of a function yielding a class is a classical parameterised specification such as for lists and sets below (X.T denotes selection of the T component of the structure X).

Semantically, both LISTS and LSETS denote functions which, given an algebra X in the set denoted by ELEM, produce the set of algebras denoted by the respective body under that interpretation of the formal parameter. These functions are persistent (the parameter is protected) *by definition* as in Extended ML [ST 89] (cf. algebra modules in OBSCURE [LL 88]): there is no way for the body to alter models of the parameter, even if the body includes axioms that are inconsistent with those in the parameter specification (in this case, the set of algebras produced as a result is simply empty).

We observe that for any structure X: ELEM, the two classes LISTS X and LSETS X share a common signature (up to renaming) and the common properties of a monoid; such commonalities will be further exploited through structuring by inheritance, see §3.4. Higher-order functions will be discussed in §4. Note that some specifications in this paper are *loose* and admit more than just the models that their names suggest, for example LSETS admits not only true sets, but also multi-sets, ordered sets etc.; see §3.4.

```
LISTS:  (X: ELEM) —> class =
  ⌈ List:     type
    eList:    type  ≤ List
    neList:   type  ≤ List
    empty:    eList
    single:   X.T —> neList
    ++:       neList × List —>   neList
    ++:       List × neList—>    neList
    ++:       List × List —>     List
    for all x, y, z: List =>
       empty ++ x = x,  x ++ empty = x,   (x ++ y) ++ z  =  x ++ (y ++ z)
    head:     neList —> X.T
    tail:     neList —> List
    for all e: X.T; b: List =>
       head ((single  e) ++ b) = e,  tail ((single  e) ++ b) = b  ⌋
```

```
LSETS:  (X: ELEM) —> class =
  ⌈ Set:      type
    eSet:     type  ≤ Set
    neSet:    type  ≤ Set
    empty:    eSet
    single:   X.T —> neSet
    ∪:        neSet × Set —>   neSet
    ∪:        Set × neSet —>   neSet
    ∪:        Set × Set —>     Set
    for all x, y, z: Set =>
       empty ∪ x = x,  x ∪ empty = x,   (x ∪ y) ∪ z  =  x ∪ (y ∪ z)
    ∈:        X.T ×  Set —> bool
    for all d, e: X.T; a, b: Set =>
       e ∈  empty = false,       d ∈  single  e  <–>  d = e,       e ∈  (a ∪ b)  =  (e ∈  a) ∨ (e ∈  b)
⌋
```

3. Inheritance

3.1. Subclasses

The subclass relationship corresponds to subset on the semantical level: a class A is a subclass of a class B iff all models of A are admissible models of B (possibly with appropriate restriction of the signature, see §6.3). In the examples above, PO is a subclass of ELEM, written PO ≤ ELEM, since the signature of ELEM is included in that of PO and there are only *additional* operations and axioms in PO w.r.t. ELEM. In general, the subclass predicate ≤ (also the class membership predicate **in**) generates a structural condition (signature inclusion) that can be checked statically as in SML, and a verification condition (model set inclusion) that can often be inferred statically from the way in which the classes / structures are constructed. Before we discuss the methodological aspects, let us look at some more examples.

```
PO1:    class =
   ELEM  +
   ⌈  ≤:      T × T —> bool
   for all x, y, z: T =>
      x ≤ x,    x ≤ y ∧ y ≤ x → x = y,    x ≤ y ∧ y ≤ z → x ≤ z ⌋
```

The example PO1 is a little contrived, but it illustrates that the subclass relationship can be expressed syntactically by using the class union operator + (also by using the subclass predicate ≤ or the class membership predicate **in**). The subclass relationship can now be statically inferred and recorded for later use (as for subsort/subtype inferencing); note that multiple inheritance is possible. The most common use of the operation + is to perform a mere enrichment of a class, i.e. addition of operations to the signature or of axioms. Note that the items defined in the left hand operand of + are visible in the right hand operand (so the definition of ≤ may refer to T).

A more interesting example is the definition of integers in terms of natural numbers. There are many ways to do this; one way is shown in INTS, where the signature of NATS is extended by a new type Int and additional types for the operations pred and succ (in the presence of an order-sorted approach for function signatures, cf. §5). All models of INTS are admissible models for NATS, thus INTS ≤ NATS. Note that the subtype relationship is in the opposite direction: Nat ≤ Int since all values of Nat are admissible values of Int. There is a derived subtype/subclass relationship for functions, see §4.5.

```
INTS:   class =
   NATS  +
   ⌈  Int:      type  ≥  Nat
      pred:     Int —> Int
      succ:  Int —> Int
   for all x: Int =>
      pred  (succ x)  =  x,      succ (pred  x)  =  x     ⌋
```

Parameterised classes can be similarly put into a subclass relationship; more precisely, the relationship can be inferred for the functions applied to appropriate actual parameters: thus, for example, LISTS2 X ≤ LISTS X for any X: ELEM (nat is assumed to be a predefined subtype of int).

```
LISTS2:     (X: ELEM) —> class =
   LISTS  X  +
   ⌈  freq:      X.T —> List —> nat
   for all x, y, z: List =>  …  ⌋

ORDSETS:     (X: PO) —> class =
   LSETS  X  +
   ⌈  min:   neSet —> X.T
      rest:  neSet —> Set
   for all a: X.T; s: neSet =>
      (min  s) ∈ s,              a ∈ s  →  X.≤ (min  s, a)
      a ∈ rest  s  →  a ∈ s,     a ∈ s  ∧ ¬ X.≤ (a, min  s)  →  a ∈ rest  s  ⌋
```

The subclass relationship is closely related to the condition that an actual parameter to a function is required to satisfy (as for subtypes): if X: A and A ≤ B then X is an admissible parameter for a function (Y: B) —> ... Parameter passing will generate a verification condition in general, but admissibility of an actual parameter may also be inferred statically from the existing subclass relationships: the stronger the stated relations in the specification the better. In the example ORDSETS, (partially) ordered sets are defined as an enrichment of LSETS. No verification is needed to show that X may be passed as an actual parameter to LSETS, since PO ≤ ELEM.

3.2. Subclass Inheritance, Refinement, Implementation

The real power and usefulness of the subtyping idea comes in, if it is also applied at the level of classes. The idea dates back to Simula [DMN 84] (with its successors in the area of Object-Oriented languages, e.g. Eiffel [Meyer 88]) and may have an interesting revival for formal specifications: the notion of a subclass can be equated with "the same or fewer models satisfying the superclass" in the context of a semantics of loose specifications. This corresponds to an intuitive notion of *inheritance* of properties, and the methodological notion of *refinement*: additional equations may make the initial model go "down" in the lattice of models (if there is one) since more terms can be identified, and additional operations (plus appropriate axioms to define them) may make the terminal model go "up" since more terms can be distinguished; inconsistent additions make the set of models empty. Together with an appropriate facility (see §3.3) for hiding and renaming, subclass inheritance directly provides a simple notion of *implementation* [ST 88] (but one may in practice require a more sophisticated notion of implementation up to observational equivalence ([Rei 81], [SW 83], [MG 85], [ST 87], [NO 88]).

In any case, the notion of subclass refinement should support the development process of both requirement and design specifications, in which additional design decisions restrict the set of admissible models, eventually leading towards a specification that includes all the intended constraints. Note that a requirement specification can be "frozen" (used as a well-defined and distinguished class for all users) and can serve for subsequent development of different design specifications as a hierarchy of subclasses. Such a hierarchy is illustrated by the examples of requirement specifications in §3.4.

It is quite important to notice that, apart from the methodological advantages, the structural aspects of subclasses also structure proofs (the objects of the proofs and the search for proofs, see [SB 83]) and, at least partially, allow a discharge of proof obligations by static analysis.

Union of classes means intersection of model sets as in ASL [SW 83] (name clashes must be resolved by renaming beforehand, otherwise functions are overloaded, (sub)type names are shared etc.).

3.3. Signature Morphisms, Hiding and Renaming

Structures or classes are comprised of aggregates (tuples) of declarations (in addition they also contain axioms in the case of classes, or definitional equations in the case of structures). Selection is by dot notation (as for qualified identifiers in SML); this notation is extended to allow also "aggregates" of selector identifiers, see below.

Signature morphisms can be expressed by an extended dot notation for selection and a notation for renaming in structures/classes; the notation is used analogously for tuples of values (of a product type, cf. records with tags in SML).

The notation is the same for all items (values of product type, product types, structures, classes). The term

X . (a' = a, .. , z' = z)

denotes a new aggregate item X' made of the components whose identifiers are listed (a etc.); consistent renaming (a' for a etc.) takes place. For a class, X' is the original class X restricted to the listed components, thus the result is a superclass of the original one. For a class X just denoting a signature, the result is the resp. subsignature; when axioms are present, the semantical notion of (restriction or) hiding has to be applied. For a structure, X' is the original structure X restricted to the listed components, thus the result is a structure of a superclass of the original class of X. For types and values, the definition is analogous. Note that X need not be the name of an item, but could be the definition of an item itself. This way, hiding for classes can be expressed directly (one may wish to have additional syntactic sugar for this).

3.4. Structuring Requirement Specifications

The activity of developing a first requirement specification of a software system is a very difficult task in itself ("requirement engineering"). Here, as at the level of subsequent development of design specifications, it is of great importance to be able to re-use existing specifications (and associated proofs etc.) that are "similar" in some sense. This means that existing specifications should have been put into the specification database in some generalised form, allowing some abstraction. The most common forms of abstraction seem to be functional abstraction (to be discussed below) and some kind of inheritance relation between specifications. Thus adding a specification to the database ("learning") should involve an important activity of generalisation corresponding to standardisation and abstraction, also, very importantly, a setting up of relationships in the database (corresponding to inheritance and instantiation) to ease later retrieval. The importance of building requirement specifications from pieces ("traits" in Larch [GH 86]) and of using inheritance relations for re-use (e.g. [Wir 88], [WBH 88], [WHS 89]) have been recognised.

Figure 1: Partial order of inheritance (including multiple inheritance).

```
                BINS            MONOID
            ≥       ≥              ≥
   BINTREES             MONS
                    ≥              ≥
            LISTS              LSETS
                          ≥           ≥
                 ORDSETS                LMULTISETS
                                             ≥
                                           SETS
```

Fig. 1 contains some examples of classes (see below) related by inheritance. In this diagram, F ≤ G really means that F X ≤ G X for any appropriate X (i.e. X: PO). MONOID is not parameterised, so, in the diagram, MONS $\leq$ MONOID means MONS X $\leq$ MONOID for any appropriate X. The subclass relationship as used in this diagram is up to renaming in some cases; for example, LISTS X $\leq$ MONS X $\leq$ MONOID is more precisely, LISTS X $\leq$ MONS.σ and MONS X $\leq$ MONOID.σ' where σ and σ' denote appropriate signature morphisms. This is a convention that is used for notational convenience in this diagram only; when A $\leq$ B in SPECTRAL, then A might have more components than B, but the components in B must appear in A with the same names (cf. §3.1).

```
BINS:   (X: ELEM) —> class =
  ⌈ Bin:      type
    eBin:     type  ≤ Bin
    neBin:    type  ≤ Bin
    empty:    eBin
    single:   T —> neBin
    ++:       neBin × Bin —>   neBin
    ++:       Bin × neBin—>    neBin
    ++:       Bin × Bin —>     Bin     ⌋

BINTREES:   (X: ELEM) —> class =
  BINS  X  +
  ⌈ left, right:   neBin —> Bin
  for all a: neBin; b: Bin  =>
    left (a ++ b) = a,   right (b ++ a) = a    ⌋

MONS:    (X: ELEM) —> class =
  BINS  X  +  MONOID . (Bin = M, empty = n, ++ = ⊗)

LISTS:   (X: ELEM) —> class =
  (MONS  X) . (List = Bin, eList = eBin, neList = neBin, empty, single, ++)  +
  ⌈ head:     neList —> X.T
    tail:     neList —> List
  for all e: X.T; b: List =>  head ((single  e) ++ b) = e,  tail ((single  e) ++ b) = b     ⌋

LSETS:  (X: ELEM) —> class =
  (MONS  X) . (Set = Bin, eSet = eBin, neSet = neBin, empty, single, ∪ = ++)  +
  ⌈ ∈:     X.T ×  Set —> bool
  for all e: X.T; a, b: Set =>
    e ∈ empty = false,       d ∈ single  e  <–>  d = e,       e ∈ (a ∪ b)  =  (e ∈ a) ∨ (e ∈ b)   ⌋

ORDSETS:    (X: PO) —> class =
  LSETS  X  +
  ⌈ min:   neSet —> X.T
    rest:  neSet —> Set
  for all a: X.T; s: neSet =>
    (min  s) ∈ s,             a ∈ s  →  X.≤ (min  s, a)
    a ∈ rest  s  →  a ∈ s,    a ∈ s  ∧ ¬ X.≤ (a, min  s)  →  a ∈ rest  s  ⌋

LMULTISETS:    (X: ELEM) —> class =
  (LSETS  X)  +
  ⌈ for all a, b: Set =>    a ∪ b = b ∪ a  ⌋
```

```
SETS:   (X: ELEM) —> class =
   LMULTISETS  X  +
   ⌈   for all a: Set =>         a ∪ a = a  ⌋
```

4. Higher-Order Functions

4.1. *Parameterised Specifications* of Programs versus *Specifications* of Parameterised Programs

In structuring in-the-large by (higher-order) functions, there is an important distinction to make between "*parameterised (program specification)*" and "*(parameterised program) specification*" (see [SST 90] and [ST 91a]). The approach in SPECTRAL supports a distinction very much in the spirit of that paper, although technically slightly different: the *parameterised specification* of a program denotes a *function yielding a class*, and the *specification* of a parameterised program denotes a *class* of functions yielding structures. The interesting outcome of this language design is that in both cases the parameters are structures (of a given class), or types or values, but not classes. It does not seem necessary to allow user-definable functions taking classes as parameters, the only reasonable ones being the pre-defined class composition operations (function space and union, see §6.2). Therefore there has been a conscious decision not to have functions from specifications to specifications as in [SST 90] and [ST 91a].

Below are some examples to illustrate these points.

parameterised specification of a program: function yielding a class

Consider the specification SORT of a sorting function. A structure X providing a type with a partial order and a structure L giving lists of these (with additional operations such as freq) are passed explicitly as parameters. Note that these structures are explicitly typed to be of their resp. classes (see also §4.2).

```
SORT:  (X: PO) —> (L: LISTS2  X) —> class =
   ⌈   sort:  L.List —> L.List
   for all a, b: X.T; l, x, y, z: L.List =>
       sort  l = x ++ (L.single  a) ++ y ++ (L.single  b) ++ z  →  X.≤ (a, b)
       L.freq  a  x  =  L.freq  a  (sort  x)                                ⌋
```

specification of a parameterised program: class of functions yielding structures

FSORT below denotes a *class* of functions yielding structures:

```
FSORT:     class =       (X: PO) —> (L: LISTS2  X) —> SORT  X  L
```

parameterised program: function yielding a structure

A parameterised program then corresponds to a function yielding a structure of a given class. Here, Sort X L: SORT X L for any X: PO and L: LISTS2 X; that is, Sort: FSORT. The structure Sort X L contains only the function sort; the function insert is hidden since it does not appear in SORT X L:

```
Sort:    (X: PO) —> (L: LISTS2  X) —> SORT  X  L  =
  ⌈ insert:    X.T —> L.List —> L.neList
    for all  x: X.T;  l: L.neList  =>
       sort  L.empty   =   L.empty
       sort  l                =  insert  (L.head  l)  (sort  (L.tail  l))
                      insert  x  L.empty     =  L.single  x
          X.≤ (x, L.head  l) →   insert  x  l  =  L.++  (L.single  x,  l)
       ¬  X.≤ (x, L.head  l) →   insert  x  l  =  L.++  (L.single  (L.head  l),  insert  x  (L.tail  l))     ⌋
```

4.2. Dependent Classes

The examples above make heavy use of dependent classes (cf. the analogy to dependent types in §5.6). In SORT, for example, the class of the second parameter L is an instantiation of LISTS2 with X, the first parameter. Similarly, the result class of FSORT depends on the parameters. These dependencies state important assertions. One can argue whether dependent classes or sharing constraints (as in SML) are simpler concepts; however, sharing constraints are superfluous in the presence of dependent classes. The examples above of SORT and Sort (or SETSasLISTS in §4.4) seem to be more elegant using dependent types; also, the passing of a particular list (implementation) as a parameter is more explicit. Examples like SORT and LISTS2 above correspond to dependent product types (Π-types); classes such as PO and MONOID correspond to dependent sum types (Σ-types); see e.g. [Con 86], [LB 88], [MacQ 86b], [MH 88].

4.3. Higher-Order Functions

Below are some more examples to demonstrate (as in SORT etc. above) that *higher-order* functions on classes and structures are very useful, even necessary to express certain specifications. Consider the problem of enriching a class by additional operations, in a hierarchical way so that the models of the original class are untouched (conservative/persistent extension). Two solutions are possible: the first, LISTOPS, defines the additional operations separately relating to an existing structure L (of class LISTS X) given as parameter. The second solution defines an extended class LISTS2 (cf. §3.1 above); it requires that instantiations have to be made of the extended class to get the additional operations, not "on second thought" for an already existing instantiation as in the case of LISTOPS.

```
LISTOPS:  (X: ELEM) —> (L: LISTS  X) —> class =
  ⌈ freq:  X.T —> L.List —> nat
  for all x, y, z: L.List =>   …  ⌋
```

LISTOPS2 is a less satisfactory variant in which LL, a "generic implementation" for LISTS, not a particular instantiated structure, is passed as a parameter. In this case, a local instantiation is made; this does not yield the desired effect since L.List now refers to the new instantiation L and is a different type from that for a similar instantiation of LISTS that may already exist in the context of an instantiation of LISTOPS2, or, for that matter, the L passed as a parameter to LISTOPS (as in SML, application of LL is generative, with each application producing a new structure containing "fresh" types).

```
FLISTS:    class  =    (X: ELEM) —> LISTS  X
```

```
LISTOPS2:     (X: ELEM) —> (LL: FLISTS) —> class =
   ⌈  L:      LISTS  X  =  LL  X
      freq:  X.T —> L.List —> nat
   for all x, y, z: L.List =>   …  ⌋
```

Consider now alternatives to the specification of a sorting operation in SORT. The specification SORT2 provides the additional operations by an extra parameter LO, while the specification SORT3 passes a "generic implementation" of class FLISTOPS; the local instantiation is no harm here since L is passed explicitly as a parameter.

```
SORT2:     (X: PO) —> (L: LISTS  X) —> (LO: LISTOPS  X  L) —> class =
   ⌈  sort:  L.List —> L.List
   for all a, b: X.T; l, x, y, z: L.List  =>
      sort  l = x ++ (L.single  a) ++ y ++ (L.single  b) ++ z  →  X.≤ (a, b)
      LO.freq  a  x  =  LO.freq  a  (sort  x)   ⌋

FLISTOPS:     class =  (X: ELEM) —> (L: LISTS  X) —> LISTOPS  X  L

SORT3:     (X: PO) —> (L: LISTS  X) —> (LLO: FLISTOPS) —> class =
   ⌈  sort:  L.List —> L.List
      LO:   LISTOPS  X  L  =  LLO  X  L
   for all a, b: X.T; l, x, y, z: L.List  =>
      sort  l = x ++ (L.single  a) ++ y ++ (L.single  b) ++ z  →  X.≤ (a, b)
      LO.freq  a  x  =  LO.freq  a  (sort  x)   ⌋
```

4.4. Parameterised Programs

In structures, values for components are given, in particular function "bodies". Consider for example an implementation of sets by lists (SETSasLISTS2 is an equivalent version of SETSasLISTS using the hiding and renaming notation described in §3.3):

```
SETSasLISTS:     (X: ELEM) —> (L: LISTS  X) —> SETS  X  =
   ⌈  Set       = L.List
      eSet      = L.eList
      neSet     = L.neList
      empty     =  L.empty
      ∪         = L.++
      single    = L.single
      for all e: X.T; s: neSet  =>
         e ∈ empty  =  false,   e ∈ s  =  (e = L.head  s) ∨ (e ∈  (L.tail  s))  ⌋

SETSasLISTS2:   (X: ELEM) —> (L: LISTS  X) —> SETS  X  =
   L . ( Set = List, eSet = eList, neSet = neList, ∪ = ++, empty, single ) +
   ⌈  for all e: X.T; s: neSet  =>
         e ∈ empty  =  false,   e ∈ s  =  (e = L.head  s) ∨ (e ∈  (L.tail  s))  ⌋
```

It is quite important to realise that the structuring for specifications (classes) and programs (structures) can be quite different. Programs often need additional access to structures and classes that are not needed, would even be severe overspecification, in their specifications (it is a major mistake in the design of Ada that there are no special with-clauses for private parts and that generic parameters must include the information needed exclusively for a particular implementation; this inhibits easy exchange of implementations). In the examples above, an implementation for lists is passed explicitly as a parameter and can thus be chosen and changed according to need. The operation + denotes the extension of structures (note that the components of the left operand are visible in the right operand).

The "program" SETSasLISTS (also SETSasLISTS2) seems to be wrong w.r.t. its specification, since there are axioms in SETS ($a \cup b = b \cup a$, $a \cup a = a$) that are not satisfied by the representation of sets as lists. However, the representation is correct "up to behavioural equivalence", meaning that all observable consequences of SETS X (equations between terms of type bool or X.T, in this case) are satisfied by SETSasLISTS X L. Thus we regard SETSasLISTS as a correct implementation of its specification. See e.g. [Rei 81], [SW 83], [MG 85], [Sch 87], [ST 87], [NO 88] and [ST 89] for more motivation and technical details.

The possibilities for currying and successive parameterisation are even better in the following version (at the expense of an extra local instantiation), since we can partially instantiate, for example, as SETSasLISTS3 StandardLISTS first (where StandardLISTS is some generic list implementation), and then give an actual parameter for elements later:

```
SETSasLISTS3:   (LL: FLISTS) —> (X: ELEM) —> SETS  X  =
   (LL  X) . ( Set = List, eSet = eList, neSet = neList, ∪ = ++, empty, single ) +
   ⌈  for all e: X.T; s: neSet  =>
         e ∈ empty  =  false,   e ∈ s  =  (e = L.head  s) ∨ (e ∈  (L.tail  s))  ⌋
```

A standard model for implementing sets as lists is one in which the lists are ordered. How do we implement sets in this way, since no order is given? This is a classical example, and a very practical problem, e.g. if an additional operation choose: neSet —> X.T is required (that chooses an arbitrary element from a given set, but consistently always the same element, no matter how the set was constructed). A solution is of course possible, if an order does exist on the element type, for a particular instantiation. This is a good example of structuring that is different for the implementation and the requirement specification (it would be an unfortunate overspecification to change the requirement specification from sets to ordered sets, similar to representing sets directly as lists as is often done in functional languages), and a nice example for the use of inheritance and higher order functions.

```
ORDSETSasLISTS:  (X: PO) —> (L: LISTS2  X) —> (Sort: SORT  X  L) —> ORDSETS  X  =
   SETSasLISTS  X  L  +
   ⌈  for all s: neSet  =>
         min  s  = L.head  (Sort.sort  s)
         rest  s  = ...                          ⌋

ORDSETSasLISTS2:
  (LLO: FLISTOPS) —> (FSort: FSORT) —> (X: PO) —> (L: LISTS  X) —> ORDSETS  X  =
   SETSasLISTS  X  L  +
   ⌈  for all s: neSet  =>
         min  s  = L.head  ( (FSort  X  (L + LLO  X  L)).sort  s)
         rest  s  = ...                                              ⌋
```

If, for a given structure X, an order exists, i.e. X **in** PO, then the implementation of sets SETSasLISTS X LX can be replaced by the implementation ORDSETSasLISTS X LX (Sort X

LX) since ORDSETS X ≤ SETS X. Note that LX: LISTS2 X must hold; otherwise the higher-order variant ORDSETSasLISTS2 can be used.

4.5. Function Subclasses

For subsorts, an order on function sorts can be defined and extended to higher-order. The same can be done here for subclasses: the basic idea is, that A —> B´ ≤ A´ —> B, if A´ ≤ A and B´ ≤ B, in other words, a function defined on a subdomain A´ can be replaced (implemented) by a function with a wider domain A yielding a result in a subdomain B´ (actually, the general form of contravariance has to be restricted, see [Qian 90]). For implementation schemata, this means that a schema can be replaced by another schema that is defined on a wider class of models, and a schema can be replaced by another schema that returns a smaller class of models as result (and the combination of the two).

5. Types, Subtypes and Polymorphism

5.1. Subtypes: "Subsorts" versus Constrained Domains

Subtypes have been introduced as a solution to the problem of defining partial functions (also errors or exceptions and exception handling), more precisely, functions that are only defined on a subdomain [GJM 85], [GM 87]. Methodologically, this is a way to encourage the user to specify constraints of domains of functions as tightly as possible; one pragmatic side-effect is that it encourages proper definition of tightly specified ranges as well.

Static type checking cannot be fully achieved in the presence of subtypes, particularly in a language with hierarchical specifications (cf. §2.4) and types constrained to be term generated (cf. §5.3). The situation is made even more difficult for types where subtypes are characterised by predicates as in the example of OrdListOf (this assumes the existence of a previously-defined predicate issorted; see also ListOf in §5.4).

```
OrdListOf: (X: PO) —> type = ( sl: ListOf X.T || issorted (X.≤) sl )
```

Similar examples arise in connection with "bounded" types (bounded lists, bounded integer arithmetic).

In SPECTRAL, we allow subtyping both in the "subsorting" and the "domain constrained by predicate" sense to allow as much type information as possible to be included in specifications. Only some of this information can be automatically checked; the remainder must be left as proof obligations that would be indicated in the text of the specification by explicit (sub)type qualification ("retracts"). An advantage of allowing the user to state type information that cannot be automatically checked is that this extra information will often enable other proof obligations (e.g. domain constraints on functions) to be discharged by "syntactic" means.

An important consideration is the interaction with overload resolution. The results from [GM 87] should be taken for overlapping domains (semantics must be the same in the overlap) and from [Qian 90] for higher-order subtypes. The interaction with implicit polymorphism and the type union operation are not fully worked out yet.

5.2. Definedness, Partial and (Non-)Strict Functions

At the moment, the PROSPECTRA approach to partiality, non-strictness etc. (cf. [Krie 91], Part II) has been taken in SPECTRAL; further investigation is needed. The extension of a type T to a type $T^{\perp}$, that is the type T with an additional bottom (undefined) element $\perp$, should fit well with the subtype concept. Problems generally arise when combining higher-order functions with the identification of truth values in the logic and the type of boolean values, see [FF 90]; another possibility for this is described in [ST 91b].

In the PROSPECTRA approach there is a special, non-operational domain of two-valued logic denoted by the type logical; it is only used for specification purposes and is distinguished from bool and its extension to the three-valued type $bool^{\perp}$ that contains the bottom element $\perp$. Predicates are functions that deliver the type logical as a result (as opposed to functions returning a result of type bool) and can only be used for specification purposes and not in a program (e.g. in an if then else). Thus predicates need not be operational (cf. e.g. equality in axioms, the connectives of the logic, or the definedness predicate in §6.2). Except for the predefined operations on logical, functions must not have logical in a parameter type. All functions on values (except predicates) are continuous.

The result of a function is defined (and in the specified result domain), if all parameters are defined and in their resp. domains. Thus such functions are total over the parameter domains; potential non-termination must be indicated by an additional bottom element in the result type.

Non-strict parameters are indicated by an additional bottom element in the parameter type; otherwise strict and lazy evaluation will give the same result. Non-strict functions can be used for infinite objects, in particular to define streams for the specification of concurrent systems, cf. the stream constructor operation cons below. Note that "properly" strict functions (non-definedness of a parameter implies non-definedness of the result) are implicitly introduced for constructors of disjoint union types.

cons : elem $\times$ $stream^{\perp}$ —> stream

if_then_else: (T: **type**) —> $bool^{\perp} \times T^{\perp} \times T^{\perp}$ —> $T^{\perp}$

5.3. Disjoint Union

Apart from pre-defined types and types defined in a class/structure, types can be constructed by using the composition operations (cf. §6.2), e.g. cartesian product or disjoint union. In the case of disjoint union, there is a special syntax for the abbreviation of constructor operations for values of the type. The simplest example is that of a type (Empty) with just one constant (nullary constructor operation), empty:

Empty : **type** = ⌈ empty: Empty ⌋

Another simple use of this notation is in CopyT, which produces a new type that is an isomorphic "copy" of an existing type T. The constructor operation copy: T —> CopyT creates values of the type CopyT; copy is implicitly surjective ("no junk") and injective ("no confusion").

CopyT: **type** = ⌈ copy : T —> CopyT ⌋

If a parameter name is given, it corresponds to an implicitly defined selector function (so in CopyT2, original: CopyT2 —> T is the left inverse of copy: T —> CopyT2):

```
CopyT2: type = ⌈ copy : (original: T) —> CopyT2 ⌋
```

Types with constructor operations may be combined using the disjoint union operator |. In t1 | t2, t1 and t2 must have disjoint sets of constructor operations. Then a new type is formed having both t1 and t2 as subtypes. All values of this type are either from t1 or from t2 ("no junk") and there is no overlap ("no confusion"). As an example, consider an alternative definition of natural numbers (cf. §2.2).

```
NATS:   class =
  ⌈  Nat:  type  =
        ⌈  zero:     Zero  ⌋
     |  ⌈  succ:  (pred: Nat) —> Pos  ⌋     ⌋
```

An equivalent form is the following (this is simply an abbreviation of the above). In both cases, Zero and Pos become subtypes of Nat.

```
NATS:   class =
  ⌈  Nat:  type  =
        ⌈  zero:     Zero
           succ:  (pred: Nat) —> Pos  ⌋     ⌋
```

Disjoint union may be used with recursion to define types inductively, as in NATS above and INTS below (cf. §3.1). The interpretation is as in SML, i.e. values may be viewed as terms built from constructor operations, where "no junk" amounts to an induction principle.

```
INTS:   class =
  NATS  +
  ⌈  Neg:  type  =
        ⌈  neg:  (pos: Pos) —> Neg  ⌋
     Int:   type  =
        Nat  |  Neg  ⌋
```

5.4. Functions on Types

"Parameterised type constructors" denote functions from types to types, for example ListOf, as an alternative to the parameterised specifications above:

```
ListOf:  (T: type) —> type =
  Empty
 |    ⌈ cons:   (head: T) × (tail: ListOf  T) —>  NeListOf  T ⌋
```

As discussed in §5.3, selection operations head: NeListOf T —> T and tail: NeListOf T —> ListOf T are defined implicitly. Note that a type constructor can also be made dependent on a structure, see OrdListOf in §5.1.

5.5. Polymorphism

Functions having types as parameters and yielding values denote the usual polymorphic functions (this is so-called "parametric" polymorphism [Rey 83]). As an example consider polymorphic equality (cf. §6.2), for use in axioms, and the if-then-else operation in §5.2:

= : (T: **type**) —> T × T —> logical

It is not a bad idea, in some cases, to be able to include an actual type parameter explictly, especially to support tight subsorting. In general, however, such type parameters should be inferred by the type analysis algorithm. Thus an actual type parameter is optional for all type parameters of functions, provided it can be inferred.

An interesting result of orthogonality and generality is the use of structures (instead of mere types) as parameters. This allows for the expression of additional functions and properties (in this case, the argument structures could conceivably be made optional under some circumstances). As an example consider sort:

sort: (X: PO) —> ListOf X.T —> OrdListOf X

5.6. Dependent Types

Dependent types are a very powerful concept to express required relationships, cf. also dependent classes in §4.2. The fact that many such dependencies are expressed structurally and can be statically checked avoids proof obligations that would otherwise be required. Note that, in full generality, types can be dependent on values. However, there is no conditional other than on values, and values can only be used to construct types in constraint predicates that are not checked statically anyway; thus dependency on values never influences type-checking. Also, as dependency on values only influences constraints in subtypes (see §5.1) or additional specifications that trigger proof obligations, no run-time type checking should be required (but assertions may depend on premises that are used in conditionals).

A standard example for the expressional power of dependent types is the following:

Vector: (T: **type**) —> (n: Nat) —> **type** = (v: ListOf T || length v = n)
Rectangle: (T: **type**) —> (m: Nat) × (n: Nat) —> **type** = (Vector T m) × (Vector T n)
Square: (T: **type**) —> (n: Nat) —> **type** = Rectangle n n

6. Notation

6.1. Abstract Syntax and Concrete Syntax Phrases

The Abstract Syntax is coded as a Term Algebra in the language itself. Note that subtypes (Name ≤ Exp) and type constructors (ListOf, NeListOf) are used.

```
Name:   type =
   Id
|⌈   mkQualName:  (prefix: Name) × (id: Id) —>                        QualName  - - ⟦ prefix . id ⟧
   mkAggrName:  (prefix: Name) × (aggregate: TupleExp) —>  AggrName⌋  - - ⟦ prefix . aggregate ⟧
```

```
Exp:    type =
   Name
|⌈   mkAppl:       (fun: Exp) × (arg: Exp) —>                            ApplExp    - - ⟦ fun arg ⟧
   mkTuple:    (exps: NeListOf  Exp) —>                             TupleExp   - - ⟦ ( exp1, ..., expn ) ⟧
   mkCtxt:     (defs: NeListOf  Def) —>                             CtxtExp    - - ⟦ ⌈ def1; ...; defn ⌋ ⟧
   mkForAll:   (defs: NeListOf  Domain) × (exps: ListOf  Exp) —> ForAllExp  - - ⟦for all defs => exps ⟧
   mkExist:  (defs: NeListOf  Domain) × (exps: ListOf  Exp) —> ExistExp ⌋  - - ⟦exist defs => exps ⟧
```

```
TExp:   type =
   Exp
|⌈   mkDomain:     (ids: NeListOf  Id) × (kind: TExp) —>            Domain      - - ⟦ ids: kind ⟧
   mkRestrDom:  (dom: Domain) × (constraint: ListOf  Exp) —>  RestrDom   - - ⟦ dom || constraint ⟧
   mkProduct:  (items: ListOf  TExp) —>                           Product     - - ⟦ item1 × ... × itemn ⟧
   mkType:                                                          TypeExp    - - ⟦ type ⟧
   mkClass:                                                         ClassExp⌋  - - ⟦ class ⟧
```

```
Def:    type =
   Domain                                                                      - - declaration
|⌈   mkRecDecl:    (dom: Domain) × (item: TExp) —>                 RecDecl ⌋   - - ⟦ dom = item ⟧
|  Exp                                                                         - - axioms etc.
```

The concrete syntax phrases given in the brackets ⟦ ⟧ define the major unparsing ("paraphrasing") rules of the abstract syntax as a text; they also define the major body of the concrete input syntax. Bracketing of expressions according to nesting of applications, infix (or possibly mixfix) function symbols (including special combinations of characters), operator precedence etc. have been abstracted away from. A unit of development is a Def, possibly restricted to large items such as classes or structures. A "program" is a sequence of such units, with linear visibility (non-linear visibility, e.g. for recursive definitions of types, is under consideration). Of course one would like to have a partial order of dependency w.r.t. visibility of other units ("import"); this syntax is not included above since static resolution of legal name visibility is assumed to have happened already. One may also wish to have additional syntax to declare definitions to be local to some construct, or to open scopes explicitly (e.g. to have direct access to components of classes/structures). The exact visibility rules need to be spelled out.

Note that an explicit functional abstraction construct is not given since it is "hidden" in the potentially recursive definitions, see §6.3.

The given Abstract Syntax is as compact as possible, suitable for defining the semantics of the language, transformation rules etc. Experience with support tools and transformation development in PROSPECTRA has shown that compactness (and orthogonality) is vital; of course, there is often a trade-off between expression of properties in a structural way (by syntax) or by static semantic predicates.

6.2. Predefined Operators

on Values

Only the most important operators on values are given; in addition there would be the usual operators on values of type logical and other types like if then else on bool. The predefined operators below often carry important semantics with them and could not be defined in the language itself; it is often somewhat arbitrary whether to include an operation in the abstract syntax above as a constructor or as a predefined operator here (e.g. cartesian product is a constructor since it is n-ary). (Sub)type qualification will probably need an additional static semantic restriction such as S≤T ∨ T≤S or some suitable generalisation.

= :	(T: **type**) —> T × T —> logical	- - (polymorphic) (strong) equality
defined:	(T: **type**) —> T —> logical	- - (polymorphic) definedness predicate
in:	(T: **type**) —> T × **type** —> logical	- - (polymorphic) type membership predicate
:: :	(T: **type**) —> T × (S: **type**) —> S	- - (polymorphic) type qualification

on Types

—>, | :	**type** × **type** —> **type**	- - function space, disjoint union
≤ :	**type** × **type** —> logical	- - subtype
⊥ :	**type** —> **type**	- - extended by Bottom element

on Structures

in:	(C: **class**) —> C × **class** —> logical	- - (polymorphic) class membership predicate
:: :	(T: **class**) —> T × (S: **class**) —> S	- - (polymorphic) class qualification
+ :	(C, D: **class**) —> C × D —> C + D	- - (polymorphic) extension of structures

on Classes

—>, + :	**class** × **class** —> **class**	- - function space, union of classes
≤ :	**class** × **class** —> logical	- - subclass

6.3. Deviations from Standard ML and Extended ML

Deviations from SML in the strict syntactic sense are not semantic deviations, rather abstractions and extensions for specification purposes. It is assumed that the generation of explicit discrimination for union types with elimination of subtypes, and the solution of the dependent types versus sharing constraint problem are possible; this seems relatively unproblematic but has not yet been studied in detail. The possibility of separating the definition of a "functor signature" (such as FSORT in §4.1) from the definition of a functor with that signature, as well as higher-order "functors" could be considered as extensions of SML proper eventually.

The uniform syntax proposed here (id: kind = exp) is a consequence of the generalisation. However, for a properly checked specification, it is easy to generate an SML unparsing or translation to SML abstract syntax by using static semantic information about kinds.

Function declarations with right-hand sides correspond to fun's in SML. In other words, definitions of functions with right-hand sides always have an implicit abstraction; the kind of the right-hand side is that of the result kind (i.e. the last in a curried sequence) of the function. Definitions yielding values or types may be recursive (but reflexive domains with "heavy" recursion [BT 83] are excluded), those yielding structures or classes must not be recursive.

In this language design, a structure corresponds to a finished program module; it denotes a single algebra in contrast to Extended ML where a structure may contain axioms and therefore denotes a set of algebras. The present approach is conceptually simpler but forces (for better or worse) a rather discrete transition from a class to a finished structure during the development process (e.g. by transformation of a monomorphic design specification to a corresponding functional program). Note also, that in a pure specification-oriented development methodology, structure-"values" and functions yielding structures as results need never appear explicitly except in this "last" stage of (specification-) development; there is never a need to define a structure, all development can be done using refinement of classes. Thus incorporation of a particular target programming language such as SML is of course necessary to obtain an executable program in the end but has no effect on the process of program development. It also follows that incorporation of a different target language is comparatively easy.

7. Conclusion

An attempt has been made to design a specification language that is concise and small and yet includes powerful concepts, in particular for structuring in-the-large. This is achieved by integrating a small number of general and orthogonal concepts in a minimal but flexible syntactic framework. Constructs that might otherwise be built into the syntax are defined as predefined operations in this design. Generality of function spaces and inheritance provide expressive power; dependent types/classes achieve compactness.

We expect that the static checking potential inherent in the typing and, more importantly perhaps, class inheritance construct will eliminate many proof obligations and help in structuring the proofs that remain.

Parallellism and concurrency require further research (although an approach to system specification and design using the stream based approach is already possible, see e.g. [Broy 88, 89], [Krie 91]).

We have recently become aware of related work [MMMK 90]. There are several similarities to the approach presented here: these include the use of 4 kinds of items (inherited from ML), and a similar notion of inheritance (although its exact semantics is not clear to us). There are some major differences: in order to support an object-oriented style of programming they use an interesting concept of coercion from structures to values that leads to a relaxation of the separation between the value/type level and the structure/class level of the language without sacrificing static type checking (this needs further investigation in our context), but there seems to be no attempt to generalise SML functors and their specifications to higher order as in the present approach; furthermore the semantics of inheritance when axioms are deleted is unclear. The sharing constraints of ML seem to be still supported (although a concept of dependent type exists); the use of "internal interfaces" seems to be for this purpose.

The paper presents the design of SPECTRAL, and of some of the motivations underlying the design choices. The semantic underpinnings of the language remain to be worked out (but see [Rei 90] for a partial first attempt using the notion of sketches from category theory); questions such as the extent to which static typechecking can be carried out remain unresolved. Providing a satisfactory semantics and answering these questions will be a very non-trivial task. However, the sematics of a programming language (Quest) with abstract data types, polymorphism, modules and inheritance has been given [CL 90] and the semantics of higher-order specification modules has been investigated [SST 90], [ST 91a]. Combining these treatments will be a challenging exercise that may lead to reconsideration of some of the detailed design choices.

We hope that the design of SPECTRAL will be a serious basis for a development methodology of correct programs in the future, supported by a powerful development environment (for example an adaptation of the PROSPECTRA system).

Acknowledgements

The paper has benefitted from discussions with Manfred Broy, Jörn von Holten, Yunbo Liu, Fernando Orejas, Frank Pfenning, Zhenyu Qian, Horst Reichel, Stefan Sokolowski, Andrzej Tarlecki, Martin Wirsing, and Burkhard Wolff.

References

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

[Broy 88] M. Broy: An example for the design of distributed systems in a formal setting: the lift problem. Universität Passau, Tech. Rep. MIP 8802 (1988).

[Broy 89] M. Broy: Towards a design methodology for distributed systems. *in:* M. Broy (ed.):: *Constructive Methods in Computing Science*. NATO ASI Series F55, Springer (1989) 311-364.

[BT 83] A. Blikle and A. Tarlecki: Naive denotational semantics. *Information Processing '83*. North-Holland (1983).

[CL 90] L. Cardelli, G. Longo: a semantic basis for Quest. Digital Systems Research Center, Palo Alto, Tech. Rep. No. 55 (1990). (extended abstract in *Proc. ACM Conf. on Lisp and Functional Programming*, Nice June 1990).

[CO 88] S. Clerici and F. Orejas: GSBL: an algebraic specification language based on inheritance. *Proc. 1988 European Conf. on Object Oriented Programming,* Oslo. *LNCS 322*, 78-92 (1988).

[COMPASS 91] M. Bidoit, H.-J. Kreowski, F. Orejas, P. Lescanne and D. Sannella (eds.): *A comprehensive algebraic approach to system specification and development: Annotated Bibliography. LNCS*, to appear (1991).

[Con 86] R. Constable et al: *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall (1986).

[DMN 84] O.-J. Dahl, B. Myrhaug and K. Nygaard: *Simula 67 common base language*. Report S-22, Norwegian Computing Center, Oslo (1970); revised edition (1984).

[FF 90] S. Finn and M. Fourman: LAMBDA logic manual. Draft Report, Abstract Hardware Ltd. (1990).

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

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

[GH 86] J.V. Guttag and J.J. Horning: Report on the Larch shared language. *Science of Computer Programming 6* (2), 103-134 (1986).

[GJM 85] J.A. Goguen, J.-P. Jouannaud and J. Meseguer: Operational semantics of order-sorted algebra. *Proc. 12th Intl. Conf. on Automata, Languages and Programming*, Nafplion, Greece. *LNCS 194* (1985).

[GM 87] J.A. Goguen and J. Meseguer: Order-sorted algebra solves the constructor-selector, multiple representation and coercion problems. *Proc. 2nd IEEE Symp. on Logic in Computer Science*, Cornell (1987).

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

[Har 86] R. Harper: Introduction to Standard ML. Report ECS-LFCS-86-14, Univ. of Edinburgh. Revised edition (1989).

[Krie 89] B. Krieg-Brückner: Algebraic specification and functionals for transformational program and meta program development. *in* Díaz, J., Orejas, F. (eds.): *Proc. TAPSOFT '89* (Barcelona), Vol. 2. *LNCS 352* (1989) 36-59.

[Krie 90] B. Krieg-Brückner: PROgram development by SPECification and TRAnsformation. *Technique et Science Informatiques: Advanced Software Engineering in ESPRIT* (special issue) (1990) 134-149

[Krie 91] B. Krieg-Brückner (ed.): PROgram development by SPECification and TRAnsformation: Vol. I: Methodology, Vol. II: Language Family, Vol. III: System. PROSPECTRA Reports M.1.1.S3-R-55.2, -56.2, -57.2. Universität Bremen, 1990. (to appear in *LNCS*).

[LB 88] B. Lampson and R. Burstall: Pebble, a kernel language for modules and abstract types. *Information and Computation 76*, 278-346 (1988).

[LL 88] T. Lehmann and J. Loeckx: The specification language of OBSCURE. *in:* D. Sannella (ed.): *Selected Papers of the 5th Workshop on Specification of Abstract Data Types*, Gullane, Scotland. *LNCS 332*, 131-153 (1988).

[MacQ 86a] D. MacQueen: Modules for Standard ML. In: Report ECS-LFCS-86-2, Univ. of Edinburgh (1986).

[MacQ 86b] D. MacQueen: Using dependent types to express modular structure. *Proc. 13th ACM Conf. on Principles of Programming Languages*, 277-286 (1986).

[Meyer 88] B. Meyer: *Object-Oriented Software Construction*. Prentice-Hall (1988).

[MG 85] J. Meseguer and J.A. Goguen: Initiality, induction and computability. *Algebraic Methods in Semantics* (M. Nivat and J. Reynolds, eds.), 459-540. Cambridge Univ. Press (1985).

[MH 88] J. Mitchell and R. Harper: The essence of ML. *Proc. 15th ACM Conf. on Principles of Programming Languages,* 28-46 (1988).

[MMMK 90] J. Mitchell, S. Meldal, N. Madhav and D. Katiyar: An extension of Standard ML modules with subtyping and inheritance (extended abstract). Draft report, Stanford University (1990). to appear in *Proc. ACM Conf. on Principles of Programming Languages* (1991)

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

[NO 88] M.P. Nivela and F. Orejas: Initial behaviour semantics for algebraic specifications. *in:* D. Sannella (ed.): *Selected Papers of the 5th Workshop on Specification of Abstract Data Types*, Gullane, Scotland. *LNCS 332*, 184-207 (1988).

[Qian 90] Z. Qian: Higher-order order-sorted algebra. *Proc. Algebraic and Logic Programming*, Nancy. *LNCS 463*, 86-100 (1990)

[Reade 89] C. Reade: *Elements of Functional Programming*. Addison-Wesley (1989).

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

[Rei 90] H. Reichel: A sketch approach to SPECTRAL semantics. Talk given at Informatik Kolloqium, Universität Bremen (1990).

[Rey 83] J. Reynolds: Types, abstraction and parametric polymorphism. *Information Processing '83*, 513-523. North-Holland (1983).

[San 90] D. Sannella: Formal program development in Extended ML for the working programmer. *Proc. 3rd BCS/FACS Workshop on Refinement*, Hursley Park. *LNCS*, to appear (1990).

[SB 83] D. Sannella and R. Burstall: Structured theories in LCF. *Proc. 8th Colloq. on Trees in Algebra and Programming*, L'Aquila, Italy. *LNCS 159*, 377-391 (1983).

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

[SST 90] D. Sannella, S. Sokolowski and A. Tarlecki: Toward formal development of programs from algebraic specifications: parameterisation revisited. Report 6/90, Univ. of Bremen (1990).

[ST 85] D. Sannella and A. Tarlecki: Program specification and development in Standard ML. *Proc. 12th ACM Symp. on Principles of Programming Languages*, New Orleans, 67-77 (1985).

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

[ST 87] D. Sannella and A. Tarlecki: On observational equivalence and algebraic specification. *J. Comp. and Sys. Sciences 34*, 150-178 (1987).

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

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

[ST 91a] D. Sannella and A. Tarlecki: A kernel specification formalism with higher-order parameterisation. *Proc. 7th Workshop on Specification of Abstract Data Types*, Wusterhausen. *LNCS,* to appear (1991).

[ST 91b] D. Sannella and A. Tarlecki: Extended ML: past, present and future. *Proc. 7th Workshop on Specification of Abstract Data Types*, Wusterhausen. *LNCS,* to appear (1991).

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

[Wik 87] Å. Wikström: *Functional Programming Using Standard ML*. Prentice-Hall (1987).

[Wir 88] M. Wirsing: Algebraic description of reusable software components. in: P. Wodon, E. Milgrom (eds.): *COMPEURO 88, System Design: Concepts, Methods and Tools*. IEE Comp. Soc. Press 834, 300-313 (1988).

[WBH 88] M. Wirsing, R. Breu, R. Hennicker: Reusable specification components. in: M. Chytil (ed.): *MFCS 88, Symp. on Mathematical Foundations of Computer Science,* Karlsbad Aug. 88. *LNCS 324*, 121-137 (1988).

[WHS 89] M. Wirsing, R. Hennicker, R. Stabl: MENUE - an example for the systematic reuse of specifications. in: C. Ghezzi, J. A. McDermid (eds.): *ESEC 89, 2nd European Software Engineering Conf.*, Warwick 89. *LNCS 387,* 20-41 (1989).