Acta Informatica 21, 443-472 (1984)

Acta Informatica

A Set-Theoretic Semantics for Clear

D.T. Sannella

Department of Computer Science, University of Edinburgh. James Clerk Maxwell Building, Mayfield Road, Edinburgh EH9 3JZ, UK

Summary. A semantics for the Clear specification language is given. The language of set theory is employed to present constructions corresponding to Clear's specification-combining operations, which are then used as the basis for a denotational semantics. This is in contrast to Burstall and Goguen's 1980 semantics which described the meanings of these operations more abstractly via concepts from category theory.

1. Introduction

A number of techniques for the formal specification of software have been developed in recent years. Prominent in this area is work by Guttag and his colleagues [17, 19] and by the ADJ group [16] and many others on algebraic methods of specification. In this framework, a specification consists of a *signature* - a set of *sorts* (kinds of data) and some *operators* (for constructing and manipulating data) - together with *axioms* (typically equations) describing constraints on the results produced by operators. Such a specification describes a collection of *algebras* (a set of data objects for each sort, and a function on those sets for each operator), where each algebra in the collection is a *model* of the specification (it satisfies the axioms). Programs can be considered to be algebras, so all programs satisfying a specification are in its collection of models.

Most workers in algebraic specification concentrate on the specification of abstract data types, for which the method is particularly well suited. Although an algebraic specification could be written for a large system, such a specification would be impossible to understand because it would contain so many axioms. The value of a specification depends on the ease with which it was written and can be understood; a large number of pages densely packed with axioms are not of much use to anybody.

The Clear specification language [4] was invented by Burstall and Goguen to combat just this problem. Clear is a language for writing *structured* algebra-

ic specifications; that is, it provides facilities for combining small specifications in various ways to make large specifications. With a tool such as this, the specification of a large real-world system could be built from small, easy to understand and (in many cases) reusable bits.

An obvious way to combine specifications is to simply add them together, giving a specification which includes the sorts, operators and axioms of each component. Clear also provides a facility for constructing a *parameterised* specification which can be applied to various different specifications to systematically enrich them in some way. A typical example is a parameterised specification of sorting, which would produce a specification of a program to sort lists of numbers when applied to the specification of natural numbers together with the usual $\leqq$ order relation. An operation called **data** can be applied when adding new sorts and operators to a specification; this constrains the collection of models to a small number of "best" ones. Finally, some of the operators and sorts of a specification can be "hidden" to yield a less elaborate specification. Clear is different from other specification languages because it attempts to take proper account of shared sub-specifications. That is, the specification-building operations are defined in such a way that if a certain component is used in building several larger specifications, then the specification which results from combining these larger units will contain only a single "copy" of the common subpart.

In order to make use of specifications it is essential that they be written in a specification language equipped with a complete formal semantics. It is not enough to write specifications in a language having only a formal syntax; this merely gives a dangerous illusion of precision. Only by means of a formal semantics can specifications be given a precise and unambiguous meaning. The exact meaning of any specification can then be determined mechanically by consulting the semantics.

Burstall and Goguen in [5] have given a denotational semantics for Clear, relying heavily on a number of concepts from category theory to give the meaning of Clear's specification-building operations. This semantics was developed at the same time as the Clear language itself. This had a positive effect on the resulting language (cf. [1]); the desire to give Clear an elegant category-theoretic semantics led to certain features being rejected and suggested generalisations of others. A special advantage of using the language of category theory is the generality of the result. The semantics in [5] abstracts away from the particular logical system to be used for writing specifications (i.e. the definitions of signature, axiom and model) using the device of an *institution* (see [14]; [5] used the term "language"), defining all at once the semantics of a large class of Clear-like languages.

A different semantics for Clear is presented here. This uses straightforward set-theoretic constructions to define the semantics of the specification-building operations. The result is equivalent to the semantics of [5], except that a number of errors have been corrected. The language described is the Clear used for examples in [5, 6] in which many-sorted signatures and algebras (without error operators) are used and axioms are equations (this language will be referred to as *ordinary Clear* in the sequel) rather than a family of lan-

guages. The semantics is less general than the semantics of [5] since it describes only Clear under this particular institution but as discussed in Sect. 9 it appears to be capable of straightforward modification to cover all institutions of interest.

Much of the material included here is taken with only minor changes from [5]. The overall organisation of the presentation used in that paper has been adopted here for the most part. Clear and its (category-theoretic) semantics is the result of much hard work by Burstall and Goguen; naturally the author does not claim credit for the design of the language or for those parts of the semantics which have not been altered.

The reader may wish to consult [16] for a leisurely explanation of the basic concepts of algebraic specification, [6] for an introduction to Clear as a specification language, and [5] to compare the two versions of the semantics. In order to make this paper self-contained, definitions of basic concepts such as signature, algebra and equation are given in Sect. 2, data constraints (used to give the semantics of the **data** operation) are discussed in Sect. 3, and a brief introduction to Clear is given in Sect. 4. Section 5 considers the problems involved in dealing with shared sub-specifications properly, and Sect. 6 discusses the difference between metatheories (used to give requirements for the arguments of a parameterised specification) and ordinary specifications. In Sect. 7 the semantic operations which form the basis of the semantics are defined. Section 8 lists the semantic equations for Clear; these are similar to those in [5] but not identical. Finally, Sect. 9 concludes with comments on some points raised by the semantics.

2. Basic Concepts and Notation

The basic algebraic concepts which underlie the semantics of Clear are introduced in this section. Many of the notions are similar to those used by other authors (see e.g. [16]). The definitions themselves are adapted from [5].

2.1. Signatures

A signature is a set of sorts (data type names) together with a set of operators (function names), where each operator has a type of the form $s1,\ldots,sn\to s$ where $s1,\ldots,sn,s$ are sorts. A signature morphism maps the sorts and operators of one signature to sorts and operators in another in such a way that types are preserved.

Definition. A *signature* Σ is a pair $\langle S,\Omega\rangle$ where S is a set (of sorts) and Ω is a family of sets $\{\Omega_{w,s}\}_{w\in S^*,s\in S}$ (of operators). We write $f\colon w\to s$ to denote $w\in S^*$, $s\in S$, $f\in\Omega_{w,s}$.

Definition. A *signature morphism* $\sigma\colon\langle S,\Omega\rangle\to\langle S',\Omega'\rangle$ is a pair $\langle\sigma_{sorts},\sigma_{opns}\rangle$ where $\sigma_{sorts}\colon S\to S'$ and σ_{opns} is a family of maps $\{\sigma_{w,s}\colon\Omega_{w,s}\to\Omega'_{\sigma^*(w),\sigma(s)}\}_{w\in S^*,s\in S}$ where $\sigma^*(s1,\ldots,sn)$ denotes $\sigma_{sorts}(s1),\ldots,\sigma_{sorts}(sn)$ for $s1,\ldots,sn\in S$. We will write $\sigma(s)$ for $\sigma_{sorts}(s)$, $\sigma(w)$ for $\sigma^*(w)$ and $\sigma(f)$ for $\sigma_{w,s}(f)$, where $f\colon w\to s$ is in Ω.

2.2. Algebras

A Σ-algebra has a set (the elements of a data type) for each sort of Σ and a function on those sets for each operator of Σ. A Σ-homomorphism maps the "data types" of one Σ-algebra to those of another in such a way that the functions are preserved.

Let $\Sigma = \langle S, \Omega \rangle$ be a signature.

Definition. A *Σ-algebra* A consists of an S-indexed family of *carrier* sets $|A| = \{|A|_s\}_{s \in S}$ and for each $f: s1, \ldots, sn \to s$ a (total) function $f_A: |A|_{s1} \times \ldots \times |A|_{sn} \to |A|_s$.

Definition. A *Σ-homomorphism* from a Σ-algebra A to a Σ-algebra B, $h: A \to B$, is a family of functions $\{h_s\}_{s \in S}$ where $h_s: |A|_s \to |B|_s$ such that for any $f: s1, \ldots, sn \to s$ in Ω and $a_1 \in |A|_{s1}, \ldots, a_n \in |A|_{sn}$ $h_s(f_A(a_1, \ldots, a_n)) = f_B(h_{s1}(a_1), \ldots, h_{sn}(a_n))$. A bijective homomorphism $h: A \to B$ is called an *isomorphism*, written $A \cong B$.

Given a Σ'-algebra A' and an injective signature morphism $\sigma: \Sigma \to \Sigma'$, we can recover the Σ-algebra buried inside A' (since A' is just an extension of this algebra). The definition extends without modification to the case in which σ is not injective, where the Σ-algebra will contain multiple copies of some of the carriers and functions of A'.

Definition. For any signature morphism $\sigma: \Sigma \to \Sigma'$ and Σ'-algebra A', the *σ-reduct* of A' is the Σ-algebra $A'|_\sigma$ such that for $s \in S$, $|A'|_\sigma|_s =_{\text{def}} |A'|_{\sigma(s)}$ and for $f: w \to s$ in Σ, $f_{A'|_\sigma} =_{\text{def}} \sigma(f)_{A'}$. When σ is obvious we sometimes use the notation $A'|_\Sigma$.

2.3. Equations

Definition. A *Σ-term* is a well-typed term built from operators in Σ and variables of sorts in Σ. A *ground* Σ-term is a Σ-term which contains no variables. If t is a Σ-term and $\sigma: \Sigma \to \Sigma'$ is a signature morphism, then the *σ-translation* $\sigma(t)$ of t is the Σ'-term obtained by replacing each operator f in t by $\sigma(f)$ and each variable x_s by $x_{\sigma(s)}$.

Definition. A *Σ-equation* $\forall X.t = t'$ is a set X of variables of sorts in S together with a pair t, t' of Σ-terms (possibly containing variables from X) of the same sort. An equation containing no variables is called *ground*. If $\sigma: \Sigma \to \Sigma'$ is a signature morphism and $e =_{\text{def}} \forall X.t = t'$ is a Σ-equation, then the *σ-translation* $\sigma(e)$ of e is $\forall \hat{X}.\sigma(t) = \sigma(t')$ where $\hat{X} = \{x_{\sigma(s)} | x_s \in X\}$.

Definition. A Σ-algebra A *satisfies* a Σ-equation $\forall X.t = t'$ (written $A \models \forall X.t = t'$) if the equation is 'true' (both sides evaluate to the same thing) for all assignments of values in A to the variables in X. A Σ-algebra A satisfies a set E of Σ-equations ($A \models E$) if A satisfies every Σ-equation in E.

2.4. Equational Theories

An equational presentation is a signature together with a set of equations on that signature. The closure of a set of equations is that set together with all its (model-theoretic) logical consequences. An equational theory is then a signature together with a closed set of equations. A theory morphism between two equational theories is a signature morphism between their signatures which preserves the equations.

Definition. An *equational Σ-presentation* is a pair $\langle \Sigma, E \rangle$ where Σ is a signature and E is a set of Σ-equations. A Σ-algebra A *satisfies* an equational presentation $\langle \Sigma, E \rangle$ if A satisfies E. Then A is called a *model* of $\langle \Sigma, E \rangle$.

Definition. If E is a set of Σ-equations, let E^* be the set[1] of all Σ-algebras which satisfy E. If M is a set of Σ-algebras, let M^* be the set of all Σ-equations which are satisfied by each algebra in M. The *closure* of a set E of Σ-equations is the set E^{**}, written $\bar{E}$. E is *closed* if $E = \bar{E}$.

Definition. An *equational Σ-theory* T is an equational presentation $\langle \Sigma, E \rangle$ where E is closed.

Definition. An *equational theory morphism* $\sigma: \langle \Sigma, E \rangle \to \langle \Sigma', E' \rangle$ is a signature morphism $\sigma: \Sigma \to \Sigma'$ such that $\sigma(e) \in E'$ for each $e \in E$.

3. Data Constraints and Data Theories

An equational theory will typically have a number of different (non-isomorphic) models. Some of these will be trivial, and others will contain extra useless values. Most approaches to the specification of abstract data types and programs restrict consideration to a special subset of models; probably the best-known example is the "initial algebra" approach of [16] in which an equational theory is taken to specify only its *initial* models.

Definition. A model A of an (equational) theory T is an *initial* model of T if for every model B of T there is a unique homomorphism $h: A \to B$.

Equivalently, a Σ-algebra A is an initial model of $T = \langle \Sigma, E \rangle$ if it satisfies the following conditions:

- "No junk": Every element in A is the value of some ground Σ-term.
- "No confusion": For every ground Σ-equation e, A satisfies e iff $e \in E$.

For a proof that these definitions are equivalent and for three other equivalent definitions see [15].

It is well-known that the initial models of any (equational) theory form an isomorphism class; this allows us to refer to *the* initial model.[2] Note that the

[1] Actually, this is not a set at all but a proper *class*. The set/class distinction will be ignored throughout this paper

[2] Category-theoretically speaking, the initial model of an equational theory T is the initial object in the category Mod(T) of T-models and homomorphisms between them

"no junk" condition corresponds to an induction principle; since all values in the initial model are generated by terms, proof by structural induction on terms is possible.

Clear adopts a generalisation of the initial model approach. A Clear theory may include *data constraints* which must be satisfied (in addition to the equations) by any model of the theory. A data constraint specifies that for each model, the subalgebra of that model corresponding to a certain subtheory must be a "free extension" of the subalgebra corresponding to a certain smaller subtheory (i.e. initial *relative* to it). This is necessary for specifications such as Set-of-X (see Sect. 4), where X may be arbitrary. Each application of the **data** operation in a Clear specification contributes a data constraint.

Definition. A *Σ-data constraint* c is a pair $\langle i, \sigma \rangle$ where $i: T \hookrightarrow T'$ is an equational theory inclusion and $\sigma: \mathrm{sig}(T') \to \Sigma$ is a signature morphism.

A data constraint is a description of an enrichment (the theory inclusion goes from the equational theory to be enriched to the enriched theory) together with a signature morphism "translating" the constraint to the signature Σ.

Definition. A Σ-algebra A *satisfies* a Σ-data constraint $\langle i: T \hookrightarrow T', \sigma: \mathrm{sig}(T') \to \Sigma \rangle$ if:

– "No junk": Every element of $A|_{\mathrm{sig}(T')}$ is the value of some $\mathrm{sig}(T')$-term containing variables only in sorts of T, for some assignment of values to variables.

– "No confusion": The values of two $\mathrm{sig}(T')$-terms are the same in $A|_{\mathrm{sig}(T')}$ for an assignment of values to variables iff they are forced to be equal by the interpretation of T and the equations of T'.

Again, the "no junk" condition corresponds to an induction principle. See [5] for a category-theoretic version of this definition.

A signature morphism from Σ to Σ' can be applied to a Σ-constraint to translate it to a Σ'-constraint, just as it can be applied to a Σ-equation to give a Σ'-equation.

Definition. If $\sigma: \Sigma \to \Sigma'$ is a signature morphism and $\langle i, \sigma' \rangle$ is a Σ-data constraint, then the *σ-translation* of $\langle i, \sigma' \rangle$ is the Σ'-data constraint $\langle i, \sigma' \circ \sigma \rangle$.

Since data constraints "behave" just like equations (in the sense that satisfaction of a data constraint by an algebra and the translation of a data constraint by a signature morphism are defined), they can be added to the equation set in an equational presentation to give a data presentation (or presentation for short).

Definition. A *(data) Σ-presentation* is a pair $\langle \Sigma, EC \rangle$ where Σ is a signature and EC is a set of Σ-equations and Σ-constraints.

The notions of *satisfaction* (of a data presentation), *closure*, *(data) theory*, and *(data) theory morphism* follow as in the equational case. The denotation of a Clear specification is a (data) theory $\langle \Sigma, EC \rangle$, specifying all Σ-algebras which satisfy the equations and data constraints in EC (actually, a *based* data theory – see Sect. 5).

The data constraints described here are a special case of those discussed in [5]; general data constraints never arise in ordinary Clear. Essentially the same

concept was described earlier in [20] (cf. [23]) under the name *initial restriction*. Data constraints are also used in the LOOK specification language [8]. A more general form of data constraints (*free generating constraints*) is discussed in [9].

It is possible to employ a weaker version of data constraints, sometimes called *hierarchy constraints* [26] cf. [2] or *generating constraints* [10], in which an algebra satisfies a constraint iff it satisfies the "no junk" condition. In order to avoid trivial models it is necessary to allow specifications to contain inequations or else to impose the extra condition that all models satisfy true ≠ false.

4. An Introduction to Clear

Below is an example of a Clear specification which displays most of the features of the language. Some brief explanatory notes follow the example.

```
const Bool =
    theory
        data sorts bool
             opns true, false : bool
                  not : bool → bool
             eqns not(true) = false
                  not(false) = true   endth
const Boolopns =
    enrich Bool by
        opns and, or, ⇒ : bool, bool → bool
        eqns all p:bool. p and true = p                    all p:bool. p and false = false
             all p:bool. p or true = true                  all p:bool. p or false = p
             all p, q:bool. p ⇒ q = not(p and not(q))      endn

meta Ident =
    let Ident0 = enrich Boolopns by
                    sorts element
                    opns ≡ : element, element → bool
                    eqns all m:element. m ≡ m = true
                         all m, n:element. m ≡ n = n ≡ m
                         all m, n, p:element. (m ≡ n and n ≡ p) ⇒ m ≡ p = true   endn
    in derive sorts element
             opns eq : element, element → bool
        using Bool
        from Ident0
        by eq is ≡   endde

proc Set(X:Ident) =
    let Set0 = enrich X by
                  data sorts set
                       opns φ : set
                            singleton : element → set
                            ∪ : set, set → set
                       eqns all S:set. φ ∪ S = S
                            all S:set. S ∪ S = S
                            all S, T:set. S ∪ T = T ∪ S
                            all S, T, V:set. S ∪ (T ∪ V) = (S ∪ T) ∪ V
```

```
    in enrich Set0+Boolopns by
        opns ∈: element, set → bool
             choose: set → element
        eqns all a:element. a∈φ = false
             all a, b:element. a∈singleton(b) = eq(a, b)
             all a:element, S, T:set. a∈(S∪T) = (a∈S) or (a∈T)
             all a:element, S:set. choose(singleton(a)∪S)∈(singleton(a)∪S) = true   enden

const Nat =
    enrich Bool by
        data sorts nat
             opns 0: nat
                  succ: nat → nat
                  +: nat, nat → nat
             eqns all n:nat. 0+n = n
                  all m, n:nat. succ(m)+n = succ(m+n)   enden

Set(Nat[element is nat, eq is = =])
```

Note that the **data** operation is associated with an enrichment, not just with a specification (**theory**...**endth** is equivalent to **enrich** Empty **by**...**enden**). As well as adding a data constraint describing the enrichment, **data** contributes an extra operator $= = : s, s \rightarrow$ bool for each new sort s. If p and q are terms of sort s, $p = = q$ = true iff $p = q$ in all models of the specification (which must satisfy the new data constraint). Thus, the models of Bool and Nat are the standard ones, with the expected interpretation of $= =$. It is also possible to enrich a specification without using the **data** operation, as in Boolopns (where in fact an application of **data** would not change the class of models) and in Ident (where **data** would force all models to have an empty carrier for the sort *element*). The **derive** operation is used in Ident to 'forget' the operators of Boolopns which made it convenient to write the equations of Ident0 but which are not required in the result, and to rename the operator $\equiv$ to eq. The **using** clause here says that the result should incorporate Bool.

Ident is a *metatheory*, i.e. it is used as the *requirement* (*metasort*) of a parameterised specification (*procedure*) to describe its allowable actual parameters. The **meta** construct for declaring metatheories did not appear in previous versions of Clear because (incorrectly) the distinction between a metatheory and an ordinary *constant* theory was not made. See Sect. 6 for a detailed discussion of this point. Set is a parameterised specification with a single parameter which may be applied to any specification which "matches" Ident. A *fitting morphism* must be provided to give the correspondence between the sort/operator names in Ident and in the actual parameter. The application of **data** in Set contributes the data constraint $\langle \text{Ident} \hookrightarrow \text{Set0}, id \rangle$ with the result that in each model A of Set, $|A|_{\text{set}} \cong \mathscr{P}(|A|_{\text{element}})$ (the set of finite subsets of $|A|_{\text{element}}$) with the appropriate interpretations for the operators ϕ, *singleton* and $\cup$. But note that *choose* has not been completely specified; all that has been said is that *choose* selects some element from any nonempty set. Which element to pick is left unspecified, as is the result of *choose*(ϕ). Thus, the models of *Set*(*Nat*[...]) do not form an isomorphism class; we say that *Set*(*Nat*[...]) is a *loose* specification.

Infix operators have been used freely in the example above, although formally the syntax of Clear does not provide for them. Another liberty

concerns "overloaded" operators (such as ==: bool,bool→bool and ==: nat,nat→bool in Nat); although the definition of signature allows an operator to have multiple types, the semantics provides no way of resolving references to such operators. In practice a typechecker can usually disambiguate such references according to the context.

5. Dealing with Shared Subtheories

Consider the following specification fragments taken from the example in the last section:

```
const Boolopns = enrich Bool by
                    opns and, or, ⇒: bool, bool → bool
                    eqns ...
const Nat = enrich Bool by
                data sorts nat
                    opns 0: nat
                         succ: nat → nat
                         +: nat, nat → nat
                    eqns ...
```

Notice that both Boolopns and Nat "include" the theory Bool; Bool is a *shared subtheory* of Boolopns and Nat. What does this mean formally? And, how does the semantics of Clear define the theory-combining operations so that the theory Boolopns + Nat includes only *one* copy of Bool?

In [4], shared subtheories are explained by analogy with the EQ predicate of LISP [21]. The EQUAL function in LISP tests whether two lists *look* the same (i.e. whether they contain the same elements in the same order), while EQ tests whether two lists *are* the same (occupy the same list cells in storage – note that EQ(a,b) implies EQUAL(a,b) but not vice versa). The important features of EQ are given by the following examples (a,b and c are arbitrary lists):

i) EQ(CONS(a,b), CONS(a,b)) = false (but EQUAL(...,...) = true)
ii) (EQ(l,l) where l = CONS(a,b)) = true
iii) EQ(CAR(CONS(a,b)), $CDR(CONS(c,a))$) = true

These examples show that

i) Writing down a CONS expression twice gives two different lists.
ii) Two uses of the same variable refer to the same list.
iii) Two different lists can share a common sublist.

Now to complete the analogy, the theory-building operations of Clear act like CONS and the behaviour of EQ indicates what is meant by "identical" in the following:

Requirement. The theory-building operations should be defined in such a way that a theory can never contain two identical subtheories.

This leads (for example) to the following informal constraint on the *combine* (+) operation:

Constraint. If B is a subtheory of A and D is a subtheory of C, then B and D should be identified when forming $A+C$ iff they are identical.

In order to write a semantics for Clear we must devise some representation of theories which makes it easy (or at least possible) to determine if two theories are identical, so that the above constraint can be satisfied. The general category-theoretic semantics of [5] uses a rather complicated representation of a theory which shows explicitly how the theory is related to every one of its subtheories. We can use a much simpler representation because the only way that a theory and one of its subtheories can be related in ordinary Clear is by inclusion.

An important observation is the fact that the requirement above is inherited by the sorts and operators of a theory (where identity is again by analogy to EQ in LISP), giving:

Requirement. The theory-building operations should be defined in such a way that a theory can never contain two identical sorts or operators.

Moreover, if this low-level requirement is satisfied (and the operations are defined in a reasonable way) then the previous requirement will be satisfied as well. The above contraint on *combine* also has a low-level equivalent.

The semantics of EQ in LISP is defined in terms of a model of storage where lists are stored in addressable cells and EQ simply checks whether its arguments begin at the same address (see [21]). By associating a unique address with each non-EQ list cell, the meaning of EQ is reduced to equality of addresses. By analogy, if we associate an appropriate *tag* with each sort and operator we can easily determine whether two *tagged sorts* or *tagged operators* are identical in the sense indicated above. If the name of the theory of origin of a sort or operator is used as a tag, then the sort or operator name together with the tag forms a unique and precise name for the object (sort or operator). Then if (for example) f is an operator belonging to both A and B, f will appear once in $A+B$ if f has the same tag (theory of origin) in both A and B; otherwise f of A and f of B are really different operators which just happen to have the same name, and $A+B$ should include both. The language IOTA [22] also uses tags (to qualify operator names).

Each theory is therefore represented for the purposes of our semantics as a *tagged theory* (a theory where the names are all tagged). The tagged theories Boolopns and Nat look like this, where tags are shown as subscripts:

Boolopns =
sorts $\text{bool}_{\text{Bool}}$
opns $\text{true}_{\text{Bool}}, \text{false}_{\text{Bool}} : \text{bool}_{\text{Bool}}$
$\text{not}_{\text{Bool}} : \text{bool}_{\text{Bool}} \rightarrow \text{bool}_{\text{Bool}}$
$\text{and}_{\text{Boolopns}} : \text{bool}_{\text{Bool}}, \text{bool}_{\text{Bool}} \rightarrow \text{bool}_{\text{Bool}}$
...
eqns ...

Nat =
sorts $\text{bool}_{\text{Bool}}, \text{nat}_{\text{Nat}}$
opns $\text{true}_{\text{Bool}}, \text{false}_{\text{Bool}} : \text{bool}_{\text{Bool}}$
$\text{not}_{\text{Bool}} : \text{bool}_{\text{Bool}} \rightarrow \text{bool}_{\text{Bool}}$
$0_{\text{Nat}} : \text{nat}_{\text{Nat}}$
...
eqns ...

Boolopns + Nat is simply the set-theoretic *union* of these two tagged theories:

sorts $\text{bool}_{\text{Bool}}$, nat_{Nat}
opns $\text{true}_{\text{Bool}}$, $\text{false}_{\text{Bool}} : \text{bool}_{\text{Bool}}$
$\text{not}_{\text{Bool}} : \text{bool}_{\text{Bool}} \rightarrow \text{bool}_{\text{Bool}}$
$\text{and}_{\text{Boolopns}} : \text{bool}_{\text{Bool}}, \text{bool}_{\text{Bool}} \rightarrow \text{bool}_{\text{Bool}}$
...
$0_{\text{Nat}} : \text{nat}_{\text{Nat}}$
...
eqns ...

The particular tags used are not important; all that matters is that the tags for two different sorts (or operators) which have the same name, are different. Thus, X146 and Y27 would serve as well as Bool and Nat above. Also (for example) *true* and *false* need not have the same tag. This fact will be useful in the semantics; it turns out to be inconvenient to tag sorts and operators with the name of their theory of origin.

A problem arises when we consider how to treat the operation of applying a parameterised specification to an argument. Suppose that Set is the parameterised specification defined in the last section. It is natural to regard its body as a tagged theory:

sorts $\text{bool}_{\text{Bool}}$, $\text{element}_{\text{Ident}}$, set_{Set}
opns $\phi_{\text{Set}} : \text{set}_{\text{Set}}$
$\text{singleton}_{\text{Set}} : \text{element}_{\text{Ident}} \rightarrow \text{set}_{\text{Set}}$
...
eqns ...

The expressions *Set*(*Nat*[*element* **is** *nat*, *eq* **is** ==]) and *Set*(*Bool*[*element* **is** *bool*, *eq* **is** ==]) will also denote tagged theories. But what should the tags be? A first attempt might be the following:

Set(Nat[element **is** nat, eq **is** ==]) =
sorts $\text{bool}_{\text{Bool}}$, nat_{Nat}, set_{Set}
opns $0_{\text{Nat}} : \text{nat}_{\text{Nat}}$
...
$\phi_{\text{Set}} : \text{set}_{\text{Set}}$
$\text{singleton}_{\text{Set}} : \text{nat}_{\text{Nat}} \rightarrow \text{set}_{\text{Set}}$
...
eqns ...

Set(Bool[element **is** bool, eq **is** ==]) =
sorts $\text{bool}_{\text{Bool}}$, set_{Set}
opns $\text{true}_{\text{Bool}}$, $\text{false}_{\text{Bool}} : \text{bool}_{\text{Bool}}$
...
$\phi_{\text{Set}} : \text{set}_{\text{Set}}$
$\text{singleton}_{\text{Set}} : \text{bool}_{\text{Bool}} \rightarrow \text{set}_{\text{Set}}$
...
eqns ...

Now consider the theory *Set*(*Nat*[...]) + *Set*(*Bool*[...]). We would expect this theory to contain two sorts with the name *set*, one from each of the two theories. But the result contains only one sort named *set* since in both of the tagged theories above this sort has the same tag. A similar problem arises with the operators on sets, although types serve to distinguish the two *singleton* operators.

The solution is to assign new (and distinct) tags to the sort *set* and the operators ϕ, *singleton* etc. in the process of applying the parameterised specifi-

cation. This gives:

Set(Nat[element **is** nat, eq **is** = =])=
sorts $\text{bool}_{\text{Boo!}}$, nat_{Nat}, set_{New1}
opns $0_{\text{Nat}} : \text{nat}_{\text{Nat}}$
...
$\phi_{\text{New1}} : \text{set}_{\text{New1}}$
$\text{singleton}_{\text{New1}} : \text{nat}_{\text{Nat}} \rightarrow \text{set}_{\text{New1}}$
...
eqns ...

Set(Bool[element **is** bool, eq **is** = =])=
sorts $\text{bool}_{\text{Bool}}$, set_{New2}
opns $\text{true}_{\text{Bool}}, \text{false}_{\text{Bool}} : \text{bool}_{\text{Bool}}$
...
$\phi_{\text{New2}} : \text{set}_{\text{New2}}$
$\text{singleton}_{\text{New2}} : \text{bool}_{\text{Bool}} \rightarrow \text{set}_{\text{New2}}$
...
eqns ...

When these two theories are combined, the desired result is produced. Note that sorts such as *bool* and *nat* (and their associated operators) retain their original tags, so combinations such as *Set*(*Nat*[...])+*Nat* will contain just one copy of each.

The problem now is: how do we determine which sorts are to be retagged during the application of a parameterised specification? In this example, one possibility would be to retag all sorts and operators having the tag *Set* but such a simple approach does not work for more complicated examples.

The solution to this problem which gives the same effect as the semantics of [5] is to retag in $P(A[...])$ only those sorts and operators of P which do not originate in a constant subtheory of P, i.e. in a subtheory originally produced by a declaration **const** $TN = \ldots$. (This is not the only possible solution, and in fact it gives rise to the problem of "proliferation" mentioned in [6] – i.e. *Set*(*Nat*[...])+*Set*(*Nat*[...]) will contain two sorts with the name *set*.) In order to implement this strategy it is necessary to keep track of all the constant subtheories of a theory. Since all these subtheories appear in the constant theory environment (see below) it is sufficient to keep track of their names. Adding this set of names (called a *base*) to a tagged theory gives a *based theory*. The addition of a base does not complicate the definition of the sum of two theories; the base of the sum is simply the union of the bases.

Definition. A *based theory* is a pair $\langle T, B \rangle$ where T is a theory with tagged sorts and operators and B (the *base*) is a set containing the names of the constant subtheories of T. $\langle T, B \rangle$ is normally written T_B.

Definition. A *based theory morphism* σ: $T_B \rightarrow T'_{B'}$ (where $B \subseteq B'$) is a theory morphism σ: $T \rightarrow T'$ such that σ restricted to the theories named in B is the identity.

These based theories should not be confused with the based theories of [5] mentioned earlier. Although the definitions are different, both kinds of based theories serve the same purpose so we use the same name to draw attention to this similarity.

The base of a based theory contains names of theories in the environment of constant theories, which records declarations of the form **const** $TN = \ldots$:

Definition. The *constant theory environment* ρ is a function which maps names to based theories.

The constant theory environment is used in the usual way to retrieve the theory associated with a particular theory name. The semantics also requires separate metatheory and parameterised specification environments – see Sect. 8.5.

Note that the constant subtheories of a constant theory include the theory itself, but the constant subtheories of the theory which appears on the right-hand side of a declaration do not include the theory being declared. That is, in the declaration

const TN = expr

the base of the theory which *expr* denotes (this is the based theory which is bound to TN in the constant theory environment) will not contain TN, while subsequent uses of the theory name TN will denote a theory having a base which includes TN.

A subtle point is the way that the base of a theory influences sharing. In all contexts which do not include application of a parameterised specification or declaration, a tagged sort/operator ω_A will be identified with ν_B iff $\omega = \nu$ and $A = B$. But in general contexts this is only the case if every parameterised specification and metatheory which includes ω_A (or ν_B) has a theory in its base which includes ω_A (resp. ν_B), since otherwise ω_A and ν_B are subject to retagging.

6. Metatheories

Metatheories are used in Clear specifications to give the *requirements* (*meta-sorts*) of parameterised specifications. For example (simplifying a fragment from Sect. 4):

meta Idmeta = **enrich** Bool **by**
sorts element
opns eq: element, element → bool
eqns **all** m: element. eq(m, m) = true
...

proc Set(X: Idmeta) = **enrich** X **by** ...

Here, Idmeta is a metatheory "describing" all theories having at least one sort and an equivalence relation on that sort. Any such theory can be used as an argument of Set. In this section the relation between metatheories and ordinary (constant) theories is discussed. The category-theoretic Clear semantics of [5] did not treat this aspect correctly, using constant theories to give requirements of parameterised specifications. (A corrected version of this semantics is given in [24].)

A metatheory is not a new kind of theory, but only an ordinary based theory used in a special way. A metatheory M described the class containing those based theories T for which a based theory morphism $\sigma: M \to T$ exists. This is the formal equivalent of the condition that an actual parameter theory must match the corresponding requirement theory with respect to the renaming of sorts and operators given by the fitting morphism. This fitting morphism (supplied by the user) is used to construct the result of applying a param-

eterised specification. But in order for this to work the metatheory M must be constructed in a slightly different way from a constant theory; this is the reason why the **meta** construct is used to define a metatheory. The **meta** construct did not appear in previous versions of Clear because (incorrectly) this distinction was not made.

An easy way to understand the difference between constant theories and metatheories is to observe what happens when the metatheory Idmeta above is replaced by a constant theory Idconst as the requirement of a parameterised specification:

const Idconst = **enrich** Bool **by**
sorts element
opns eq: element, element → bool
eqns **all** m: element. eq(m, m) = true
...
proc Setconst(X: Idconst) = **enrich** X **by** ...

Idconst yields the following based theory:

sorts $\text{bool}_{\text{Bool}}$, $\text{element}_{\text{Idconst}}$
opns $\text{true}_{\text{Bool}}$, $\text{false}_{\text{Bool}}$, ...
$\text{eq}_{\text{Idconst}}$
eqns ...
base Bool, Idconst

What are the possible actual parameter theories to which Setconst can be applied? Recall that a *based* theory morphism is used to fit an actual parameter to its corresponding requirement theory; the morphism goes from the requirement to the actual parameter. Since the base of the target of a based theory morphism must include the base of the source (and the morphism restricted to the base must be the identity), the actual parameter must contain Idconst as a constant subtheory. In essence, the only theory Setconst can be applied to is Idconst itself. This is clearly neither intended nor desirable.

The declaration of Idmeta above yields the following based theory:

sorts $\text{bool}_{\text{Bool}}$, $\text{element}_{\text{Idmeta}}$
opns $\text{true}_{\text{Bool}}$, $\text{false}_{\text{Bool}}$, ...
$\text{eq}_{\text{Idmeta}}$
eqns ...
base Bool

The only difference between Idconst and Idmeta as based theories is that while Idmeta has a base consisting only of Bool, the base of Idconst contains Idconst itself as well. This change is all that is necessary to make Idmeta the appropriate requirement for the parameterised specification Set above. Since the base of Idmeta contains Bool, any actual parameter of Set must include Bool as a subtheory. But it need only *match* the rest of Idmeta; that is, it must include a sort with an equivalence relation. Suitable actual parameter theories and fitting morphisms are:

Nat[element **is** nat, eq **is** = =]
Bool[element **is** bool, eq **is** = =]

and many others. In general, the difference between constant theories and metatheories is just that metatheories are not recorded in the bases of theories which contain them.

In the example above, a constant theory (Bool) was included in a metatheory (Idmeta). In general, metatheories can be put together (with each other and with constant theories) using the same operations as for constant theories, since they are nothing more than a special kind of based theory. When such a conglomerate is used as a requirement theory, any matching actual parameter must include all of the constant subtheories of the requirement as well as sorts and operators which match those of the metatheories.

The concept of a metatheory in Clear is similar to the notion of a *sype* in the language IOTA [22]; there too, a sype is not very different from an ordinary type, although it can be regarded as a higher order concept.

7. Semantic Operations

In this section the semantic operations which "implement" the theory-building operations of Clear are defined. This forms the quintessence of Clear's semantics; the semantic equations given in Sect. 8 serve only to attach a syntax to the operations defined here. The definitions depend heavily upon the special representation of based theories described in Sect. 5; the objects defined in Sect. 2 are used as well (signatures, equations, constraints) but their representations are not important.

Definition. If $\Sigma = \langle S, \Omega \rangle$ and $\Sigma' = \langle S', \Omega' \rangle$ are tagged signatures then the *union* of Σ and Σ', written $\Sigma \cup \Sigma'$, is $\langle S \cup S', \tilde{\Omega} \cup \tilde{\Omega}' \rangle$ (where $\tilde{\Omega}$ and $\tilde{\Omega}'$ are the extensions of Ω and Ω' to indexed sets of operators over $S \cup S'$).

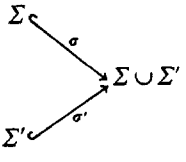
7.1. *Combine*

This implements the "+" theory-building operation of Clear.

combine: based-theory × based-theory → based-theory

$\text{combine}(\langle \Sigma, EC \rangle_B, \langle \Sigma', EC' \rangle_{B'}) = \langle \Sigma \cup \Sigma', \overline{\sigma(EC) \cup \sigma'(EC')} \rangle_{B \cup B'}$

where σ and σ' are the signature inclusions



We will sometimes use "+" in the sequel rather than *combine*; this should cause no confusion.

The result has the sorts and operators of both theories, the closed union of the axioms (translated to give $\Sigma \cup \Sigma'$-equations and -constraints), and the union of the two bases. Since Σ and Σ' are tagged signatures, the union $\Sigma \cup \Sigma'$ respects shared sorts and operators.

7.2. Enrich

An enrichment consists of some new sorts, operators (with their types) and equations. The *enrich* operation takes a based theory and an enrichment and produces the enriched based theory. Each new sort and operator must be given a unique tag, according to the discussion in Sect. 5. This tagging is not done by the *enrich* operation itself; we require that new sorts and operators be given unique tags *before* they are used to enrich a theory. This is necessary to avoid complications in cases where the type of a new operator includes both old and new sorts. The tags are attached by the semantic equations (as part of the semantics of sort and operator declarations - Sect. 8.3).

enrich: based-theory
$\times$ tagged-sort-set $\times$ (tagged-operator $\times$ type)-set $\times$ equation-set
$\rightarrow$based-theory

$\text{enrich}(\langle \Sigma, EC\rangle_B, S', \Omega', E') = \langle \Sigma \cup \langle S', \Omega'\rangle, \overline{\sigma(EC) \cup E'}\rangle_B$

where Ω' is indexed over sorts$(\Sigma) \cup S'$
E' is a set of $\Sigma \cup \langle S', \Omega'\rangle$-equations
and σ is the signature inclusion

$$\Sigma \overset{\sigma}{\hookrightarrow} \Sigma \cup \langle S', \Omega'\rangle$$

7.3. Data Enrich

When a theory is enriched by some new **data**, the axioms of the resulting theory contain a data constraint describing the enrichment. Moreover, an equality predicate $==: s, s \rightarrow$bool for each new sort s is introduced. Otherwise the result is the same as for ordinary (non-**data**) enrich. We employ a model-theoretic approach to obtain the equations which specify the meaning of the new equality predicates.

Definition. Suppose Σ is a tagged signature which includes the sort $bool_{Bool}$ and the operators $true_{Bool}$, $false_{Bool}: bool_{Bool}$, A is a Σ-algebra, EC is a set of Σ equations and constraints, x is a new tag, S is a subset of the sorts of Σ, and $s \in S$. Then:

- Σ^s_x is Σ with an additional operator $==_x: s, s \rightarrow \text{bool}_{\text{Bool}}$. Σ^S_x is defined similarly (i.e., an additional operator $==_x$ for each sort in S).

- A^s_x is a Σ^s_x-algebra just like A but with an operator $==_x$ satisfying $==_x(a, b) = \text{true}_{\text{Bool}}$ iff $a = b$, for all $a, b \in |A|_s$. A^S_x is defined similarly.

- EC^S_x is the set of Σ^S_x-equations and constraints given by M^*, where $M = \{A^S_x | A \in EC^*\}$.

If S is the set of new sorts and EC is the set of equations and constraints already in a theory, then EC^S_x includes EC as well as all the equations needed to define the new equality predicates on sorts in S.

data-enrich: based-theory
$\times$ tagged-sort-set $\times$ (tagged-operator $\times$ type)-set $\times$ equation-set
$\times$ tag$\rightarrow$based-theory

data-enrich$(\langle \Sigma, EC\rangle_B, S', \Omega', E', x) = \langle (\Sigma enr)_x^{S'}, \overline{(ECenr \cup \langle F, id_{\Sigma enr}\rangle)}_x^{S'}\rangle_{Benr}$

where $\langle \Sigma enr, ECenr\rangle_{Benr} = \text{enrich}(\langle \Sigma, EC\rangle_B, S', \Omega', E')$

and F is the equational theory inclusion

$$\langle \Sigma, \bar{\phi}\rangle \xrightarrow{F} \langle \Sigma enr, \bar{E}'\rangle$$

data-enrich gives an error if Σenr does not include $bool_{Bool}$ and $true_{Bool}$, $false_{Bool}: bool_{Bool}$.

The result is the same as the result of *enrich*, with the addition of an operator $= =$ for each new sort, the equations concerning those operators, and the data constraint $\langle F, id_{\Sigma enr}\rangle$ where F is the equational theory inclusion describing the enrichment.

7.4. *Derive*

The *derive* operation is used to "forget" sorts and operators of a theory, possibly renaming the ones remaining. The renaming is accomplished by a signature morphism which takes the new names into the old names. Given a Σ-theory, a Σ'-theory and a signature morphism σ: $\Sigma \to \Sigma'$, *derive* produces a theory with the signature and base of the Σ-theory, and all the Σ-equations and constraints which are satisfied in all models of the Σ'-theory - that is, the inverse image under σ of the equations and constraints of the Σ'-theory.

derive: based-theory × signature-morphism × based-theory → based-theory

derive$(\langle \Sigma, EC\rangle_B, \sigma, \langle \Sigma', EC'\rangle_{B'}) = \langle \Sigma, \sigma^{-1}(EC')\rangle_B$

where $\sigma^{-1}(EC') = \{e \mid \sigma(e) \in EC'\}$

derive gives an error if σ: $\langle \Sigma, EC\rangle_B \to \langle \Sigma', EC'\rangle_{B'}$ is not a based theory morphism.

The result is a theory because of the following fact:

Fact (see [5]): If EC is closed then $\sigma^{-1}(EC)$ is closed.

Also, $EC \subseteq \sigma^{-1}(EC')$ since σ is a theory morphism.

7.5. *Apply*

Apply defines the meaning of applying a parameterised specification to its arguments. A parameterised specification is represented as a based theory (the body) together with a list of based theories (the requirements). This is the first argument of *apply*; the second is a list of (based-theory × signature-morphism)-pairs (actual parameter × fitting morphism). The third argument is a tag to be attached to the "new" sorts and operators, and the fourth argument is the present constant theory environment.

apply: (based-theory × based-theory*) [parameterised specification]
× (based-theory × signature-morphism)* [parameters]
× tag
× environment → based-theory

The definition of *apply* uses two auxiliary functions. The first applies a signature morphism $\sigma: \Sigma A \to \Sigma B$ to a theory T with a signature Σ which includes ΣA; the sorts and operators in Σ but not in ΣA are not affected. This is used to apply a fitting morphism to the body of a parameterised specification, and is also useful in defining the second auxiliary function.

_ **altered by** _ : theory × signature-morphism → theory

Suppose $\Sigma = \langle S, \Omega \rangle$, $\Sigma A = \langle SA, \Omega A \rangle$, $\Sigma B = \langle SB, \Omega B \rangle$ and $\langle \sigma_{sorts}, \sigma_{opns} \rangle = \sigma: \Sigma A \to \Sigma B$. Then:

$\langle \Sigma, EC \rangle$ **altered by** $\sigma = \langle \Sigma', \overline{\sigma'(EC)} \rangle$

where Σ' and σ' are constructed as follows:

for $s \in S$, let $\sigma'_{sorts}(s) = \begin{cases} \sigma_{sorts}(s) \text{ if } s \in SA \\ s \text{ otherwise} \end{cases}$

let $S' = \{\sigma'_{sorts}(s) \mid s \in S\}$

for $w \in S^*$, $s \in S$ and $f: w \to s$ in Ω,

let $(\sigma'_{opns})_{ws}(f) = \begin{cases} (\sigma_{opns})_{ws}(f) \text{ if } f: w \to s \text{ is in } \Omega A \\ f \text{ otherwise} \end{cases}$

for $w' \in S'^*$ and $s' \in S'$, let $\Omega'_{w's'} = \bigcup_{\langle w, s \rangle \in J} \{(\sigma'_{opns})_{ws}(f) \mid f: w \to s \text{ in } \Omega\}$

where $J = \{\langle w \in S^*, s \in S \rangle \mid (\sigma'_{sorts})^*(ws) = w's'\}$

then $\Sigma' = \langle S', \Omega' \rangle$

and $\sigma': \Sigma \to \Sigma' = \langle \sigma'_{sorts}, \sigma'_{opns} \rangle$

an error results if $\Sigma A \not\subseteq \Sigma$

Informally, $\langle \Sigma, EC \rangle$ **altered by** σ just replaces the sorts and operators of Σ which are in ΣA by their images in ΣB.

The second auxiliary function attaches a given new tag to all of the sorts and operators in a theory, excluding those sorts and operators which belong to a distinguished subsignature.

_ **retagged with** _ **preserving** _ : theory × tag × signature → theory

$\langle \Sigma, EC \rangle$ **retagged with** x **preserving** $\Sigma' = \langle \Sigma, EC \rangle$ **altered by** *mtag*

where *mtag* is a signature morphism which gives each of the sorts and operators in $\Sigma - \Sigma'$ the tag x

an error results if $\Sigma' \not\subseteq \Sigma$

Apply is now defined with the help of these two functions. The idea is to first attach the given new tag to each sort and operator in the body of the parameterised specification, excluding those belonging to a requirement theory or base theory. This is necessary so that (for example) the sort *set* in the theory *Set*(*Nat*[...]) will always remain distinct from the sort *set* in *Set*(*Bool*[...]) as discussed in Sect. 5. The fitting morphisms are then applied to change each reference to the requirement signature into the corresponding reference to a sort or operator in the signature of the actual parameter, and the base of the parameterised specification is attached. Finally, the actual parameters are added using *combine* to give the result. An error results unless all the fitting morphisms are based theory morphisms.

$$\begin{aligned}&\text{apply}(\langle P_{BP}, \langle M_1, \ldots, M_n\rangle\rangle, \langle\langle A_1, \sigma_1\rangle, \ldots, \langle A_n, \sigma_n\rangle\rangle, x, \rho)\\ &\quad = A_1 + \ldots + A_n + ((P \textbf{ retagged with } x \textbf{ preserving } \Sigma old)\\ &\qquad\qquad \textbf{altered by } \sigma_1 \cup \ldots \cup \sigma_n)_{BP}\\ &\text{where } \Sigma old = \text{sig}(M_1) \cup \ldots \cup \text{sig}(M_n) \cup \bigcup_{TN \in BP} \text{sig}(\rho(TN))\end{aligned}$$

apply gives an error if some σ_i: $M_i \to A_i$ is not a based theory morphism.

This construction is rather more elaborate than any of those given previously. In order to understand it, consider first the simple case in which theories contain only sorts (no operators or equations/constraints) and the parameterised specification has only one argument. For example:

$P =$ **sorts** $\text{bool}_{\text{Bool}}, \text{m}_{\text{M}}, \text{nat}_{\text{Nat}}, \text{p}_{\text{P}}$	**base** Bool, Nat
$M =$ **sorts** $\text{bool}_{\text{Bool}}, \text{m}_{\text{M}}$	**base** Bool
$A =$ **sorts** $\text{bool}_{\text{Bool}}, \text{a}_{\text{A}}, \text{a}'_{\text{A}}$	**base** Bool, A
$\sigma = [\text{bool}_{\text{Bool}} \mapsto \text{bool}_{\text{Bool}}, \text{m}_{\text{M}} \mapsto \text{a}_{\text{A}}]$	

Now let us evaluate $apply(\langle P, M\rangle, \langle A, \sigma\rangle, J36, \rho)$ where ρ is an environment including (at least) Bool, Nat and A. The "old" sorts upon which P was built (Σold) are:

sorts $\text{bool}_{\text{Bool}}, \text{m}_{\text{M}}, \text{nat}_{\text{Nat}}$

Retagging P (without its base) while preserving Σold gives:

sorts $\text{bool}_{\text{Bool}}, \text{m}_{\text{M}}, \text{nat}_{\text{Nat}}, \text{p}_{J36}$

This is exactly P except that the sort p (which is "new" in P) is tagged with $J36$ to ensure that it remains distinct from the sort p in the application of P to some other parameter. Applying the fitting morphism σ and reattaching the base of P gives:

sorts $\text{bool}_{\text{Bool}}, \text{a}_{\text{A}}, \text{nat}_{\text{Nat}}, \text{p}_{J36}$ **base** Bool, Nat

and combining this with the actual parameter A gives the final result:

sorts $\text{bool}_{\text{Bool}}, \text{a}_{\text{A}}, \text{nat}_{\text{Nat}}, \text{p}_{J36}, \text{a}'_{\text{A}}$ **base** Bool, Nat, A

For a more difficult example, consider the specification of Sect. 4. According to the semantic equations (see Sect. 8), the denotation of the expression

Set(Nat[element **is** nat, eq **is** $==$])

is the result of evaluating $apply(\langle P, M\rangle, \langle A, \sigma\rangle, J37, \rho)$, where $\langle P, M\rangle$ is the denotation of the parameterised specification Set (P is the body and M is the requirement Ident), A and σ are the denotations of the argument Nat and the fitting morphism respectively, $J37$ is some new tag, and ρ is the constant theory environment.

The denotation of Nat is the following based theory (ignoring equations and constraints):

sorts $\text{bool}_{\text{Bool}}, \text{nat}_{\text{Nat}}$
opns $0_{\text{Nat}}, \text{succ}_{\text{Nat}}, ==_{\text{Nat}}, \text{true}_{\text{Bool}}, \ldots$
eqns ...
base Bool, Nat

Ident has the following denotation:

sorts $\text{bool}_{\text{Bool}}$, $\text{element}_{\text{Ident}}$
opns eq_{Ident}, $\text{true}_{\text{Bool}}$, ...
eqns ...
base Bool

Note that since Ident is a metatheory, it is not recorded in the bases of theories (like Ident itself) with contain it. The body of Set denotes the following based theory:

sorts $\text{bool}_{\text{Bool}}$, $\text{element}_{\text{Ident}}$, set_{Set}
opns ϕ_{Set}, $\text{singleton}_{\text{Set}}$, eq_{Ident}, $\text{true}_{\text{Bool}}$, ...
eqns ...
base Bool

The constant theory environment contains Bool and Nat (and Boolopns); Ident and Set are in the metatheory and parameterised specification environments, respectively.

Referring to the definition of *apply*, the value of Σold is:

sorts $\text{bool}_{\text{Bool}}$, $\text{element}_{\text{Ident}}$
opns eq_{Ident}, $\text{true}_{\text{Bool}}$, ...

Retagging P (without its base) with the new tag $J37$ while preserving Σold gives:

sorts $\text{bool}_{\text{Bool}}$, $\text{element}_{\text{Ident}}$, set_{J37}
opns ϕ_{J37}, singleton_{J37}, eq_{Ident}, $\text{true}_{\text{Bool}}$, ...
eqns ...

Applying the fitting morphism $[\text{element}_{\text{Ident}} \mapsto \text{nat}_{\text{Nat}}, \text{eq}_{\text{Ident}} \mapsto ==_{\text{Nat}}]$ to this theory and reattaching the base of Set yields:

sorts $\text{bool}_{\text{Bool}}$, nat_{Nat}, set_{J37}
opns ϕ_{J37}, singleton_{J37}, $==_{\text{Nat}}$, $\text{true}_{\text{Bool}}$, ...
eqns ...
base Bool

Finally, this is combined with the actual parameter Nat to give the answer:

sorts $\text{bool}_{\text{Bool}}$, nat_{Nat}, set_{J37}
opns ϕ_{J37}, singleton_{J37}, 0_{Nat}, succ_{Nat}, $==_{\text{Nat}}$, $\text{true}_{\text{Bool}}$, ...
eqns ...
base Bool, Nat

Note that applying a parameterised specification P with formal parameter X and requirement M to an argument A using a fitting morphism σ is the same (because of the restriction that P must include M) as rewriting the body of the parameterised specification, with A substituted for X and all occurrences of sorts and operators in M translated using σ to the matching bits of A. The definition of *apply* simulates this rewriting, using the trick of attaching fresh

tags to the sorts and operators which are "new in P" (i.e. not included in the base or requirement theories) to distinguish them from the corresponding objects produced in a different application of the same parameterised specification.

7.6. *Copy*

The *copy* operation is used to make a fresh copy of a theory, preserving a given set of subtheories.

copy: based-theory × based-theory × tag → based-theory
copy$(T_B, \langle \Sigma', EC' \rangle_{B'}, x) = (T$ **retagged with** x **preserving** $\Sigma')_{B \cap B'}$

Given two based theories (the second theory is the combination of the subtheories to be shared), *copy* simply gives the new tag x to the sorts and operators of the first theory which are not in the second theory. The base of the result is the intersection of the bases of the argument theories.

7.7. *Copy-meta*

The *copy-meta* operation is used in the semantics of parameterised specification declaration. Consider the following declaration:

proc P(X: Ident, Y: Ident) = **enrich** $X + Y$ **by** ...

In cases like these (i.e. whenever a multiple-parameter specification has requirements which share non-constant subparts) the following operation is used to make fresh copies of the requirement theories while preserving any constant subtheories:

$$\text{copy-meta}(T_{\{TN_1, \ldots, TN_m\}}, x, \rho) = \text{copy}(T_{\{TN_1, \ldots, TN_m\}}, \rho(TN_1) + \ldots + \rho(TN_m), x)$$

8. Semantic Equations

Below are the semantic equations which associate the syntactic constructs of Clear with their semantics. The equations are divided into several levels. Level I deals with the semantics of sort and operator names, and depends on the notion of a *dictionary*. Level IIa contains the semantics of enrichments (sort and operator declarations, and equations), and level IIb describes signature changes (used in **derive** and in application of a parameterised specification). Finally, level III gives the semantics of Clear's theory-building operations and declarations, based on levels IIa and IIb and the operations on based theories defined in Sect. 7. Much of the material in this section is taken from [5], although there are some corrections and many minor changes.

8.1. Dictionaries

In Clear the notation s **of** TN (where s is a sort name and TN is a theory name) may be used to refer to a sort which is included in a subtheory TN of the current theory (similarly o **of** TN for operators). This may be necessary if the sort (or operator) name alone is ambiguous. A dictionary gives the correspondence between such an expression and the tagged sort or operator to which it refers.

Definition. A *dictionary* is a pair of functions $\langle sd, od \rangle$ where

sd: sort-name × theory-name → tagged-sort
od: operator-name × theory-name → tagged-operator

The operation *dict* is used to construct a dictionary from a based theory; the resulting dictionary interprets sort and operator expressions referring to sorts and operators in that theory.

dict: based-theory × environment → dictionary
$\text{dict}(T_B, \rho) = \langle sd, od \rangle$
where $sd(s, TN)$ = the unique tagged sort with name s in $\rho(TN)$
and $od(o, TN)$ = the unique tagged operator with name o in $\rho(TN)$

$sd(s, TN)$ gives an error if $TN \notin B$, or if there is no unique sort called s in $\rho(TN)$ (similarly for $od(o, TN)$).

Note that this definition says that the notation s **of** TN (similarly o **of** TN) may only be used to refer to theories which are in the base of the current theory.

8.2. Level 1: Sorts, Operators and Terms

Syntactic categories

s: sort name	(lower case identifier)
o: operator name	(identifier or operator symbol)
TN: theory name	(capitalised identifier)
sex: sort expression	
oex: operator expression	
x: variable	(identifier)
tex: term expression	

Syntax

$\text{sex} ::= s \mid s$ **of** TN	e.g. element **of** X
$\text{oex} ::= o \mid o$ **of** TN	e.g. not **of** Bool
$\text{tex} ::= x \mid \text{oex}(\text{tex}_1, \ldots, \text{tex}_n)$	e.g. or(true **of** Bool, p) (infixes etc. also permitted)

Values

d: dictionary
X: sort-indexed variable set
tm: term

Semantic functions

Sex: sort-expression→signature→dictionary→tagged-sort
Oex: operator-expression→signature→dictionary→tagged-operator
Tex: term-expression→signature→dictionary→sorted-variable-set→term

Semantic equations

Sex⟦s⟧Σd = the unique tagged sort in sorts(Σ) with name s
Sex⟦s **of** TN⟧$\Sigma d = sd(s, TN)$ *where* $\langle sd, od\rangle = d$

Oex⟦o⟧Σd = the unique tagged operator in operators(Σ) with name o
Oex⟦o **of** TN⟧$\Sigma d = od(o, TN)$ *where* $\langle sd, od\rangle = d$

Tex⟦x⟧$\Sigma dX = x$ (a Σ-term on X) if $x \in X$ *else* **error**
Tex⟦oex($\text{tex}_1, \ldots, \text{tex}_n$)⟧$\Sigma dX =$
let f = Oex⟦oex⟧Σd *in*
let $tm_1, \ldots, tm_n$ = Tex⟦tex_1⟧$\Sigma dX, \ldots,$ Tex⟦tex_n⟧ΣdX *in*
$f(tm_1, \ldots, tm_n)$ (a Σ-term on X)

8.3. Level IIa: Enrichments

Syntactic categories

sd: sort declaration
od: operator declaration
varl: variable list
eq: equation expression
enrb: enrichment body
enr: enrichment

Syntax

sd::= s	e.g. nat
od::= o: $\text{sex}_1, \ldots, \text{sex}_n \to \text{sex}$	e.g. <: nat, nat→bool
varl::= $x_{11}, \ldots, x_{1n_1}$: $\text{sex}_1, \ldots, x_{m1}, \ldots, x_{mn_m}$: sex_m	e.g. i, j:nat, p:bool
eq::= **all** varl. $\text{tex}_1 = \text{tex}_2$	e.g. **all** p:nat. $p+0=p$
enrb::= **sorts** $\text{sd}_1, \ldots, \text{sd}_m$ **opns** $\text{od}_1 \ldots \text{od}_n$ **eqns** $\text{eq}_1 \ldots \text{eq}_p$	
enr::= enrb | **data** enrb	

The operator declaration o: *sex* is permitted as an abbreviation for o: →*sex*. Furthermore, the notation

$o_1, \ldots, o_m$: $\text{sex}_1, \ldots, \text{sex}_n \to \text{sex}$

is allowed for operator declarations, defined by the obvious expansion into a sequence of declarations, and the notation $\text{tex}_1 = \text{tex}_2$ is allowed as an abbreviation for **all**. $\text{tex}_1 = \text{tex}_2$ (ground equation).

Semantic functions

Sd: sort-declaration→tag→tagged-sort
Od: operator-declaration→tag→signature→dictionary
→(tagged-operator × type)

Varl: variable-list→signature→dictionary→sorted-variable-set
Eq: equation-expression→signature→dictionary→equation
Enrb: enrichment-body→tag→signature→dictionary
→(tagged-sort-set × (tagged-operator × type)-set × equation-set)
Enr: enrichment→tag→based-theory→dictionary→based-theory

Semantic equations

$\mathrm{Sd}[\![s]\!]x = s_x$
$\mathrm{Od}[\![o\colon \mathrm{sex}_1,\ldots,\mathrm{sex}_n\to\mathrm{sex}]\!]x\Sigma d =$
let $s_1,\ldots,s_n,\ s = \mathrm{Sex}[\![\mathrm{sex}_1]\!]\Sigma d,\ldots,\mathrm{Sex}[\![\mathrm{sex}_n]\!]\Sigma d, \mathrm{Sex}[\![\mathrm{sex}]\!]\Sigma d$ *in*
$\langle o_x, \langle\langle s_1,\ldots,s_n\rangle, s\rangle\rangle$
$\mathrm{Varl}[\![x_{11},\ldots,x_{1n_1}\colon\mathrm{sex}_1,\ldots,x_{m1},\ldots,x_{mn_m}\colon\mathrm{sex}_m]\!]\Sigma d =$
let $s_1,\ldots,s_m = \mathrm{Sex}[\![\mathrm{sex}_1]\!]\Sigma d,\ldots,\mathrm{Sex}[\![\mathrm{sex}_m]\!]\Sigma d$ *in*
$\{\langle x_{11},s_1\rangle,\ldots,\langle x_{1n_1},s_1\rangle,\ldots\langle x_{m1},s_m\rangle,\ldots,\langle x_{mn_m},s_m\rangle\}$
$\mathrm{Eq}[\![\textbf{all varl. } \mathrm{tex}_1 = \mathrm{tex}_2]\!]\Sigma d =$
let $X = \mathrm{Varl}[\![\mathrm{varl}]\!]\Sigma d$ *in*
let $tm_1, tm_2 = \mathrm{Tex}[\![\mathrm{tex}_1]\!]\Sigma d X, \mathrm{Tex}[\![\mathrm{tex}_2]\!]\Sigma d X$ *in*
$\forall X.\ tm_1 = tm_2$ (a Σ-equation)
$\mathrm{Enrb}[\![\textbf{sorts } \mathrm{sd}_1,\ldots,\mathrm{sd}_m \textbf{ opns } \mathrm{od}_1\ldots\mathrm{od}_n \textbf{ eqns } \mathrm{eq}_1\ldots\mathrm{eq}_p]\!]x\Sigma d =$
let $S' = \{\mathrm{Sd}[\![\mathrm{sd}_1]\!]x,\ldots,\mathrm{Sd}[\![\mathrm{sd}_m]\!]x\}$ *in*
let $\Sigma' = \Sigma \cup \langle S', \phi\rangle$ *in*
let $\Omega' = \{\mathrm{Od}[\![\mathrm{od}_1]\!]x\Sigma' d,\ldots,\mathrm{Od}[\![\mathrm{od}_n]\!]x\Sigma' d\}$ *in*
let $\Sigma'' = \Sigma' \cup \langle \phi, \Omega'\rangle$ *in*
let $E' = \{\mathrm{Eq}[\![\mathrm{eq}_1]\!]\Sigma'' d,\ldots,\mathrm{Eq}[\![\mathrm{eq}_e]\!]\Sigma'' d\}$ *in*
$\langle S', \Omega', E'\rangle$
$\mathrm{Enr}[\![\mathrm{enrb}]\!]xTd = \mathrm{enrich}(T, \mathrm{Enrb}[\![\mathrm{enrb}]\!]x\,\mathrm{sig}(T)d)$
$\mathrm{Enr}[\![\textbf{data } \mathrm{enrb}]\!]xTd = \mathrm{data\text{-}enrich}(T, \mathrm{Enrb}[\![\mathrm{enrb}]\!]x\,\mathrm{sig}(T)d, x)$

8.4. Level IIb: Signature Changes

Syntactic categories

sc: sort change
oc: operator change
sic: signature change

Syntax

$\mathrm{sc} ::= s_1$ **is** $\mathrm{sex}_1,\ldots,s_n$ **is** sex_n
$\mathrm{oc} ::= o_1$ **is** $\mathrm{oex}_1,\ldots,o_n$ **is** oex_n
sic::=sc, oc

e.g. element **is** nat,
order **is** < **of** Nat

Semantic functions

Sc: sort-change→signature→signature→dictionary
→(tagged-sort→tagged-sort)
Oc: operator-change→signature→signature→dictionary
→(tagged-operator→tagged-operator)
Sic: signature-change→signature→signature→dictionary
→signature-morphism

Semantic equations

$\mathrm{Sc}[\![s_1 \text{ is } \mathrm{sex}_1, \ldots, s_n \text{ is } \mathrm{sex}_n]\!]\Sigma\Sigma' d' =$
$\{\langle \mathrm{Sex}[\![s_1]\!]\Sigma d, \mathrm{Sex}[\![\mathrm{sex}_1]\!]\Sigma' d'\rangle, \ldots, \langle \mathrm{Sex}[\![s_n]\!]\Sigma d, \mathrm{Sex}[\![\mathrm{sex}_n]\!]\Sigma' d'\rangle\}$
where $d = \langle \phi, \phi \rangle$ (the null dictionary)

$\mathrm{Oc}[\![o_1 \text{ is } \mathrm{oex}_1, \ldots, o_n \text{ is } \mathrm{oex}_n]\!]\Sigma\Sigma' d' =$
$\{\langle \mathrm{Oex}[\![o_1]\!]\Sigma d, \mathrm{Oex}[\![\mathrm{oex}_1]\!]\Sigma' d'\rangle, \ldots, \langle \mathrm{Oex}[\![o_n]\!]\Sigma d, \mathrm{Oex}[\![\mathrm{oex}_n]\!]\Sigma' d'\rangle\}$
where $d = \langle \phi, \phi \rangle$ (the null dictionary)

$\mathrm{Sic}[\![\mathrm{sc}, \mathrm{oc}]\!]\Sigma\Sigma' d' = \text{make-signature-morphism}(\Sigma, \mathrm{Sc}[\![\mathrm{sc}]\!]\Sigma\Sigma' d', \mathrm{Oc}[\![\mathrm{oc}]\!]\Sigma\Sigma' d', \Sigma')$
where make-signature-morphism$(\Sigma, \sigma_{sorts}, g, \Sigma')$ is the signature morphism $\langle \sigma_{sorts}, \sigma_{opns} \rangle$: $\Sigma \to \Sigma'$ with $(\sigma_{opns})_{ws}$ the set of all pairs $\langle f, f' \rangle \in g$ such that f: $w \to s$ is in Ω

8.5. Environments

Reference has already been made in previous sections to an environment of theories. In that case the reference was to the *constant* theory environment, only one of the three environments which will be needed. This is a function binding names to based theories. The other two environments store metatheory and parameterised specification bindings; the metatheory environment is again a function binding names to based theories, while in the parameterised specification environment each name is bound to a value consisting of a based theory (the body) together with a list of based theories (the requirement theories).

Several operations are defined for manipulating these environments. The operation

bind: name × value × environment → environment

returns an environment with an added association between the name and value given (the type of value depends on the environment). Similarly,

bind: name* × value* × environment → environment

binds a list of names to the corresponding elements in a list of values. These operations suffice for binding names to values in the metatheory and parameterised specification environments, but adding bindings to the constant theory environment involves a slight complication. Recall that the base of a theory contains the names of its constant subtheories, and that the constant subtheories of a constant theory include the theory itself. This implies that the base of every theory in the constant theory environment should include the name to which the theory itself is bound. This addition to the base is made when a new binding is added to the constant theory environment using the following operation:

bind-const: name × based-theory × constant-theory-environment
→ constant-theory-environment

bind-const$(TN, T_B, \rho) =$ bind$(TN, T_{B \cup \{TN\}}, \rho)$

There is an analogous operation for binding a list of names to the corresponding elements in a list of based theories:

bind-const: name* × based-theory* × constant-theory-environment
→constant-theory-environment
bind-const($\langle TN1, \ldots, TNm\rangle$, $\langle T1_{B1}, \ldots, Tm_{Bm}\rangle$, ρ) =
bind($\langle TN1, \ldots, TNm\rangle$, $\langle T1_{B1 \cup \{TN1\}}, \ldots, Tm_{Bm \cup \{TNm\}}\rangle$, ρ)

Since each of the three environments is a function from names to values, bindings can be retrieved using function application, i.e. $\rho(TN)$ is the value bound to TN in ρ.

8.6. *Level III: Theory Building Operations*

Let $\mathscr{I}$ be a countably infinite list of distinct tags. This is where the tags required by the representation of based theories discussed in Sect. 5 come from. The functions

hd: tag-list→tag
tl: tag-list→tag-list
split: tag-list × nat→(tag-list)*

are defined by the following axioms:

$\text{hd}[x_1 x_2 \ldots] = x_1$
$\text{tl}[x_1 x_2 \ldots] = [x_2 \ldots]$
$\text{split}\langle [x_1 x_2 \ldots], n\rangle = \langle [x_1 x_{n+1} x_{2n+1} \ldots], [x_2 x_{n+2} x_{2n+2} \ldots], \ldots, [x_n x_{2n} x_{3n} \ldots]\rangle$

These functions are used in the semantic equations below to provide tags and lists of tags wherever they are required. All these tags originate from $\mathscr{I}$ and are hence distinct.

Syntactic categories

PN: parameterised specification name (capitalised identifier)
e: expression
spec: specification

Syntax

e::= TN | **theory** enr **endth**
| e1 + e2
| **enrich** e **by** enr **enden**
| **derive** enr **using** $e_1, \ldots, e_n$ **from** e **by** sic **endde**
| $PN(e_1[\text{sic}_1], \ldots, e_n[\text{sic}_n])$
| **let** $TN = e_1$ **in** e_2
| **copy** e **using** $e_1, \ldots, e_n$
spec::= e | **const** TN = e spec
| **meta** TN = e spec
| **proc** $PN(TN_1 : e_1, \ldots, TN_n : e_n)$ = e spec
e.g. **const** Bool = **theory**...**endth**
meta Triv = **theory**...**endth**
proc String(X:Triv) = **theory**...**endth**
String(Bool[element **is** bool])

Values

T: based theory (sometimes M, P or A for metatheory, parameterised specification body or actual parameter respectively)
ρ: constant theory environment (name→based-theory)
μ: metatheory environment (name→based-theory)
π: parameterised specification environment
(name→based-theory × based-theory*)
L: tag-list

Semantic functions

E: expression→constant-theory-environment→metatheory-environment
→parameterised-specification-environment→tag-list
→based-theory
Spec: specification→constant-theory-environment→metatheory-environment
→parameterised-specification-environment→tag-list
→based-theory

Semantic equations

$$\mathrm{E}[\![TN]\!]\rho\mu\pi L = \begin{cases} \rho(TN) & \text{if } TN \in \mathrm{domain}(\rho) \\ \mu(TN) & \text{if } TN \in \mathrm{domain}(\mu) \\ \textbf{error} & \text{if } TN \text{ is in neither or both domains} \end{cases}$$

$\mathrm{E}[\![$**theory** enr **endth**$]\!]\rho\mu\pi L = \mathrm{Enr}[\![\mathrm{enr}]\!]\mathrm{hd}(L)\Phi\mathrm{dict}(\Phi, \rho)$
(Φ is the empty based theory)

$\mathrm{E}[\![\mathrm{e}_1 + \mathrm{e}_2]\!]\rho\mu\pi L =$
let $L_1, L_2 = \mathrm{split}(L, 2)$ *in*
$\mathrm{E}[\![\mathrm{e}_1]\!]\rho\mu\pi L_1 + \mathrm{E}[\![\mathrm{e}_2]\!]\rho\mu\pi L_2$

$\mathrm{E}[\![$**enrich** e **by** enr **enden**$]\!]\rho\mu\pi L =$
let $T = \mathrm{E}[\![\mathrm{e}]\!]\rho\mu\pi\mathrm{tl}(L)$ *in*
$\mathrm{Enr}[\![\mathrm{enr}]\!]\mathrm{hd}(L)T\mathrm{dict}(T, \rho)$

$\mathrm{E}[\![$**derive** enr **using** $\mathrm{e}_1, \ldots, \mathrm{e}_n$ **from** e **by** sic **endde**$]\!]\rho\mu\pi L =$
let $L_1, \ldots, L_{n+1} = \mathrm{split}(L, n+1)$ *in*
let $T = \mathrm{E}[\![\mathrm{e}_1]\!]\rho\mu\pi L_1 + \ldots + \mathrm{E}[\![\mathrm{e}_n]\!]\rho\mu\pi L_n$ *in*
let $T' = \mathrm{Enr}[\![\mathrm{enr}]\!]\mathrm{hd}(L_{n+1})T\mathrm{dict}(T, \rho)$ *in*
let $T'' = \mathrm{E}[\![\mathrm{e}]\!]\rho\mu\pi\mathrm{tl}(L_{n+1})$ *in*
let $\sigma = \mathrm{Sic}[\![\mathrm{sic}]\!]\mathrm{sig}(T')\mathrm{sig}(T'')\mathrm{dict}(T'', \rho)$ *in*
$\mathrm{derive}(T', \sigma, T'')$

$\mathrm{E}[\![PN(\mathrm{e}_1[\mathrm{sic}_1], \ldots, \mathrm{e}_n[\mathrm{sic}_n])]\!]\rho\mu\pi L =$
let $L_1, \ldots, L_{n+1} = \mathrm{split}(L, n+1)$ *in*
let $A_1, \ldots, A_n = \mathrm{E}[\![\mathrm{e}_1]\!]\rho\mu\pi L_1, \ldots, \mathrm{E}[\![\mathrm{e}_n]\!]\rho\mu\pi L_n$ *in*
let $\langle P, \langle M_1, \ldots, M_n \rangle\rangle = \pi(PN)$ *in*
let $\sigma_1, \ldots, \sigma_n = \mathrm{Sic}[\![\mathrm{sic}_1]\!]\mathrm{sig}(M_1)\mathrm{sig}(A_1)\mathrm{dict}(A_1, \rho),$
$\ldots$
$\mathrm{Sic}[\![\mathrm{sic}_n]\!]\mathrm{sig}(M_n)\mathrm{sig}(A_n)\mathrm{dict}(A_n, \rho)$ *in*
$\mathrm{apply}(\langle P, \langle M_1, \ldots, M_n \rangle\rangle, \langle\langle A_1, \sigma_1\rangle, \ldots, \langle A_n, \sigma_n\rangle\rangle, \mathrm{hd}(L_{n+1}), \rho)$

$\mathrm{E}[\![\textbf{let } TN = \mathrm{e}_1 \textbf{ in } \mathrm{e}_2]\!]\rho\mu\pi L =$
$\quad let\ L_1, L_2 = \mathrm{split}(L, 2)\ in$
$\quad let\ T = \mathrm{E}[\![\mathrm{e}_1]\!]\rho\mu\pi L_1\ in$
$\quad let\ \rho' = \text{bind-const}(TN, T, \rho)\ in$
$\quad let\ T'_B = \mathrm{E}[\![\mathrm{e}_2]\!]\rho'\mu\pi L_2\ in$
$\quad T'_{B-\{TN\}}$
$\mathrm{E}[\![\textbf{copy e using } \mathrm{e}_1, \ldots, \mathrm{e}_n]\!]\rho\mu\pi L =$
$\quad let\ L_1, \ldots, L_{n+2} = \mathrm{split}(L, n+2)\ in$
$\quad let\ T = \mathrm{E}[\![\mathrm{e}]\!]\rho\mu\pi L_1\ in$
$\quad let\ T' = \mathrm{E}[\![\mathrm{e}_1]\!]\rho\mu\pi L_2 + \ldots + \mathrm{E}[\![\mathrm{e}_n]\!]\rho\mu\pi L_{n+1}\ in$
$\quad \mathrm{copy}(T, T', \mathrm{hd}(L_{n+2}))$
$\mathrm{Spec}[\![\mathrm{e}]\!]\rho\mu\pi L = \mathrm{E}[\![\mathrm{e}]\!]\rho\mu\pi L$
$\mathrm{Spec}[\![\textbf{const } TN = \mathrm{e spec}]\!]\rho\mu\pi L =$
$\quad let\ L_1, L_2 = \mathrm{split}(L, 2)\ in$
$\quad let\ \rho' = \text{bind-const}(TN, \mathrm{E}[\![\mathrm{e}]\!]\rho\mu\pi L_1, \rho)\ in$
$\quad \mathrm{Spec}[\![\mathrm{spec}]\!]\rho'\mu\pi L_2$
$\mathrm{Spec}[\![\textbf{meta } TN = \mathrm{e spec}]\!]\rho\mu\pi L =$
$\quad let\ L_1, L_2 = \mathrm{split}(L, 2)\ in$
$\quad let\ \mu' = \mathrm{bind}(TN, \mathrm{E}[\![\mathrm{e}]\!]\rho\mu\pi L_1, \mu)\ in$
$\quad \mathrm{Spec}[\![\mathrm{spec}]\!]\rho\mu'\pi L_2$
$\mathrm{Spec}[\![\textbf{proc } PN(TN_1 : \mathrm{e}_1, \ldots, TN_n : \mathrm{e}_n) = \mathrm{e spec}]\!]\rho\mu\pi L =$
$\quad let\ L_1, \ldots, L_{n+2} = \mathrm{split}(L, n+2)\ in$
$\quad let\ M_1, \ldots, M_n = \text{copy-meta}(\mathrm{E}[\![\mathrm{e}_1]\!]\rho\mu\pi\mathrm{tl}(L_1), \mathrm{hd}(L_1), \rho),$
$\qquad\qquad \ldots$
$\qquad\qquad \text{copy-meta}(\mathrm{E}[\![\mathrm{e}_n]\!]\rho\mu\pi\mathrm{tl}(L_n), \mathrm{hd}(L_n), \rho)\ in$
$\quad let\ \rho' = \text{bind-const}(\langle TN_1, \ldots, TN_n\rangle, \langle M_1, \ldots, M_n\rangle, \rho)\ in$
$\quad let\ P_B = \mathrm{E}[\![\mathrm{e}]\!]\rho'\mu\pi L_{n+1}\ in$
$\quad let\ \pi' = \mathrm{bind}(PN, \langle P_{B-\{TN_1, \ldots, TN_n\}}, \langle M_1, \ldots, M_n\rangle\rangle, \pi)\ in$
$\quad \mathrm{Spec}[\![\mathrm{spec}]\!]\rho\mu\pi' L_{n+2}\ if\ \{TN_1, \ldots, TN_n\} \subseteq B\ else\ \textbf{error}$

The denotation of a specification *spec* in the initial environments ρ, μ, π is then given by the value of $\mathrm{Spec}[\![spec]\!]\rho\mu\pi\mathscr{I}$ (recall that $\mathscr{I}$ is an infinite supply of distinct tags). The initial constant theory environment ρ should normally include a binding of the theory name Bool to a theory containing (at least) the sort $bool_{Bool}$ and operators $true_{Bool}$ and $false_{Bool}$ with base {Bool}; these exact names are required by the *data-enrich* operation.

9. Conclusion

There are a number of algebraic specification languages besides Clear which have a formal semantics, including CIP-L [2], LOOK [8], ACT ONE [7], ASL [27] and the Larch Shared Language [18]. In comparison with these, the semantics of Clear in [5] as well as the one given here seem overly complex. The reason for this is that Clear attempts to take proper account of shared subtheories. A number of complications in the semantics are required to handle this feature, namely the addition of tags to sort and operator names (to indicate their theory of origin), of bases to theories (to keep track of constant subtheories) and the distinction between constant theories and metatheories. Frills like the automatic addition of an equality predicate for "data" sorts entail some complication as well, but here the effect is more localised.

We inherit from the semantics of [5] the problem of "proliferation" mentioned in Sect. 5 whereby each application of a parameterised specification gives a fresh copy of the resulting theory, and so e.g. $Set(Nat[...]) + Set(Nat[...])$ contains two sorts with the name *set*. A set-theoretic semantics of Clear which avoids this problem is given in [24]; the basic idea is to tag sorts and operators created by application of a parameterised specification with tags of the form $P(A[\sigma])$. Another problem we inherit from [5] concerns the semantics of **derive**. Intuitively, we would expect that to every model A of the specification **derive**...**using**...**from** T' **by** σ there should correspond a model A' of T' such that $A'|_\sigma \cong A$. This is not the case with the present definition of **derive**; the difficulty is that T' might contain data constraints which cannot be expressed in the signature of T. This situation could be put right by adopting a more elaborate definition of data constraint similar to the *canonical* constraints of [9]. But in the context of an arbitrary institution as in [5] it is necessary to descend to the level of models as in [27] to give a semantics of **derive** with the desired property.

By describing Clear under an arbitrary institution, the semantics of [5] is more general than the semantics of ordinary Clear given here. But it is easy to see that our semantics is independent of the definitions of model (algebra and satisfaction) and – with the exception of level IIa of the semantic equations (as in [5]) – of axiom (equation), so long as these definitions satisfy the simple consistency conditions required by an institution. The semantics does depend on the definitions of signature and signature morphism, as a consequence of the way tags are used to handle sharing. But (apart from the lower levels of the semantic equations, as in [5]) the semantics really only relies on the following essential features of their definitions:

– signatures are n-tuples of *sets*

– a signature morphism $\sigma: \Sigma \to \Sigma'$ consists of an n-tuple of *functions* (maps) between the components of Σ and Σ'.

In addition, the tagging trick depends on the following:

– enrichments denote theory *inclusions* (in [5] they may denote arbitrary theory morphisms)

The semantics could easily be modified to work for any institution satisfying these conditions. This means that essentially the same semantics works for (e.g.) Clear with errors [12, 11], ordered sorts [13], polymorphism [25], and/or partial algebras [3]. No institution has been proposed to my knowledge which does not satisfy the conditions above.

Acknowledgements. I am indebted to Rod Burstall and Joseph Goguen for [5] from which much of the material included here is borrowed. My thanks to Rod Burstall for guidance and encouragement, to David Rydeheard for category-theoretic expertise, and to Brian Monahan, Bill Wadge and Martin Wirsing for helpful comments. Support was provided by Edinburgh University and the Science and Engineering Research Council.

10. References

1. Ashcroft, E.A., Wadge, W.W.: R_x for semantics. TOPLAS 4, 283–294 (1982)
2. Bauer, F.L., Broy, M., Dosch, W., Gnatz, R., Geiselbrechtinger, F., Hesse, W., Krieg-Brückner, B., Laut, A., Matzner, T., Möller, B., Partsch, H., Pepper, P., Samelson, K., Wirsing, M., Wössner, H.: (the CIP Language Group) Report on a wide spectrum language for program specification and development. Report TUM-I8104, Technische Universität München, 1981
3. Broy, M., Wirsing, M.: Partial abstract types. Acta Informat. 18, 47–64 (1982)

4. Burstall, R.M., Goguen, J.A.: Putting theories together to make specifications. Proc. 5th Intl. Joint Conf. on Artificial Intelligence, Cambridge, pp. 1045–1058, 1977
5. Burstall, R.M., Goguen, J.A.: The semantics of Clear, a specification language. Proc. of Advanced Course on Abstract Software Specifications, Copenhagen. Springer LNCS 86, pp. 292–332, 1980
6. Burstall, R.M., Goguen, J.A.: An informal introduction to specifications using Clear. In: The Correctness Problem in Computer Science. R.S. Boyer, J.S. Moore (eds.). New York: Academic Press, pp. 185–213, 1981
7. Ehrig, H., Fey, W., Hansen, H.: ACT ONE: an algebraic specification language with two levels of semantics. Report Nr. 83-03, Institut für Software und Theoretische Informatik, Technische Universität Berlin, 1983
8. Ehrig, H., Thatcher, J.W., Lucas, P., Zilles, S.N.: Denotational and initial algebra semantics of the algebraic specification language LOOK Draft report, IBM research, 1982
9. Ehrig, H., Wagner, E.G., Thatcher, J.W.: Algebraic constraints for specifications and canonical form results (draft version). Report Nr. 82-09, Institut für Software und Theoretische Informatik, Technische Universität Berlin, 1982
10. Ehrig, H., Wagner, E.G., Thatcher, J.W.: Algebraic specifications with generating constraints. Proc. 10th ICALP, Barcelona. Springer LNCS 154, pp. 188–202, 1983
11. Gogolla, M., Drosten, K., Lipeck, U., Ehrich, H.D.: Algebraic and operational semantics of specifications allowing exceptions and errors. Fb. 140, Abteilung Informatik, Universität Dortmund, 1982
12. Goguen, J.A.: Abstract errors for abstract data types. Proc. IFIP Working Conf. on the Formal Description of Programming Concepts, New Brunswick, New Jersey, 1977
13. Goguen, J.A.: Order sorted algebras: exceptions and error sorts, coercions and overloaded operators. Semantics and Theory of Computation Report No. 14, Dept. of Computer Science, UCLA, 1978
14. Goguen, J.A., Burstall, R.M.: Introducing institutions. Proc. Logics of Programming Workshop. E. Clarke (ed.). Carnegie-Mellon University, 1983
15. Goguen, J.A., Meseguer, J.: An initiality primer. Draft report, SRI International, 1983
16. Goguen, J.A., Thatcher, J.W., Wagner, E.G.: An initial algebra approach to the specification, correctness, and implementation of abstract data types. IBM research report RC6487, 1976. Also in: Current Trends in Programming Methodology, Vol. 4: Data Structuring. R.T. Yeh (ed.). Englewood Cliffs, NJ: Prentice-Hall, pp. 80–149, 1978
17. Guttag, J.V.: The specification and application to programming of abstract data types. Ph.D. thesis, University of Toronto, 1975
18. Guttag, J.V., Horning, J.J.: Preliminary report on the Larch Shared Language. Report CSL-83-6. Computer Science Laboratory. Xerox PARC, 1983
19. Guttag, J.V., Horowitz, E., Musser, D.R.: Abstract data types and software validation. CACM **21**, 1048–1064 (1978)
20. Kaphengst, H., Reichel, H.: Algebraische Algorithmentheorie. VEB Robotron, Zentrum für Forschung und Technik, Dresden, 1971
21. McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., Levin, M.I.: LISP 1.5 Programmer's Manual. MIT Press, 1962
22. Nakajima, R., Yuasa, T.: The IOTA Programming System. Springer LNCS 160, 1983
23. Reichel, H.: Initially restricting algebraic theories. Proc. 9th MFCS, Rydzyna, Poland. Springer LNCS 88, pp. 504–514, 1980
24. Sannella, D.T.: Semantics, implementation and pragmatics of Clear, a program specification language. Ph.D. thesis, Dept. of Computer Science, University of Edinburgh, 1982
25. Sannella, D.T., Burstall, R.M.: Structured theories in LCF. Proc. 8th Colloq. on Trees in Algebra and Programming, L'Aquila, Italy. Springer LNCS 159, pp. 377–391, 1983
26. Sannella, D.T., Wirsing, M.: Implementation of parameterised specifications. Report CSR-103-82, Dept. of Computer Science, University of Edinburgh; extended abstract in: Proc. 9th ICALP, Aarhus, Denmark. LNCS 140, pp. 473–488, 1982
27. Sannella, D.T., Wirsing, M.: A kernel language for algebraic specification and implementation. Report CSR-131-83, Dept. of Computer Science, University of Edinburgh; extended abstract in: Proc. Intl. Conf. on Foundations of Computation Theory, Borgholm, Sweden. Springer LNCS 158, pp. 413–427, 1983

Received August 23, 1984