AD-A193 795 REVIEW OF TYPE CHECKING AND SCOPE RULES OF THE SPECIFICATION LANGUAGE Z(U) ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (ENGLAND) C T SENNETT NOV 87 1/1

UNCLASSIFIED RSRE-87017 DRIC-BR-105618 F/G 12/5 NL

END
DATE
FILMED
8 8

AD-A193 795

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Report 87017

Title: Review of the type checking and scope rules of the specification language Z

Author: C T Sennett

Date: November 1987

Summary

This report gives the detailed type checking and scope rules for the specification language Z in the form of an implementation specification for a type checking tool for Z, written in Z itself.

Copyright
©
Controller HMSO London
1987

CONTENTS

1. Introduction 1
2. Basic representations 5
3. The identifier environment 9
4. Unification of Z types 14
5. Normalisation of types 22
6. References to identifiers 24
7. Anonymous instantiation 31
8. Formation of instantiations 34
9. Named instantiation 38
10. Given set definitions and generic parameters 41
11. Declarations and inclusions 44
12. Syntactic, datatype and schema definitions 48
13. Operator and generic set definitions 51
14. Primitive types 57
15. Tuples, products, theta terms and comprehensions 61
16. Function application and partial application 65
17. Relations and predicates 69
18. Schema expressions 72
References 77
Appendix: the Z syntax 78

Accession For
NTIS GRA&I ☑
DTIC TAB ☐
Unannounced ☐
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

CHAPTER 1

INTRODUCTION

The purpose of this report is to review in detail the type checking and scope rules for the specification language Z. At present no definitive description of the language exists, although it is sufficiently well defined for the informal use which is currently made of it. The production of tools to process the language requires a complete definition during the production of which a number of decisions are made concerning various compromises between mathematical elegance and efficiency of implementation. It is the purpose of this report to review these in detail, using as a basis the syntax developed by King et al [1987, see also Sufrin 1986]. This will be referred to as the standard syntax. Rather than following this language definition mechanically some changes have been made, with the following motivations:

This report is concerned particularly with tool implementation, so a version of the syntax has been produced enabling syntax and type checking to be completed in one pass. Changes introduced under this heading do not affect the appearance of the language, but in some cases a syntactic check is replaced by a semantic check.

In some cases variations have been introduced which are a matter of personal preference. The production of variant languages in this way is actually a necessary step towards the goal of the production of a robust and usable language standard.

Most of the other cases consist not so much in change from what has been published elsewhere as a specification of the detailed meaning of the language for those areas which have not been described in detail, in particular the scope rules and the properties of the type system.

In contrast to the work of Spivey [1985], which is concerned with the formal semantics of the language, this report deals with aspects of the implementation of tools to process the language. Because of this, the report has been produced in the form of an implementation specification for a type checking tool for a language which bears a more than passing resemblance to Z but which represents the preferences of the author in those areas where the Z rules are debateable.

Z is based on typed set theory so that terms in Z have a type which corresponds to the largest set of which the term could be a member. This is not the same set for every term (the set of all sets) for the usual reasons, but instead types are associated with given sets, schemas and sets which may be constructed from them using the powerset and tuple constructors. Thus in contrast to programming languages, functions do not have a special type constructor but have the same type as relations (powerset of 2-tuples) which allows some useful expressions to be constructed without the need for special coercions.

From an implementation point of view, the most interesting aspect of type checking is in the limited polymorphism present in Z, in which some terms may have a generic type. In dealing with these terms the ideas of Milner [1978] have been followed and developed to cover the various constructions available in Z. The greatest difference in this area between Z and the language described in Milner's paper, or the related language ML, is that in ML all types are inferred whereas in Z the types (which may be polytypes) are given in a signature. This leads to rules for the handling of polymorphic signatures and also for the use of those signatures at positions where the corresponding identifiers are being defined. This will be discussed in more detail later.

The specification itself is interesting as an example of a Z specification for a reasonably large program; this report is a complete Z specification although it has not been passed through the tool it specifies as the implementation remains to be done. Consequently it no doubt contains errors, but the report is being issued now with a view to contributing to the debate on the precise form of the language. The production of the specification has been rewarding so it is worth recording some of

the reasons for feeling satisfied with the process. Apart from the obvious one of having a precise statement of the problem, these are as follows:

1. An extensive specification in Z may be produced quickly. This has a number of advantages. It is for example possible to understand the problem as a whole and design an appropriate module structure for the implementation without having to find this out the hard way at the implementation stage. This is particularly the case in the question of the design of data structures. A standard trauma in program development is to discover that the data structure one has been successfully using in the previous twenty modules does not have the capability to implement some feature required in the twenty first, leading to massive re-compilations. By having a complete specification for the whole problem, the capabilities required of the data structures can be made visible at the outset of implementation.

2. The formal specification is particularly useful when it comes to expressing error cases. There is an undoubted psychological reluctance to treating these properly and the fact that Z provides a compact notation for stating the error conditions as an increment to the standard case is an aid to overcoming this barrier.

3. The Z notation is an excellent means of communication between specifiers and implementers. The underlying set theory is easily understood and the notation is compact enough not to obscure the overall structure with irrelevant detail.

4. A particularly important part of what one might call the Z specification technique is the mathematical toolkit, the standard set of Z mathematical functions and operators which enable one to build specifications rapidly. Apart from the characteristic Z schema structures, the expressive power of the notation largely rests on this very useful library of functions.

5. Z is fun!

The structure of the design specification

The tool envisaged to meet this specification completes syntax analysis and type checking in one pass, so the specification must be for a set of compiling operations on the concrete syntax, rather than operations on the abstract syntax. A one-pass type checker will require declaration before use rules and a simple scheme of lexical analysis. No apology is offered for this, and none should be required by anyone who has suffered from trying to understand a specification where declaration before use does not apply.

The specification for each compiling operation must include the position within the syntax at which the operation is employed; the inputs to the operation in the form of lexical values (the identifiers encountered, the values of numerical constants etc); the state variables appropriate to the operation, most notably the identifier environment giving the relation between identifiers and their types; and finally, values constructed during the course of compilation, such as the types of sub-expressions. To indicate the relationship between these various items and the specification itself, the syntax notation employed in the syntax transforming tool SID will be used (Foster 1968, but see also Currie 1984). This may be briefly described by means of an example. The following fragment of input to SID gives a syntax for numerical expressions, together with compiling functions to evaluate the resulting integer.

```
BASICS

number          # decimal numbers assembled according to the usual
                  convention #
orb             # ( #
crb             # ) #
```

```
plus            # + #
minus           # - #
multiply        # * #
divide          # / #
```

RULES

```
expression  = expression addop term <opaction-pint-pint-pint--int>,
              term;

term        = term multop primary <opaction-pint-pint-pint--int>,
              primary;

primary     = <number-lv--int> number,
              addop primary <monadic-pint-pint--int>,
              orb expression crb;

anyop       = addop,
              multop;

addop       = <operator-1--int> plus,
              <operator-2--int> minus;

multop      = <operator-3--int> multiply,
              <operator-4--int> divide;
```

The first part of this fragment, under the heading BASICS lists the identifiers to be used to stand for the terminal symbols of the syntax. The syntax rules appear in the second part of the fragment, under the heading RULES: an equals sign terminates the name of the rule, a comma separates alternatives and a semi-colon terminates the definition of the rule. Each alternative within the definition is a sequence of rules or terminal symbols or compiling functions, the latter being indicated by angle brackets. SID is able to transform this syntax into a one-track form and outputs a program which will perform the appropriate syntax analysis. Where compiling functions have been included the analyser will call them at the appropriate place in the symbol stream: for example, in the rule for expression above, the function opaction will be called to form each intermediate result in an expression like 5+4+3.

Within the angle brackets, the name of each compiling function is followed bv strings involving minus signs. Each minus sign introduces a parameter to the function, a final double minus indicates the type of the result. The analyser stacks every result using a different stack for each type: thus opaction above leaves an integer on the stack of integers. Parameters to the compiling functions can only come from the stacks, so following each minus sign is the type of stack from which the value of the parameter is to be obtained and which will be supplied by the analyser when the function is called: the type will be preceded by a p or a q according to whether the value is to be supplied by a "pop" or a "top" operation. With this notation, it is useful to think of the syntax rules as delivering values onto the appropriate stack. Thus opaction in the rule for expression above takes the integer delivered by the term, the integer corresponding to the operation (+ or -) and the integer corresponding to the previous subexpression and combines them to produce a new integer which will be the result of the expression, which may be thought of as the result of the whole phrase.

There are two other forms of parameter which are allowed to compiling functions. These are -lv, in which case the lexical value of the symbol to the right will be supplied, and -n, where n is a small integer representing the value to be supplied. This latter case is used when a number of compiling functions are simple variations on a common theme: in this example the operator function presumably simply stacks its parameter to indicate to opaction which action is actually required.

After this lengthy excursion into the details of SID, it is now possible to give the conventions for specifying the compiling functions. These are:

1. Each compiling function is specified by a Z schema definition of the same name.

2. Each specification is preceded by the fragment of SID syntax which uses it.

3. Each parameter to a compiling function is represented as an input to the operation (using ?).

4. The result of each function is represented as an output (using !).

In the specification which follows, each chapter is a Z document. As the implementation is based on the Flex computing concepts [Foster et al 1982], which is an object oriented machine, documents are represented by module values, rather than the name of the document as in the standard syntax. These modules appear in the text as icons: Z_spec :Module, one for each document imported. The compiling functions for creating and using these module values are not defined in this specification as they are peculiar to the Flex architecture adopted. The Z syntax has also been extended to include an export statement in the form:

```
document keeps id, id,...
```

which indicates the identifiers made available when the document is incorporated.

Most of the strategy of type checking is discussed in the datatypes chapter which contains correspondingly more descriptive text compared with the other chapters which are concerned with the details of type checking. The appendix contains the complete syntax.

CHAPTER 2

BASIC REPRESENTATIONS

Identifiers

The lexical analyser has the task of sorting out base names and decorations and produces a member of the set Id, defined by a schema below, for every identifier encountered. The base name is represented by a line of characters, which may be emboldened, underlined or not:

```
weight ::= light | bold | underlined | underbold
decline ≙ [l : seq Char;  w :  weight]
Name ::= noname | line«decline»
```

In fact, as far as the specification is concerned, Name could be a given type, but this does at least indicate that lexical items are emboldened or not as a whole. Decoration is either a subscripted string (a version) or an attribute or both. The version is represented by a sequence of Name to allow for an arbitrarily complex label. An attribute is one of exclamation mark, query or a series of dashes: the view has been taken that these are mutually exclusive, so identifiers of the form x!! or x?! are illegal. Consequently it is possible to define a datatype Att to indicate the possible attributes of an identifier.

```
Att ::= noatt | bang | query | dashes«N»
```

The integer parameter of the dashes constructor is the number of dashes.

Each identifier has a syntactic status which is used by the lexical analyser to decide what sort of terminal symbol the identifier should be: the syntactic status is by default that of an ordinary identifier, but may be changed during the course of compiling a definition to be that of an infixed or other operator, or a generic set.

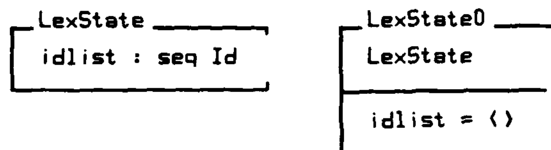
```
Synstatus ::= ident | op | encop«Name» | distinop«Name»
            | distpreop«Name» | preop | postop | rel
            | preset«Name» | postset«Name» | inset«seq Name»
```

The Name parameter of the constructors for the operators is the closing eop. For the sets, the name parameter indicates the generic parameter identifier or identifiers, used when the set is being instantiated. Identifiers are represented by the schema below, which indicates that only one version is allowed and version and attribute may be supplied in either order and represent the same identifier. That is, $x!_1$ is the same identifier as $x_1{}'$

```
┌─ Id ───
│ name, version : Name
│ att : Att; synstat : Synstatus
└──────────
```

Lexical values

The decoration of identifiers is handled by the lexical analyser, rather than syntactically, so the output from lexical analysis is an identifier, even when the decoration appears on its own (as in schema terms). It is convenient for the lexical analyser to buffer these identifiers and decorations in a global queue which forms the part of the lexical state visible to the compiling functions:



As a result, lexical values delivered by the SID generated syntax analyser need only distinguish integers, characters and strings, to handle explicit denotations for these values.

LexVal ::= num«Z» | char«Char» | string«seq Char»

Representation of types

A representation of types is proposed in Spivey [1985], but it has been found necessary to extend this, for three reasons:

1. It is necessary to cater for the distinction between generic and given types.

2. For the implementation of type checking in expressions it is necessary to infer the types of instantiation for generic identifiers. This has been done by the introduction of types constructed from type variables, which may be substituted by an inferred type value.

3. Also for the purposes of the implementation, the datatype has been extended to include predicates and an undefined type, which is used for undeclared identifiers.

Type variables are represented by type names TName, which refer to values in a type environment (which will have type TName ⇸ Type): substitutions are brought about by altering the type environment. The type names are introduced as a given set:

[TName]

```
Type ::= given«Id» | powerset «Type» | tuple«seq Type»
         | schema_type«Id ⇸ Type»
         | generic«(N × Id)» | variable«TName»
         | predicate | type_undefined
```

The elements of this disjoint union will be discussed in turn.

1. Given sets

A given set must be treated as atomic throughout the document, and may only be changed as a result of the instantiation of a previously compiled document. Consequently it must be distinguished from a generic type, which may be instantiated at different types within the document which defines it. The Id is the identifier of the given set, unique within the document. This type is also used for data types.

The types Z and Char are datatypes and built-in to the extent that numbers and strings are recognised as having the appropriate type. For the purposes of this specification, it will simply be asserted that these two types exist:

Ztype, Chartype : Type

It may be observed at this point that datatypes are one of the few constructions in Z which are not allowed to be generic. One can imagine a construction like

[T] Tree ::= leaf«T» | node«T Tree × T Tree»

for polymorphic tree structures for example, which would be useful. To cope with this, the type representation would have to be extended with a generic data type constructor dependent on a sequence of types (the generic parameters) and an identifier, the data type name. This identifier would have a value within the environment corresponding to a (preferably postfix) generic set instantiating a tuple as a parameter and delivering the appropriate type. This extension has not been made, mainly because the semantics of such data types have not been specified, but there seems to be no reason to suppose that this extension would introduce inconsistencies. This, of course, is not an argument for including it.

2. Powersets, tuples and schema types

These are standard type constructors, as given by Spivey.

3. Generic and variable

Generic types may either be instantiated by name or anonymously; in the latter case the type of their instantiation is inferred from their use, using the algorithm specified by Milner. To correspond to these two usages are two different representations, generic and variable. A generic type is constructed from an identifier corresponding to the generic parameter and an integer: the integer is for instantiation with a list of terms, rather than by name. A generic type is treated as atomic within the generic definition which uses it and elsewhere it is used to create the appropriate type on named instantiation. When, on the other hand, type instantiation is done by inference, a variable type is created from the generic type to allow substitution of the inferred type of instantiation for the generic type. The variable type is represented by a type name, which is used to refer to a type environment where the substitutions are actually made. By this means, one substitution accounts for all instances of the generic type within the type representation. (An example of a multiple instance is the identity relation which has type $\mathbb{P}(T \times T)$, where the T are generic. An instance of the identity must be inferred to have type $\mathbb{P}(\mathbb{Z} \times \mathbb{Z})$ as soon as either the domain or range are found to be integers.) A new type variable is created for each generic type on every occasion when the identifier bearing that generic type is instantiated. This is done by using a new name, drawn from the given set of type names, Tname, and different from any other name currently in use, for each of the differing identifiers in the generic type. Type checking of an expression involving such type variables is done using type unification which will result in some substitution of types for the set of names. Variable types only have a meaning within an environment giving the substition of types for names, which. is maintained as part of the global state.

4. Predicate

This is a special built-in type, not accessible to the user, which is used to unify the type checking of terms and predicates. For various reasons, both terms and predicates are members of the same syntactic class, so it is helpful to have a special type to distinguish them semantically.

5. Undefined

This is a type for undeclared identifiers, used to suppress type checking and consequential spurious error messages.

A subset of these types are the atomic types, defined by:

AType ≙ rng generic ∪ rng given ∪ {predicate, type_undefined}

```
Z_datatypes keeps
   Name, noname, line, decline,
   Att, noatt, bang, query, dashes,
   Synstatus, ident, op, encop, distinop, distpreop, preop,
   postop, rel, preset, postset, inset,
   Id, LexState, LexState0, LexVal, num, char, string,
   TName, Type, given, powerset, tuple, schema_type, generic,
   variable, predicate, type_undefined, Ztype, Chartype, AType
```

CHAPTER 3

THE IDENTIFIER ENVIRONMENT

Z_datatypes : Module

The identifier environment gives the types associated with each identifier. It is made up of scopes such as those associated with the global document, or the local declarations of a schema, and scopes from imported documents. A scope defines a look-up function, (Id ⇸ Type), for the identifiers which have been declared at the same static level: the rule of declaration before use is followed, so the scope can be changed incrementally as new identifiers are declared. A new scope is created at the beginning of a document, for the local declarations in a schema, a theorem, a comprehension and in many other places. Associated with each scope is a sequence of types used to calculate the characteristic tuple corresponding to a scope. Characteristic tuples are used to calculate the types of λ expressions and other comprehensions. For these constructions the type is regarded as a tuple formed from the declaration list, in which each identifier and inclusion contributes a member in the order in which they were introduced. For the particular case of λ expressions, the characteristic tuple is a somewhat dubious concept if two inclusions have a part of the signature in common. For example, within the context of the schema definitions A ≙ [i , j : ℕ] and B ≙ [j , k : ℕ], λ A; B • j has the type (A × B) → ℕ, which leads to difficulties when the function is applied to a tuple in which the j components differ. The view has been taken that inclusions with overlapping signatures in this way are an error when used to make up a λ expression, so some indication needs to be kept within the current scope that it is destined to form the parameter of a λ expression. This is done using the datatype:

```
scope_type ::= lambda | mu
```

Also associated with each scope is an integer used to keep track of the order of generic identifiers. This is gathered together with the other information to form a Block:

```
┌─ Block ─────────────────────────────────
│ ids : Id ⇸ Type; ctuple : seq Type
│ st : scope_type;  last_generic : ℕ
└─────────────────────────────────────────
```

An imported document also introduces a set of identifiers, but these are not allowed to override previous declarations. This is for reasons of good practice rather than logical consistency, because it is not a good idea to have the same identifier present with two different meanings within the same document. However, an identifier in a document overrides the same identifier in a previously introduced document for reasons of efficiency: it is hard to keep track of all uses, within the current document, of identifiers from external documents and it is unreasonable to expect external documents to have no identifiers in common. This behaviour may be modelled using the following definitions. First of all, the identifier environment itself:

```
┌─ Env ───────────────────────
│ blocks : seq₁ Block
│ docs : seq₁( Id ⇸ Type )
│ docnames : Id ⇸ Id ⇸ Type
├─────────────────────────────
│ rng docnames = rng docs
└─────────────────────────────
```

A particular imported document may be searched using a document name and the function docnames; alternatively all documents may be searched using docs. The

constraint ensures that either method uses consistent look-up functions. There is always one block present in the sequence of blocks, namely the global scope of the current document, and there is always one document, the Z library.

An override function may be defined for sequences of look-up functions as follows:

```
/●_ : seq₁(Id ⇸ Type) → (Id ⇸ Type)
─────────────────────────────────────
∀ l : seq₁(Id ⇸ Type)
• #l = 1 ⇒ /● l = hd l
  #l > 1 ⇒ /● l = (/● (tl l)) ● (hd l)
```

where ● is the relational overrride operator. This function delivers a look-up function in which the identifiers defined in scopes near the beginning of the sequence override those at the end, the implication being that scopes are stacked rather than queued. The function `find env` delivers the type of an identifier stored in a given environment:

```
find : Env → Id ⇸ Type
─────────────────────────────────────
∀ env : Env
•   find env = docids ● blockids
  where
    docids ≙ /● env.docs
    ids ≙ λ Block • ids
    blockids ≙ /● (env.blocks ; ids)
```

and `find_doc` finds from a given document:

```
find_doc ≙ λ env : Env; ident : Id
           | ident ∈ dom env.docnames
           • env.docnames ident
```

Declarations and inclusions change the look-up function and characteristic tuple in the current scope and nothing else, so it is convenient to define the schema:

```
─ ΔEnv ──────────────────────────────
Env; Env'
ΔBlock
─────────────────────────────────────
θBlock = hd blocks ∧ θBlock' = hd blocks'
tl blocks = tl blocks'
docs' = docs ∧ docnames' = docnames
st' = st ∧ last_generic' = last_generic
```

The initial environment consists of one empty block and a set of documents making up the Z library:

```
Z_lib : seq₁(Id ⇸ Type)
Z_lib_names : Id ⇸ Id ⇸ Type
empty_block ≙ μ Block
                | ids = {} ∧ ctuple = ⟨⟩ ∧ st = mu
                ∧ last_generic = 0
                • θBlock
```

```
┌─ Env0 ──────────────────────
│ Env
├─────────────────────────────
│ blocks = ⟨empty_block⟩
│ docs = Z_lib
│ docnames = Z_lib_names
└─────────────────────────────
```

Entering and leaving a scope

On entering a scope a new empty block is added to the environment, on leaving it, the current block is removed:

```
┌─ new_scope ─────────────────────────────
│ Env; Env'
├─────────────────────────────────────────
│ blocks' = empty_block cons blocks
│ docs' = docs
│ docnames' = docnames
└─────────────────────────────────────────
```

```
┌─ end_scope ─────────────────
│ Env; Env'
├─────────────────────────────
│ blocks' = tl blocks
│ docs' = docs
│ docnames' = docnames
└─────────────────────────────
```

and entering a λ expression is a simple variation:

```
┌─ new_lambda_scope ──────────────────────────────────────
│ Env; Env'
├─────────────────────────────────────────────────────────
│   blocks' = lambda_block cons blocks
│ where
│   lambda_block ≙ μ Block
│                   | ids = {} ∧ ctuple = ⟨⟩
│                     ∧ st = lambda ∧ last_generic = 0
│                   • θBlock
│ docs' = docs
│ docnames' = docnames
└─────────────────────────────────────────────────────────
```

Adding new identifiers to an environment

In standard Z it is possible to redeclare an identifier, providing the types are compatible. This seems to be a somewhat dubious facility as it may lead to some user mistakes going undetected, besides allowing for the implicit introduction of additional constraints which ought really to appear explicitly in a predicate. In addition the effect on the characteristic tuple of the environment is questionable. For this reason, a new declaration is not allowed to over-ride an existing declaration within the current block. In the general case, declarations involve a sequence of identifiers each to be given the same type so the declaration operation is:

```
_Declare_________________________________________
ΔEnv
new_ids : seq Id; ty : Type
rep! : seq Char
_________________________________________________
   ids' = ids ∪ good_ids
   ctuple' = ctuple⁀(new_ids ; (λ Id • ty))
   bad_ids ≠ {} ⇒ rep! = "Identifier declared twice"
where
   bad_ids ≙ rng new_ids ∩ dom ids
   good_ids ≙ {ident : rng new_ids
              | ident ∉ dom ids
              • ident ↦ ty}
_________________________________________________
```

Schema merging

In this case added identifiers are allowed to be present in the current scope, provided they have the same type. The following function delivers the inconsistent identifiers:

```
(_ inconsistent _) ≙ λ x, y : Id⇸Type
                       • {ident : Id
                         | ident ∈ dom x ∩ dom y
                         ∧ x ident ≠ y ident
                         }
```

Note that schema merging is done after type normalisation (see chapter 5) which removes all variables from a type, so a simple test for equality of types is all that is required, rather than type unification. This corresponds with the rule that types for identifiers stored within the environment should be fully defined. The new scope is formed by merging the consistent part of the look-up function:

```
_Merge___________________________________________
ΔEnv
merge_ids : Id⇸Type
rep! : seq Char
_________________________________________________
   ids' = good_ids ∪ ids
   bad_ids ≠ {} ⇒ rep! = "Identifiers inconsistent"
where
   bad_ids ≙ merge_ids inconsistent ids
   good_ids ≙ bad_ids ⩤ merge_ids
_________________________________________________
```

The basic operation for a schema inclusion is given by adding a check for overlapping schema signatures and a calculation of the characteristic tuple of the scope. This is always done, even in the global scope, for reasons of simplicity.

```
__Include ______________________________________
| Merge
|_______________________________________________
|   common_ids ≠ {} ∧ bad_ids = {} ∧ st' = lambda ⟹
|      rep! = "Overlapping schemas in λ expression"
|   ctuple' = ctuple snoc schema_type merge_ids
| where
|   bad_ids ≙ merge_ids inconsistent ids
|   common_ids ≙ dom merge_ids \ dom ids
|_______________________________________________
```

Note that the constraint on bad_ids in the schema above is there to give the schema a well-defined meaning and is required because of the simple way in which error reporting is being modelled. In the actual implementation both inconsistent and overlapping identifiers should be reported and in the rest of this specification, in similar situations, rep! will be given multiple values.

Tests on schemas

A useful check for a schema type, used elsewhere in this specification, is

```
__Schema ________________________
| ty? : Type
|________________________________
| ty? ∈ rng powerset
| powerset⁻¹ ty? ∈ rng schema_type
|________________________________
```

For some schema references, each identifier must be present and with the correct type:

```
__Schema_ok _____________________
| ΞEnv
| ty? : Type
|________________________________
|   ty? ∈ rng powerset
|   ty ∈ rng schema_type
|   schema_type⁻¹ ty ⊆ find θEnv
| where
|   ty ≙ powerset⁻¹ ty?
|________________________________
```

```
Z_scopes keeps lambda, mu, Block, Env, find, find_doc, new_scope,
               end_scope, new_lambda_scope, Declare, inconsistent,
               ΔEnv, Merge, Include, Schema, Schema_ok
```

CHAPTER 4

UNIFICATION OF Z TYPES

Z_datatypes :Module

Generic types

Most useful general purpose mathematical functions are generic, that is, they are defined for a range of types. A typical example is the function dom which may be applied to any relation, no matter what its type, to give the domain of application. Z supports this facility by allowing most constructions within the language to be generic, the type parameters being supplied with the definition. To check types during the course of an expression such as dom R, where R is a relation, it is necessary to know the particular type which this instance of dom should have. This can be provided by the user using the named instantiation facilities, but this would be impossibly tedious for functions like dom which are used so extensively. In fact it is possible to infer the required type for a generic term from its use, using an algorithm due to Milner [1978], and this is the approach adopted here.

The algorithm has two parts: in the first part, generic types are instantiated as a type expression in which the generic components have been replaced with variables. This process is specified in the module concerned with anonymous instantiation, (chapter 7). The second part of the type inference process occurs during the various forms of type checking which appear within the compiling functions throughout this specification. These all eventually involve some test for type equality: this may be a simple test if the types are not generic, but if they are, the type inference algorithm enables them to be judged equal if a substitution of types for the variables within the generic types could be found which would make them equal, for this can be the type of instantiation of some generic term. The process of substituting expressions for variables in order to make two terms equal is called unification and a theorem due to Robinson [1965] asserts that an algorithm exists to find the minimum substition for any two terms which will in fact unify, and the specification of this algorithm, particularised for the Z type expressions, is the subject of this module.

Type unification

In this implementation, the variables in a type expression are represented by type names drawn from the set TName and a type environment Tenv, a function from names to types.

$$Tenv \triangleq TName \nrightarrow Type$$

The substitution of a type for a variable is brought about by changing the type environment, which as a result contains the set of substitutions appropriate for the types under consideration. The unification algorithm is represented by a function unify, which takes the current type environment and two types and delivers, if possible, a new type environment in which the two types are equal; otherwise a reply is delivered with the new environment containing the substitutions made before the incompatibility was discovered. The type of the result of unify is given by the schema Uresult:

```
┌─ Uresult ──────────
│ tenv' : Tenv
│ rep!  : seq Char
└────────────────────
```

$$unify : (Tenv \times Type \times Type) \rightarrow Uresult$$

The unification function will be specified incrementally in terms of the various cases

for the structure of the type input values, ending up with a global constraint which defines the function. For this it is useful to gather up the parameters and result of unify into the schema:

```
__Unipars ______________________
| tenv: Tenv; ty1, ty2 : Type
| Uresult
|________________________________
```

First of all, the unification of types which are variable but for which a previous substitution has been made is specified as follows:

```
__Puns __________________________________
| Unipars
|----------------------------------------
|   ty1 ∈ rng variable ∧ n1 ∈ dom tenv ⇒
|      θUresult = unify(tenv, tenv n1, ty2)
| where
|   n1 ≙ variable⁻¹ ty1
| ∧
|   ty2 ∈ rng variable ∧ n2 ∈ dom tenv ⇒
|      θUresult = unify(tenv, ty1, tenv n2)
| where
|   n2 ≙ variable⁻¹ ty2
|________________________________________
```

Immediately after instantiation, a type variable has no type substituted for it, represented by its absence from the domain of the type environment. If a substitution does exist, the name will be present in the type environment and the substituted types are unified. Note that a proof obligation has been incurred for the case where both types are variables, in which case it is necessary to show that the two constraints may be satisfied simultaneously: this will only be the case if substitutions for both type variables are taken into account.

A variable for which no substitution exists may be substituted by any type which does not depend on this variable. This can be checked using the following function which gives the unassigned names in a type.

```
names : (Tenv × Type)→ℙ TName
─────────────────────────────
∀ tenv : Tenv; ty : Type; result : ℙ TName
| result = names(tenv, ty)
•  ty ∈ AType ∧ result = {}
   ∨
   ty ∈ rng variable
     n ∉ dom tenv ∧ result = {n}
     ∨
     n ∈ dom tenv ∧ result = names_in_type(tenv n)
   where
     n ≙ variable⁻¹ ty
   ∨
   ty ∈ rng powerset ∧ result = names_in_type(powerset⁻¹ ty)
   ∨
   ty ∈ rng tuple ∧ result = ⋃ names_in_type⦇rng(tuple⁻¹ ty)⦈
   ∨
   ty ∈ rng schema_type ∧
      result = ⋃ names_in_type⦇rng(schema_type⁻¹ ty)⦈
 where
   names_in_type ≙ λ ty : Type • names(tenv, ty)
```

Either type may be a variable, giving rise to two schemas for substitution. If both types are variables, either may be substituted for the other. If one of the types is dependent on the other, the only allowable case is for both types to be equal, in which case no substitution is required.

```
─ RHsubs ─────────────────────────
 Unipars
──────────────────────────────────
  ty2 ∈ rng variable ∧ n2 ∉ dom tenv
  n2 ∉ names(tenv, ty1) ∧
     tenv' = tenv ∪ {n2 ↦ ty1} ∧ rep! = "OK"
  ∨
  n2 ∈ names(tenv, ty1) ∧ ty1 = ty2 ∧
     tenv' = tenv ∧ rep! = "OK"
  ∨
  n2 ∈ names(tenv, ty1) ∧ ty1 ≠ ty2 ∧
     tenv' = tenv ∧ rep! = "Illegal type"
 where
  n2 ≙ variable⁻¹ ty2
```

```
─ LHsubs ─────────────────────────
 Unipars
──────────────────────────────────
  ty1 ∈ rng variable ∧ n1 ∉ dom tenv
  n1 ∉ names(tenv, ty2) ∧
     tenv' = tenv ∪ {n1 ↦ ty2} ∧ rep! = "OK"
  ∨
  n1 ∈ names(tenv, ty2) ∧ ty1 = ty2 ∧
     tenv' = tenv ∧ rep! = "OK"
  ∨
  n1 ∈ names(tenv, ty2) ∧ ty1 ≠ ty2 ∧
     tenv' = tenv ∧ rep! = "Illegal type"
 where
  n1 ≙ variable⁻¹ ty1
```

The remaining schemas cover the non-variable cases, given by this schema:

```
┌─ Novars ─────────────────────────────────
│ Unipars
├──────────────────────────────────────────
│ ty1 ∉ rng variable ∧ ty2 ∉ rng variable
└──────────────────────────────────────────
```

The undefined type is used for undeclared variables and suppresses some consequential error messages. It is defined to unify with any type.

```
┌─ Undefined ──────────────────────────────
│ Unipars
├──────────────────────────────────────────
│ ty1 = type_undefined ∨ ty2 = type_undefined
│ tenv' = tenv ∧ rep! = "OK"
└──────────────────────────────────────────
```

All other atomic types unify if they are the same:

```
┌─ Uniatoms ───────────────────────────────
│ Unipars
├──────────────────────────────────────────
│ ty1 ∈ AType \ {type_undefined}
│ ty2 ∈ AType \ {type_undefined}
│ tenv' = tenv ∧ ty1 = ty2 ∧ rep! = "OK"
└──────────────────────────────────────────
```

Powersets unify if they are constructed from types which unify:

```
┌─ Unipowers ──────────────────────────────
│ Unipars
├──────────────────────────────────────────
│   ty1 ∈ rng powerset ∧ ty2 ∈ rng powerset
│   θUresult = unify(tenv, tya, tyb)
│ where
│   tya ≙ powerset⁻¹ ty1
│   tyb ≙ powerset⁻¹ ty2
└──────────────────────────────────────────
```

Tuples require the unification of sequences, which is defined to occur between pairs of sequences of the same length and to terminate at the end of the sequence or when corresponding elements of the sequence fail to unify. Each unification takes place within the type environment resulting from previous unifications in the sequence.

```
unifyseq : (Tenv × seq Type × seq Type) ⇸ Uresult
─────────────────────────────────────────────────
∀ tenv : Tenv; tys1, tys2 : seq Type | #tys1 = #tys2 = 1
• unifyseq(tenv, tys1, tys2) = unify(tenv, hd tys1, hd tys2)
∀ tenv : Tenv; tys1, tys2 : seq Type | #tys1 = #tys2 > 1
•  ur1.rep! = "OK" ∧
     unifyseq(tenv, tys1, tys2) =
        unifyseq(ur1.tenv', tl tys1, tl tys2)
   ∨
   ur1.rep! ≠ "OK" ∧
     unifyseq(tenv, tys1, tys2) = ur1
 where
   ur1 ≙ unify(tenv, hd tys1, hd tys2)
```

```
__Unituples ______________________________________
 Unipars
──────────────────────────────────────────────────
   ty1 ∈ rng tuple ∧ ty2 ∈ rng tuple ∧ #tya = #tyb
   θUresult = unifyseq(tenv, tya, tyb)
 where
   tya ≙ tuple⁻¹ ty1
   tyb ≙ tuple⁻¹ ty2
```

The unification of schemas differs from the scheme proposed by Spivey [1985], which it is felt may be confusing to users. In this, the standard scheme, schemas unify if their identifiers are identical and corresponding types unify. This leads to problems because one requires expressions like θSTATE = θSTATE' to type check correctly, so an additional rule is made that decoration does not change the type of a schema. Unfortunately, if the decoration is bound in with the schema, either by providing it explicitly in the signature or within a schema definition such as T ≙ S' the types become different and may not check in situations in which the defining terms would. In this particular example T and S would not have the same type and neither would T and S', which is counter-intuitive. In addition it is not clear what type should be ascribed to s in the declaration s : S' or f in f ≙ λ S; S' • term.

The scheme adopted here uses the alternative discussed by Spivey, in which schema types unify if the identifiers agree modulo any decoration common to all of the identifiers in one schema, and the corresponding types unify. This means that although the underlying type representations differ, the predicate θSTATE = θSTATE' still type checks correctly and the term (θSTATE, θSTATE') will also type check as a member of a homogeneous relation on STATE. In addition, any types which would agree (in the sense of forming a correctly type checked expression) under the standard scheme, will also agree under this one, but some types will agree under this scheme which will not agree under standard one. However, checks on schema merging, and the schema operations generally, are applied to the type representation, so some operations are not allowed under this scheme which would be under the standard. A typical one would be s : STATE; s' : STATE' which does not form a suitable identifier pair for schema composition (whereas s, s' : STATE would). It is, in fact debateable which of the two approaches will be less confusing to the users, but it is in any case a fine distinction and hardly observable to the user, so there does not seem to be a problem with adopting this approach. The advantage of the approach is that the type now contains all the information necessary for checking schema operations and inclusions, which considerably simplifies the implementation.

The implementation must check the types within the schemas one at a time, so the

specification defines an ordering within the identifiers of the schema and uses this to construct the sequence of types to be checked.

```
| order : P Id→seq Id
|------------------------------------
| ∀ ids : P Id; list : seq Id
| | list = order ids
| • rng list = ids ∧ dom list = 1 .. #ids
```

This is not a complete specification as the identifiers are required to be totally ordered such that the addition of a decoration does not alter the order. The specification of this requirement in a way which does not constrain the implementation and does not occupy a page of text is beyond the author's current ability in Z.

Checking identifiers modulo a decoration requires a function to remove either a version or an attribute or both from an identifier as below:

```
decchange ::= discard | keep
undecorate ≙ λ a, v : decchange
             • λ Id
               • μ Id'
                 | name' = name ∧ synstat' = synstat
                   a = keep ∧ att' = att
                      ∨ a = discard ∧ att' = noatt
                   v = keep ∧ version' = version
                      ∨ v = discard ∧ version' = noname
                 • θId'
```

An attribute or version may only be discarded if it is *common* to a set of identifiers, represented by the following two schemas:

```
┌─ common_attribute ──────────────┐   ┌─ common_version ──────┐
│ ids : P Id                      │   │ ids : P Id            │
├─────────────────────────────────┤   ├───────────────────────┤
│ ∃ att : Att                     │   │ ∃ v : Name            │
│ • ∀ ident : ids • ident.att = att│  │ • ∀ ident : ids       │
└─────────────────────────────────┘   │   • ident.version = v │
                                      └───────────────────────┘
```

Note that we have stopped short of distinguishing between differing numbers of dashes. These two schemas may be used to define the function which gives the decoration change required:

```
what_dec ≙ λ ids : P Id
           • μ a, v : decchange
             | common_attribute ∧ a = discard
                  ∨ ¬common_attribute ∧ a = keep
               common_version ∧ v = discard
                  ∨ ¬common_version ∧ v = keep
             • (a, v)
```

With this one can define the unification of schemas as follows;

```
Unischema
  Unipars
  ─────────────────────────────────────────────
    ty1 ∈ rng schema_type ∧ ty2 ∈ rng schema_type
    ids1a = ids2a
    θUresult = unifyseq(tenv, tysa, tysb)
  where
    tys1 ≙ schema_type⁻¹ ty1
    tys2 ≙ schema_type⁻¹ ty2
    dech1 ≙ what_dec dom tys1
    dech2 ≙ what_dec dom tys2
    ids1 ≙ order dom tys1
    ids2 ≙ order dom tys2
    ids1a ≙ ids1 ; undecorate dech1
    ids2a ≙ ids2 ; undecorate dech2
    tysa ≙ ids1 ; tys1
    tysb ≙ ids2 ; tys2
```

Thus the specification for the unification of non-variable types is:

```
Non_vars ≙ Uniatoms ∨ Undefined ∨ Unipowers ∨ Unituples
           ∨ Unischema
```

with error case:

```
Typewrong
  Unipars
  ─────────────────────────────────────────────
  ¬Non_vars ∧ rep! = "Incompatible type" ∧ tenv' = tenv
```

The various cases may be collected together into one schema

```
UNIFY ≙ Puns ∧ ((RHsubs ∨ LHsubs)
               ∨ (Novars ∧ (Non_vars ∨ Typewrong)))
```

to give a definition of the unify function as:

```
∀ Unipars • θUresult = unify(tenv, ty1, ty2) ⇔ UNIFY
```

Type checking operations

Type checking takes place within the type environment which gives the current assignment of types to names. For generic type instantiation it is necessary to create names unique to the current type environment, so the state for type checking operations must maintain the set of valid names.

```
┌─ TypeState ─────────────────
│ tenv : Tenv
│ valid_names : ℙ TName
├─────────────────────────────
│ dom tenv ⊆ valid_names
└─────────────────────────────
```

Finally, a general purpose operation to check if two types are the same:

```
┌─ TypeCheck ─────────────────
│ ΔTypeState;  UNIFY
├─────────────────────────────
│ valid_names' = valid_names
└─────────────────────────────
```

Z_type_unify keeps Tenv, TypeState, TypeCheck

CHAPTER 5

NORMALISATION OF TYPES

Z_datatypes :Module

Z_type_unify :Module

As a result of type checking operations the type produced for an expression may contain a number of variables, all of which should, at various points in the syntax such as declarations, have a substitution present within the type environment. Normalisation is the name used for the process of transforming a type by carrying out the substitutions implicit within the environment and should result in a type containing no variable elements, and in a standard form. (Note that this use of the term is different from the normal Z usage which refers to the general process of deriving a type from a term.) Only types which have been normalised may be directly compared: in all other cases types should be unified using the TypeCheck operation. Normalisation is carried out using the following function:

```
normalise : (Type × (TName ⇸ Type)) → Type
```

This is defined according to the subsets of type as follows:

```
┌─ NPars ──────────────
│ ty : Type
│ tenv : TName ⇸ Type
│ result : Type
└──────────────────────
```

```
┌─ Nvar ───────────────────────────────────────
│ NPars
├──────────────────────────────────────────────
│ ty ∈ rng variable
│   n ∈ dom tenv ∧ result = normalise(tenv n, tenv)
│   ∨
│   n ∉ dom tenv ∧ result = ty
│ where
│   n ≙ variable⁻¹ ty
└──────────────────────────────────────────────
```

```
┌─ NPowers ────────────────────────────────────
│ NPars
├──────────────────────────────────────────────
│ ty ∈ rng powerset
│ result = powerset(normalise(powerset⁻¹ ty, tenv))
└──────────────────────────────────────────────
```

```
┌─ Ntuples ────────────────────────────────
│ NPars
├──────────────────────────────────────────
│ ty ∈ rng tuple
│   result = tuple(tuple⁻¹ ty ; norm)
│ where
│   norm ≙ λ ty : Type • normalise(ty, tenv)
└──────────────────────────────────────────
```

```
Nschema
NPars
────────────────────
ty ∈ rng schema_type
  result = schema_type(schema_type⁻¹ ty ; norm)
where
  norm ≙ λ ty : Type • normalise(ty, tenv)
```

```
NAtom
NPars
────────────────────
ty ∈ AType ∧ result = ty
```

NORM ≙ Nvar ∨ NPowers ∨ Ntuples ∨ Nschema ∨ NAtom

∀ NPars • result = normalise(ty, tenv) ⇔ NORM

After normalisation the type should contain no type variables, so define a function to count them:

```
names_in_type : Type → P TName
────────────────────
∀ ty : AType • names_in_type ty = {}
∀ ty : rng variable • names_in_type ty = {variable⁻¹ ty}
∀ ty : rng powerset
• names_in_type ty = names_in_type(powerset⁻¹ ty)
∀ ty : rng tuple
• names_in_type ty = ⋃ names_in_type⦇rng(tuple⁻¹ ty)⦈
∀ ty : rng schema_type
• names_in_type ty = ⋃ names_in_type⦇rng(schema_type⁻¹ ty)⦈
```

The normalisation operation must be applied to all user-defined types:

```
Normalise
ty?, ty! : Type
ΞTypeState
rep! : seq Char
────────────────────
ty! = normalise(ty?, tenv)
#(names_in_type ty!) ≠ 0
   ⇒ rep! = "Type not completely specified"
```

Z_type_norm keeps Normalise

CHAPTER 6

REFERENCES TO IDENTIFIERS

Z_datatypes :Module

Z_scopes :Module

Ordinary references

The syntax for references is:

```
reference =
    id <reference--type> ref2,
    id dlr <check_no_att> id <doc_reference--type> ref2;
```

A reference delivers a type which is the value of an identifier in the current environment (instantiation is dealt with later). The identifier is obtained from the lexical analyser's state variables and looked up in the current environment to find its type.

```
TopId
  ΔLexState
  id! : Id
  ─────────
  id! = hd idlist
  idlist' = tl idlist
```

```
ref_ok
  ΞEnv
  id? : Id; ty! : Type
  ─────────
  id? ∈ dom(find θEnv)
  ty! = find θEnv id?
```

If ref_ok cannot be satisfied, the identifier is undeclared. This may not be an error as there are various identifiers which are conventionally formed from existing identifers, namely decorated schemas and schemas used with Δ and Ξ. The first use of these identifiers when they have not been defined, and a schema of the appropriate base name has, will result in the declaration of the appropriate schema term. First of all, to express this, it is necessary to define a function to carry out the decoration and which expresses the rule that !, ? or a version may only be applied once.

```
decorate_with _ : (Name × Att)→(Id⇸Id)
─────────────────────────────────────
∀ new_version : Name; new_att : Att
• decorate_with(new_version, new_att) =
    λ Id
    | new_version ≠ noname ⇒ version = noname
      new_att ∈ rng dashes ⇒ att = noatt ∨ att ∈ rng dashes
      new_att = bang ∨ new_att = query ⇒ att = noatt
    • μ Id'
      | name' = name
        new_version = noname ⇒ version' = version
        new_version ≠ noname ⇒ version' = new_version
        att = noatt ⇒ att' = new_att
        new_att = noatt ⇒ att' = att
        new_att ≠ noatt ∧ att ≠ noatt ⇒
          att' = dashes(dashes⁻¹ att + dashes⁻¹ new_att)
        synstat' = synstat
      • θId'
```

Identifiers beginning with Δ or Ξ which have not been declared, but for which a schema definition for the identifier formed from the name without the initial Greek letter exists will have a new schema definition created automatically. The new schema involves decoration with a dash, and the schema must be capable of being decorated in this way:

```
_derived_id ____________________________________________
| ΞEnv
| id?, id! : Id; ty! : Type
|________________________________________________________
|
|   first_char = 'Δ' ∨ first_char = 'Ξ'
|   id' ∈ dom(find θEnv)
|   ty' ∈ rng powerset ∧ powerset⁻¹ ty' ∈ rng schema_type
|   dom ids ⊆ dom decorate
|   ty! = powerset(schema_type ids')
|   id! = id?
| where
|   first_char ≙ hd(line⁻¹ id?.name).l
|   id' ≙ μ Id
|         |    name = line(θdecline')
|           where
|             | Δdecline
|             |______________________
|             |
|             | θdecline = line⁻¹ id?.name
|             | l' = tl l ∧ w' = w
|           version = id?.version
|           att = id?.att ∧ synstat = id?.synstat
|         • θId
|   ty' ≙ find θEnv id'
|   ids ≙ schema_type⁻¹(powerset⁻¹ ty')
|   decorate ≙ decorate_with(noname, dashes 1)
|   ids' ≙ {id' : Id; ty : Type
|          | ∃ ident : dom ids
|            | id' = decorate ident • ty = ids ident
|          • id' ↦ ty
|          }
|________________________________________________________
```

The same possibility for implicit declaration exists for an undeclared identifier with a decoration, if a schema definition exists for the undecorated identifier. Note that for reasons of simplicity the view has been taken that the base name version must have been defined, which precludes the decoration of an imported schema if the defining document has been renamed, because this simply imports the decorated names. As with the derived schema, the decorated schema must be capable of being decorated in the way required.

```
┌─ decorated_id ──────────────────────────────────────────
│ ΞEnv
│ id?, id! : Id; ty! : Type
├─────────────────────────────────────────────────────────
│   ident ∈ dom(find θEnv)
│   ty ∈ rng powerset ∧ powerset^-1 ty ∈ rng schema_type
│   dom ids ⊆ dom decorate
│   ty! = powerset(schema_type ids')
│   id! = id?
│ where
│   ident ≙ μ Id
│             | name = id?.name
│               version = noname
│               att = noatt ∧ synstat = id?.synstat
│             • θId
│   ty ≙ find θEnv ident
│   ids ≙ schema_type^-1(powerset^-1 ty)
│   decorate ≙ decorate_with(id?.version, id?.att)
│   ids' ≙ {id' : Id; ty : Type
│           | ∃ ident : dom ids
│             | id' = decorate ident • ty = ids ident
│           • id' ↦ ty
│           }
└─────────────────────────────────────────────────────────
```

All error cases are treated as an undeclared identifier, which is a bit unfriendly in the case of incompatible decorations:

```
┌─ ref_wrong ─────────────────────
│ ΞEnv
│ id?, id! : Id; ty! : Type
│ rep! : seq Char
├─────────────────────────────────
│ ¬(decorated_id ∨ derived_id)
│ rep! = "Identifier undeclared"
│ ty! = type_undefined
│ id! = id?
└─────────────────────────────────
```

For the three undeclared cases, it is necessary to ensure that the identifier created is declared, so that a subsequent declaration will be ignored and not give rise to inconsistency.

```
┌─ decid ─────────────────────────
│ Declare[ty?/ty]
│ id? : Id;  ty! : Type
├─────────────────────────────────
│ new_ids = ⟨id?⟩ ∧ ty! = ty?
└─────────────────────────────────

undeclared ≙ (derived_id ∨ decorated_id ∨ ref_wrong) ≫
                  decid\(rep!, new_ids)
reference ≙ TopId ≫ (undeclared • ref_ok)
```

A decoration may also be applied to a schema term in the following situation, where spec_sexp2 is a syntax rule occuring in the expansion of the rules for special purpose schema expressions.

```
rename =
   lsqb rename_list rsqb <id_inst-ptype-pinstantiation--type>,
   decor <decorate-ptype--type>;

spec_sexp2 =
   lpar schema_term rpar,
   lpar schema_term rpar rename,
   reference <check_schema-ptype--type>,
   schema;
```

The compiling functions is a simple variation of the above: the decoration required is found in an identifier left at the head of the lexical analyser's queue.

```
__decorate_ok ________________________________
 id? : Id
 ty?, ty! : Type
 rep! : seq Char
______________________________________________
   ty! = powerset(schema_type ids')
   dom ids ⊆ dom decorate
 where
   ids ≙ schema_type⁻¹(powerset⁻¹ ty?)
   decorate ≙ decorate_with(id?.version, id?.att)
   ids' ≙ {id' : Id; ty : Type
          | ∃ ident : dom ids
            | id' = decorate ident • ty = ids ident
          • id' ↦ ty
          }
______________________________________________
```

The only error case occurs with incorrect decorations as the syntax ensures that the type of a schema term is always a schema type.

```
__decorate_wrong _____________________________
 id? : Id
 ty?, ty! : Type
 rep! : seq Char
______________________________________________
   ty! = ty?
   ¬(dom ids ⊆ dom decorate) ⇒ rep! = "Incorrect decoration"
 where
   ids ≙ schema_type⁻¹(powerset⁻¹ ty?)
   decorate ≙ decorate_with(id?.version, id?.att)
______________________________________________
```

```
decorate ≙ TopId ≫ decorate_ok ∨ decorate_wrong
```

Document references

Document references are preceded by a document name, which must contain no attributes. If they are present an error is reported and they are discarded.

```
┌─ check_no_att ──────────────────────────────────────────
│ ΔLexState
│ rep! : seq Char
├─────────────────────────────────────────────────────────
│ (hd idlist).att = noatt ⇒ idlist' = idlist
│ (hd idlist).att ≠ noatt ⇒
│   rep! = "Document reference may not contain attributes"
│   hd idlist' = μ Id
│                | name = ident.name
│                  version = ident.version
│                  att = noatt
│                  synstat = ident.synstat
│                 where
│                  ident ≙ hd idlist
│                • θId
│   tl idlist' = tl idlist
└─────────────────────────────────────────────────────────
```

```
┌─ TopIds ──────────────────────────────
│ ΔLexState
│ id!, doc! : Id
├───────────────────────────────────────
│ id! = idlist(2) ∧ doc! = idlist(1)
│ idlist' = tl(tl idlist)
└───────────────────────────────────────
```

```
┌─ doc_ref_ok ──────────────────────────
│ ΞEnv
│ id?, doc? : Id
│ ty! : Type
├───────────────────────────────────────
│ (θEnv, doc?) ∈ dom find_doc
│ id? ∈ dom(find_doc(θEnv, doc?))
│ ty! = find_doc(θEnv, doc?) id?
└───────────────────────────────────────
```

If the identifier is not present in the document an error is reported without attempting to look for decorated versions: this is a somewhat debateable decision.

```
┌─ doc_ref_wrong ─────────────────────────────────────────
│ ΞEnv
│ id?, doc?, id! : Id
│ ty! : Type
│ rep! : seq Char
├─────────────────────────────────────────────────────────
│ (θEnv, doc?) ∉ dom find_doc ∧ rep! = "No such document"
│ ∨
│ (θEnv, doc?) ∈ dom find_doc
│ ∧ id? ∉ dom(find_doc(θEnv, doc?))
│ ∧ rep! = "Identifier undeclared"
│ ty! = type_undefined
└─────────────────────────────────────────────────────────
```

```
doc_reference ≙ TopIds ≫ (doc_ref_ok ∨ doc_ref_wrong) ≫
                                          decid\(rep!, new_ids)

Z_references keeps reference, decorate, check_no_att,
                   doc_reference
```

CHAPTER 7

ANONYMOUS INSTANTIATION

Z_datatypes :Module

Z_scopes :Module

Z_type_unify :Module

The relevant extract from the syntax is:

```
reference =
   id <reference--type> ref2,
   id dlr <check_no_att> id <doc_reference--type> ref2;

   ref2 =
      <anon_inst-ptype--type>,
      instantiation <id_inst-ptype-pinstantiation--type>;
```

For anonymous instantiation, the input type ty? will be the type of the identifier as given by the identifier environment: the output type ty! is the instantiated type, which, if the type is generic, must be suitable for the application of the type inference rules. Consequently it is necessary to find the generic identifiers within the type. This is slightly complicated by the fact that at a defining occurence of an identifier, *for example the predicate part of an axiomatic definition*, types dependent on the generic parameters should not be instantiated at differing types. If this rule is not followed, it is possible to create some inconsistencies. For example, defining a generic function f : S→T, where S and T are generic, should result in an error if the predicate contains f = λ s : S • s, as the delivered type must be the same as the parameter. Within a generic definition, the identifiers for the generic parameters are still in scope, so this gives a test as to which generic types should be instantiated as variables and which should not: the true generics are those which appear in the type, but are not currently defined within the environment. The effect of this rule is that generic schemas used as an inclusion within a generic schema definition which is generic in the same identifiers will, if instantiated anonymously, be treated as the same generic parameters. The effect is as if the new generic definition extends the old one, which is probably what is intended.

Note that variable types are only created for the purposes of anonymous instantiation, so that the type derived from the identifier environment and supplied by reference *will contain no variable component*. This is checked using the type normalisation function (see chapter 5) which is always used prior to declaring an identifier with a given type. First of all then, a function to give the true generics within a type:

```
ids_in_type : (Env × Type) ⇸ ℙ Id
──────────────────────────────────────
∀ env : Env
•   ∀ ty : rng generic
    •   ident ∈ generic_ids ∧ ids ty = {}
        ∨ ident ∉ generic_ids ∧ ids ty = {ident}
      where
        | ident : Id
        |──────────────────────
        | ∃ n : ℕ • ty = generic(n, ident)
    ∀ ty : AType \ rng generic • ids ty = {}
    ∀ ty : rng powerset • ids ty = ids(powerset⁻¹ ty)
    ∀ ty : rng tuple • ids ty = ⋃ ids⦇rng(tuple⁻¹ ty)⦈
    ∀ ty : rng schema_type
    • ids ty =  ⋃ ids⦇rng(schema_type⁻¹ ty)⦈
  where
    ids ≙ λ ty : Type • ids_in_type(env, ty)
    generic_ids ≙ {ident : Id
                   | ∃ n : ℕ
                     • find env ident =
                           powerset(generic(n, ident))
                   }
```

The instantiation of a non-generic variable is straightforward:

```
┌─ Non_gen_var ──────────────
│ ΞTypeState
│ ΞEnv
│ ty?, ty! : Type
├────────────────────────────
│ ids_in_type(θEnv, ty?) = {}
│ ty! = ty?
└────────────────────────────
```

For generics we need a mapping from the generic identifiers to a unique set of names, not currently existing within the environment:

```
┌─ Newnames ─────────────────
│ ΔTypeState
│ ΞEnv
│ ty? : Type
│ sub! : Id ⤖ TName
├────────────────────────────
│ tenv = tenv'
│ dom sub! = ids_in_type(θEnv, ty?)
│ rng sub! ∩ valid_names = {}
│ valid_names' = valid_names ∪ rng sub!
└────────────────────────────
```

Given a mapping from identifiers to names, the following function produces a variable type from a generic:

```
inst_type : (Type × (Id ⇸ TName)) ⇸ Type
──────────────────────────────────────────
∀ s : Id ⇸ TName
•∀ ty : rng generic
 •   ident ∈ dom s ∧ inst_type(ty, s) = variable(s ident)
       ∨ ident ∉ dom s ∧ inst_type(ty, s) = ty
   where
     | ident : Id
     |──────────────────────
     | ∃ n : ℕ • ty = generic(n, ident)
 ∀ ty : AType \ rng generic • inst_type(ty, s) = ty
 ∀ ty : rng powerset
 • inst_type(ty, s) = powerset(inst_type(powerset⁻¹ ty, s))
 ∀ ty : rng schema_type
 •   inst_type(ty, s) = schema_type(schema_type⁻¹ ty ; inst)
  where
     inst ≙ λ ty : Type • inst_type(ty, s)
 ∀ ty : rng tuple
 •   inst_type(ty, s) = tuple(tuple⁻¹ ty ; inst)
  where
     inst ≙ λ ty : Type • inst_type(ty, s)
```

which may be used to create the new type from the one given by *reference*:

```
┌─ Gen_var ──────────────
│ ΞTypeState
│ sub? : Id ⇸ TName
│ ty?, ty! :Type
├────────────────────────
│ ty! = inst_type(ty?, sub?)
└────────────────────────
```

The total operation for anonymous instantiation is

```
anon_inst ≙ Non_gen_var ∨ (Newnames ≫ Gen_var)
Z_anon_inst keeps anon_inst, ids_in_type
```

CHAPTER 8

FORMATION OF INSTANTIATIONS

Z_datatypes :Module

Z_scopes :Module

Z_references :Module

Z_type_norm :Module

Named instantiation is applied to identifiers having a generic type and brings about the replacement of the generic components in the type of the identifier with the types given for this particular use of the identifier. The given types may be introduced in the form of a list of types, in which case the generic components are given by the order in which they were introduced in the generic definition which assigned a type to the identifier, or they may be introduced as a mapping between identifiers and types, to indicate which particular generic component is to be instantiated. Because of the syntactic difficulties of distinguishing between instantiation and schema renaming (both begin with an opening square bracket and can carry on with an identifier), both are treated as belonging to the same syntactic class, so the representation of an instantiation must cover both possibilities. This leads to the following datatype definition for the representation of an instantiation:

```
Instantiation ::= term_list«seq Type» | binding_list«Id ⇸ Type»
                | rename_list«Id ⇸ Id»
```

The relevant syntax for the formation of a term list instantiation is:

```
instantiation = lsqb inst_list rsqb;

inst_list =  inst_term_list, binding_list, rename_list;
#gathered together to resolve various one-track problems#

inst_term_list =
   term <tml1-ptype--instantiation>,
   term <tml1-ptype--instantiation> comma inst_termlist1;

   inst_termlist1 =
      term <tml2-ptype-pinstantiation--instantiation> inst_termlist2;

      inst_termlist2 = $, comma inst_termlist1;
```

The compiling functions tml1 and tml2 form the instantiation from the component terms, each of which must stand for a type. This is because the generic identifiers have powerset type so that they may be used in a signature and consequently they may only be instantiated as sets. This is checked in the operation below, which removes the powerset constructor to form the type which will be stored in the instantiation.

```
Check_typen
ty?, ty! : Type;  rep! : seq Char
─────────────────────────────────────
ty? ∈ rng powerset ∧ ty! = powerset⁻¹ ty?
∨
ty? ∉ rng powerset ∧ ty! = ty?
   ∧ rep! = "Incorrect type for instantiation"
```

As well as this check, the view has been taken that instantiations ought to be defined, as otherwise it would be possible to instantiate a generic term with the empty set for example. Consequently an additional check is made to ensure that the type is normalised.

```
Check_type ≙ Normalise ⨠ Check_typen
```

The operation tml1 forms the first element in the term list instantiation:

```
┌─ tml1a ──────────────────────────────
│ ty? : Type;   inst! : Instantiation
├──────────────────────────────────────
│ inst! = term_list ⟨ty?⟩
└──────────────────────────────────────
```

```
tml1 ≙ Check_type ⨠ tml1a
```

while tml2 is for subsequent elements

```
┌─ tml2a ──────────────────────────────────────────
│ ty? : Type;   inst?, inst! : Instantiation
├──────────────────────────────────────────────────
│ inst! = term_list(term_list⁻¹ inst? snoc ty?)
└──────────────────────────────────────────────────
```

```
tml2 ≙ Check_type ⨠ tml2a
```

There are similar operations for binding lists and rename lists:

```
binding_list = bl1, bl1 comma binding_list1;

  bl1 =
     id equals <bl1--id> term <bl2-pid-ptype--instantiation>;

  binding_list1 =
     bl2, bl2 comma binding_list1;

     bl2 =
       id equals <bl1--id> term
       <bl3-pid-ptype-pinstantiation--instantiation>;
```

```
┌─ bl1 ─────────────────
│ ΔLexState
│ id! : Id
├───────────────────────
│ id! = hd idlist
│ idlist' = tl idlist
└───────────────────────
```

```
┌─ bl2a ────────────────────────────────
│ id? : Id; ty? : Type
│ inst! : Instantiation
├───────────────────────────────────────
│ inst! = binding_list {id? ↦ ty?}
└───────────────────────────────────────
```

```
bl2 ≙ Check_type ⨠ bl2a
```

Note that a compiling function bl1 is required because the term may alter the identifier queue. For subsequent terms in the binding list, a binding mentioning the same identifier twice is ignored and an error reported.

```
_b13a ___________________________________________________
| id? : Id; ty? : Type;  inst?, inst! : Instantiation
| rep! : seq Char
|________________________________________________________
|   id? ∉ dom tys ∧ inst! = binding_list(tys ∪ {id? ↦ ty?})
|   ∨
|   id? ∈ dom tys ∧
|      rep! = "Identifier occurs twice" ∧ inst! = inst?
| where
|   tys ≙ binding_list⁻¹ inst?
|________________________________________________________
```

```
b13 ≙ Check_type ⨠ b13a
```

Compiling a rename list is easier because only identifiers are involved.

```
rename_list =
   id for id <rnl1--instantiation>,
   id for id <rnl1--instantiation> rename_list1;

   rename_list1 =
      id for id <rnl2-pinstantiation--instantiation> rename_list2;

      rename_list2 = $, comma rename_list1;
```

```
_rnl1 ____________________________________
| ΔLexState
| inst! : Instantiation
|_________________________________________
| inst! = rename_list {idlist(1) ↦ idlist(2)}
| idlist' = tl(tl idlist)
|_________________________________________
```

Various obvious error cases are dealt with in the compiling function for subsequent elements in the rename list.

```
_rnl2 ___________________________________________________
| ΔLexState
| inst?, inst! : Instantiation
| rep! : seq Char
|________________________________________________________
|   idlist(1) ∉ dom idmap ∧ idlist(2) ∉ rng idmap ∧
|      inst! = rename_list(idmap ∪ {idlist(1) ↦ idlist(2)})
|   ∨
|   idlist(1) ∈ dom idmap ∧
|      rep! = "Identifier occurs twice" ∧ inst! = inst?
|   ∨
|   idlist(2) ∈ rng idmap ∧
|      rep! = "Coincidental renaming" ∧ inst! = inst?
| where
|   idmap ≙ rename_list⁻¹ inst?
| idlist' = tl(tl idlist)
|________________________________________________________
```

```
Z_mk_instantiation keeps Instantiation, term_list, binding_list,
                         rename_list, Check_type, tml1, tml2,
                         bl1, bl2, bl3, rnl1, rnl2
```

CHAPTER 9

NAMED INSTANTIATION

```
Z_datatypes :Module
Z_scopes :Module
Z_mk_instantiation :Module
Z_anon_inst :Module
```

The relevant syntax is that associated with references:

```
reference =
   id <reference--type> ref2,
   id dlr <check_no_att> id <doc_reference--type> ref2;

  ref2 =
     <anon_inst-ptype--type>,
     instantiation <id_inst-ptype-pinstantiation--type>;
```

There are three similar cases for named instantiation, depending on the type of the instantiation. Note that as a result of combining schema renaming with instantiation it is possible to rename a schema within an inclusion, which turns out to be a useful facility, so it has not been disallowed.

Two functions will be defined which carry out term list and binding list instantiation. They are very similar and may be defined according to the elements of type. First of all, a function for term list instantiation:

```
tl_inst : (Type × seq Type)→Type
─────────────────────────────────────────
∀ ty, result : Type; s : seq Type
| result = tl_inst(ty, s)
• ∃ ident : Id; n : ℕ | ty = generic(n, ident)
   • n ∈ dom s ⟹ result = s n
     n ∉ dom s ⟹ result = ty
  ∨
  ty ∈ AType \ rng generic ∧ result = ty
  ∨
  ty ∈ rng powerset
    ∧ result = powerset(tl_inst(powerset⁻¹ ty, s))
  ∨
    ty ∈ rng tuple
    result = tuple(tys ⨾ inst)
  where
    tys ≙ tuple⁻¹ ty
    inst ≙ λ ty : Type • tl_inst(ty, s)
  ∨
    ty ∈ rng schema_type
    result = schema_type(tys ⨾ inst)
  where
    tys ≙ schema_type⁻¹ ty
    inst ≙ λ ty : Type • tl_inst(ty, s)
```

Binding list instantiation:

```
bl_inst : (Type × (Id ⇸ Type)) ⇸ Type
────────────────────────────────────
∀ ty, result : Type; s : Id ⇸ Type
| result = bl_inst(ty, s)
• ∃ ident : Id; n : ℕ | ty = generic(n, ident)
  • ident ∈ dom s ⇒ result = s ident
    ident ∉ dom s ⇒ result = ty
  ∨
  ty ∈ AType \ rng generic ∧ result = ty
  ∨
  ty ∈ rng powerset
    ∧ result = powerset(bl_inst(powerset⁻¹ ty, s))
  ∨
    ty ∈ rng tuple
    result = tuple(tys ; inst)
  where
    tys ≙ tuple⁻¹ ty
    inst ≙ λ ty : Type • bl_inst(ty, s)
  ∨
    ty ∈ rng schema_type
    result = schema_type(tys ; inst)
  where
    tys ≙ schema_type⁻¹ ty
    inst ≙ λ ty : Type • bl_inst(ty, s)
```

The function for schema renaming is an extended composition:

```
schema_rename_ : ((Id ⇸ Type) × (Id ⇸ Id)) ⇸ (Id ⇸ Type)
────────────────────────────────────
∀ schema_ids : Id ⇸ Type;  idmap : Id ⇸ Id
•    schema_rename(schema_ids, idmap) = idmap' ; schema_ids
  where
    idmap' ≙ λ ident : Id
              • μ ident' : Id
                | ident ∈ dom idmap ⇒ ident' = idmap ident
                  ident ∉ dom idmap ⇒ ident' = ident
                • ident'
```

The operation for instantiation checks error cases, but instantiates as many types as are correct.

```
_ inst ______________________________________
 ΞEnv
 ty?, ty! : Type
 inst? : Instantiation
 rep! : seq Char
_____________________________________________
   inst? ∈ rng term_list
   ty! = tl_inst(ty?, s)
   #dom s > #(ids_in_type(θEnv, ty?)) ⇒
      rep! = "Too many terms"
 where
   s ≙ term_list⁻¹ inst?
 ∨
   inst? ∈ rng binding_list
   ty! = bl_inst(ty?, s)
   ¬(dom s ⊆ ids_in_type(θEnv, ty?)) ⇒
      rep! = "Not generic in this identifier"
 where
   s ≙ binding_list⁻¹ inst?
_____________________________________________
```

while that for schema renaming is

```
_ rename ____________________________________________________
 ΞEnv
 ty?, ty! : Type
 inst? : Instantiation
 rep! : seq Char
_____________________________________________________________
 inst? ∈ rng rename_list
 ¬Schema ∧ ty! = ty? ∧ rep! = "Only schemas may be renamed"
 ∨
   bad_ids ≠ {} ⇒
      rep! = "Identifiers not defined in this schema"
   ty! = powerset(schema_type schema_rename(ids, good_ids))
 where
   ids ≙ schema_type⁻¹(powerset⁻¹ ty?)
   idmap ≙ rename_list⁻¹ inst?
   bad_ids ≙ rng idmap \ dom ids
   good_ids ≙ idmap ▷ dom ids
_____________________________________________________________
```

After named instantiation or schema renaming, any generic types remaining are instantiated anonymously:

```
id_inst ≙ (inst ∨ rename) ≫ anon_inst
Z_named_inst keeps id_inst, inst
```

CHAPTER 10

GIVEN SET DEFINITIONS AND GENERIC PARAMETERS

Z_datatypes :Module

Z_scopes :Module

Given set definitions

The syntax for given set definitions is very simple:

```
given_set_def = lsqb given_ids rsqb;
   given_ids = id <given_set_def>, given_ids1;
      given_ids1 = $, comma given_ids;
```

The rule given_ids is also used for the parameters in generic definitions. Given set identifiers and generic parameters must be unique within the current scope and are not allowed to have attributes or versions. This is partly because of the form chosen for the syntactic status of generic sets, but it does seem to be a reasonable restriction.

```
_check_given_id ________________________________
 ΔLexState
 rep! : seq Char
 ident! : Id
________________________________________________
   ident.att ≠ noatt ∨ ident.version ≠ noname ⇒
      rep! = "Parameter identifiers may not be decorated"
   ident! = μ Id
            | name = ident.name ∧ version = noname
              ∧ att = noatt ∧ synstat = ident.synstat
            • θId
 where
   ident ≙ hd idlist
 idlist' = tl idlist
________________________________________________
```

```
_given_set_error ______________________________
 ΞEnv
 ident? : Id
 rep! : seq Char
_______________________________________________
 ident? ∈ dom (hd blocks).ids
 rep! = "Identifier for given set already declared"
_______________________________________________
```

The type of T in [T] is just powerset of given of T in the outermost block, or generic of T in an inner block, combined with the serial number to allow for sequential instantiation. The global scope is detected by the fact that there is only one block in the environment.

```
_given_set_ok ___________________________________________
| Env; Env'
| ΔBlock
| ident? : Id
|---------------------------------------------------------
| θBlock = hd blocks ∧ θBlock' = hd blocks'
| tl blocks = tl blocks'
| docs' = docs ∧ docnames' = docnames
| st' = st
| last_generic' = last_generic + 1
| ident? ∉ dom ids
|   ids' = {ident? ↦ ty} ∪ ids
| where
|   | ty : Type
|   |-----------------------------------------------------
|   | #blocks = 1 ⇒ ty = powerset(given ident?)
|   | #blocks > 1 ⇒
|   |    ty = powerset(generic(last_generic', ident?))
|_________________________________________________________
```

In this schema, it has been necessary to expand declare, because of the change to generics.

```
given_set_def ≙ check_given_id >>
                (given_set_error ∨ given_set_ok)
```

Generic definitions

For generic parameters an extra scope is introduced to contain the identifiers and their types:

```
gen_params = glsqb <new_scope> given_ids rsqb;
```

A further scope is created to contain the newly declared identifiers, after which the usual declaration and definition functions (see later) are used:

```
generic_def =
   global_id <start_idlist--idlist> gen_params <new_scope>
   colon term <dec_ids-pidlist-ptype>
   cbar pred <unstack_pred-ptype><end_gen_def>,
   sr gen_params <new_scope> ge def_body er <end_gen_def>;
```

At the end of the generic definition the current scope is merged with the outer one and the generic parameter scope thrown away:

```
┌─ scope_to_ids ──────────────────
│ Env;  Env'
│ merge_ids : Id ⇸ Type
│ m_tuple : seq Type
├─────────────────────────────────
│ merge_ids = (hd blocks).ids
│ m_tuple = (hd blocks).ctuple
│ blocks' = tl(tl blocks)
│ docs' = docs
│ docnames' = docnames
└─────────────────────────────────
```

```
┌─ Merge_ids ─────────────────────
│ Merge
│ m_tuple : seq Type
├─────────────────────────────────
│ ctuple' = ctuple⁀m_tuple
└─────────────────────────────────
```

The characteristic tuple will not be used at the global level, but the declarations are appended here for consistency.

```
end_gen_def ≙ (scope_to_ids ⨟ Merge_ids)\(merge_ids, m_tuple)
Z_given_sets keeps given_set_def, end_gen_def
```

CHAPTER 11

DECLARATIONS AND INCLUSIONS

```
Z_datatypes :Module
Z_scopes :Module
Z_type_norm :Module
```

Declarations

The commonest form of definition is the declaration, which appears in the form of a declaration list, repeatedly throughout the syntax. The syntax for a declaration is as follows:

```
dec = id_list colon term <dec_ids-pidlist-ptype>;

id_list =
   id <start_idlist--idlist>,
   id <start_idlist--idlist> comma idlist1;

   idlist1 =
      id <stack_idlist-pidlist--idlist>,
      id <stack_idlist-pidlist--idlist> comma idlist1;
```

The declaration gives a list of identifiers and a term to define their type. As term may alter the lexical state, it is necessary to stack the identifier list, rather than using the lexical analyser's queue of identifiers:

```
start_idlist
 ΔLexState
 idlist! : seq Id
 ─────────────
 idlist! = ⟨hd idlist⟩
 idlist' = tl idlist
```

The stacking function checks for repeated identifiers:

```
stack_idlist
 ΔLexState
 idlist?, idlist! : seq Id
 rep! : seq Char
 ─────────────
    ident ∈ rng idlist? ⇒
      idlist! = idlist? ∧ rep! = "Identifier declared twice"
    ident ∉ rng idlist? ⇒
      idlist! = idlist? snoc ident
    idlist' = tl idlist
 where
    ident ≙ hd idlist
```

The identifiers may be declared if the term in the declaration specifies a type and the type is compatible with the syntactic status of the identifier. In the standard syntax the syntactic status may only be set for a global identifier and this rule has

been followed in the syntax given here. However the same compiling function is used for global and local declarations so the generalisation of this rule would be a syntactic change only. In checking the syntactic status, it is necessary first of all to check that the syntactic status of the identifier is compatible with the other members of the list of identifiers being declared. For this the arity of the function is needed and an indication of whether the identifier is to be a relation or not. This is given as an integer by the following function.

```
arity ≙ λ Id • μ n : 0..3
                  | synstat = ident ⇒ n = 0
                    synstat = preop ∨ synstat = postop
                      ∨ synstat ∈ rng encop ⇒ n = 1
                    synstat = op ∨ synstat ∈ rng distinop
                      ∨ synstat ∈ rng distpreop ⇒ n = 2
                    synstat = rel ⇒ n = 3
                  • n
```

The check for correct status is given by

```
┌─ status_ok ──────────────────────────────
│ idlist? : seq Id
│ n : 0..3
│ ty?, ty! : Type
├──────────────────────────────────────────
│ n = arity(hd idlist?)
│ ∀ ident : rng idlist? • arity ident = n
│ ty! = ty?
└──────────────────────────────────────────
```

with error case:

```
┌─ status_error ───────────────────────────
│ idlist? : seq Id
│ rep! : seq Char
│ n : 0..3
│ ty?, ty! : Type
├──────────────────────────────────────────
│ n = arity(hd idlist?)
│ ¬(∀ ident : rng idlist? • arity ident = n)
│ rep! = "Mixture of operator symbols"
│ ty? = type_undefined
└──────────────────────────────────────────
```

No corrective action is taken to remove consequential errors. The type check takes place after normalisation so the test for correct type need not take account of type variables.

```
┌─ type_ok ───────────────────────────────────────────────
│ n : 0..3;  ty?, ty! : Type
├─────────────────────────────────────────────────────────
│ n = 0 ∧ ty? ∈ rng powerset ∧ ty! = ty?
│ ∨ n = 1
│   ∃ ty1, ty2 : Type
│   • ty? = powerset(powerset(tuple(ty1, ty2)))
│   ty! = ty?
│ ∨ n = 2
│   ∃ ty1, ty2, ty3 : Type
│   • ty? = powerset(powerset(tuple(tuple(ty1, ty2), ty3)))
│   ty! = ty?
│ ∨ n = 3
│   ∃ ty1, ty2 : Type
│   •  ty? = powerset(powerset ty')
│      ty! = powerset(powerset(tuple(ty', predicate)))
│    where
│      ty' ≙ tuple(ty1, ty2)
└─────────────────────────────────────────────────────────
```

In the error case the symbols will be declared with undefined type, to reduce consequential errors:

```
┌─ type_error ────────────────────────────────────────────
│ rep! : seq Char
│ n : 0..3;  ty?, ty! : Type
├─────────────────────────────────────────────────────────
│ ¬type_ok
│ n = 0 ⇒ rep! = "The term given is not a type"
│ n ≠ 0 ⇒ rep! = "Inappropriate type for operator symbol"
│ ty! = powerset type_undefined
└─────────────────────────────────────────────────────────
```

The declaration is simply given by

```
┌─ declare_ids ──────────────────────────
│ Declare[idlist?/new_ids]
│ ty? : Type
├────────────────────────────────────────
│ ty? ∈ rng powerset ∧ ty = powerset⁻¹ ty?
└────────────────────────────────────────
```

and the compiling function by

```
dec_ids ≙ (status_error ∨ status_ok) ≫ Normalise ≫
          (type_error ∨ type_ok) ≫ declare_ids\(ty)
```

<u>Inclusions</u>

```
inclusion = reference <open_schema-ptype>;
```

For schema inclusions it is simply necessary to merge in the schema identifiers into

the current environment, after checking that the reference is to a schema and that it does not have an undefined type.

```
┌─ include_schema ──────────────────────────
│ Schema; Include
├───────────────────────────────────────────
│ merge_ids = schema_type⁻¹(powerset⁻¹ ty?)
└───────────────────────────────────────────

┌─ not_schema_term ─────────────
│ ty? : Type
│ rep! : seq Char
├───────────────────────────────
│ ¬Schema
│ rep! = "Not a schema term"
└───────────────────────────────

open_schema ≙ Normalise ≫ (include_schema\(merge_ids)
                            ∨ not_schema_term
                           )
```

A schema inclusion may also appear in a hypothesis where it appears syntactically to be a predicate:

```
hyp =
   pred <check_pred_schema-ptype>,
   sdec_list cbar pred <unstack_pred-ptype>;
```

The allowable types for a predicate at this point in the syntax are predicate, the undefined type or a schema; if it is not one of those, an error is reported. If it is a schema then it is included.

```
┌─ pred ─────────────────────────────────────────────────────
│ ty? : Type
│ rep! : seq Char
├────────────────────────────────────────────────────────────
│ ty? = predicate ∨ ty? = type_undefined ∨ not_schema_term
└────────────────────────────────────────────────────────────
```

Note that predicates should contain no type variables, so the type must be normalised to give a specification for the compiling function as

```
check_pred_schema ≙ Normalise ≫
                    (pred ∨ (¬pred ∧ include_schema\(merge_ids)))
Z_dec_and_inc keeps start_idlist, stack_idlist, dec_ids,
                    open_schema, check_pred_schema
```

CHAPTER 12

SYNTACTIC, DATATYPE AND SCHEMA DEFINITIONS

```
Z_datatypes :Module
Z_scopes :Module
Z_type_norm :Module
Z_dec_and_inc :Module
```

Syntactic definitions

Like declarations, syntactic definitions occur at various points in the syntax and, for the non-generic cases are very similar to declarations. The main difference from the type-checking point of view is that whereas in x : term, term must have the type of a set of the type of x, in x ≙ term, they have the same type. Consequently, syntactic definitions are made to appear like declarations. The syntax for the syntactic definition of a single identifier is

```
syn_def_id =
   global_id <start_idlist--idlist> def term <syn_def-pidlist-ptype>;
```

and it is imply necessary to add a powerset to the type of the term to have an equivalent operation to declaration:

```
_ add_powerset ________
| ty?, ty! : Type
|______________________
| ty! = powerset ty?
|______________________
```

syn_def ≙ add_powerset ⨠ dec_ids

Datatype definitions

```
datatype_def = id <dt_def--id> becomes branches <unstack_id-pid>;

branch =
   id <dt_constant-qid>,
   id lang term rang <dt_constructor-ptype-qid>;

branches = branch, branch bbar branches;
```

For a datatype definition, the identifier must be declared immediately because the definition is allowed to be recursive. Datatypes are in fact allowed to be mutually recursive, but in order to keep to the declaration before use rule, datatypes used before being defined in a datatype definition must have been previously introduced as a given set. For this reason, the merge operation is used rather than declare. Apart from this case of introduction as a given set, the view has been taken that datatype names should be unique, not only within the current document but also within any referenced documents, including the standard library. This avoids some problems of confusing types and allows datatypes to be uniquely specified from their name and to have the same type as a given set.

```
┌─ dt_dec ──────────────────────────────────────────
│ Merge
│ ΔLexState
│ id! : Id
├───────────────────────────────────────────────────
│ id! = hd idlist
│ ¬(id! ∉ dom(find θEnv)
│   ∨ find θEnv id! = powerset(given id!)) ⇒
│      rep! = "Datatype not unique"
│ merge_ids = {id! ↦ powerset(given id!)}
│ ctuple' = ctuple
│ idlist' = tl idlist
└───────────────────────────────────────────────────

dt_def ≙ dt_dec\(merge_ids)
```

The datatype constants are straightforward:

```
┌─ dt_constant_dec ─────────────────────────────────
│ Declare
│ ΔLexState
│ id? : Id
├───────────────────────────────────────────────────
│ new_ids = ⟨hd idlist⟩ ∧ ty = given id?
│ idlist' = tl idlist
└───────────────────────────────────────────────────

dt_constant ≙ dt_constant_dec\(new_ids, ty)
```

For the constructor functions, the requirement that recursive references to the datatype involve only finite sets is regarded as a proof obligation, rather than a failure of type checking.

```
┌─ dt_constructor_dec ──────────────────────────────
│ Declare
│ ΔLexState
│ id? : Id;  ty? : Type
├───────────────────────────────────────────────────
│ new_ids = ⟨hd idlist⟩
│ idlist' = tl idlist
│ ty = powerset(tuple(ty?, given id?))
└───────────────────────────────────────────────────

dt_constructor ≙ Normalise ≫ dt_constructor_dec\(new_ids, ty)
```

On the completion of a datatype declaration, the SID Id stack must be reset using the unstack_idfunction, but as the SID stacks are not modelled in the specification, the corresponding operation is not specified here.

Schema definitions

For compatibility with declarations and definitions, the schema name is stacked as a list of identifiers and the schema is declared using syn_def.

```
schema_def =
   id <start_idlist--idlist>
   schema_definition <syn_def-pidlist-ptype>,
   id <start_idlist--idlist> gen_params
   schema_definition <end_scope><syn_def-pidlist-ptype>;

schema_definition =
   <check_schema_id-pidlist--idlist> sdef schema_term,
   <check_schema_id-pidlist--idlist> schema;
```

For a schema name, the identifier must be undecorated:

```
__check_schema_id ________________________________
| idlist!, idlist? : seq Id
| rep! : seq Char
|__________________________________________________
|  schema_id.att ≠ noatt ∨ schema_id.version ≠ noname ⇒
|     rep! = "Schema name must be undecorated"
|  idlist! = ⟨id'⟩
| where
|  schema_id ≙ hd idlist?
|  id' ≙ μ Id
|        | name = schema_id.name
|          version = noname ∧ att = noatt
|          synstat = ident
|        • θId
|__________________________________________________
```

The syntax for schemas is

```
schema =
   sb <new_scope> dec_list <scope_to_schema_type--type> esb,
   sb <new_scope> dec_list
   st pred_list <unstack_pred-ptype><scope_to_schema_type--type> esb,
   sch <new_scope> dec_list <scope_to_schema_type--type> esch,
   sch <new_scope> dec_list
   cbar pred <unstack_pred-ptype><scope_to_schema_type--type> esch;
```

For a schema, the type is derived from the scope which was created for the schema signature, and the scope discarded:

```
__scope_to_schema_type ____________________
| end_scope
| ty! : Type
|__________________________________________
| ty! = powerset(schema_type (hd blocks).ids)
|__________________________________________
```

The function unstack_pred disposes of the type produced by pred_list, and will be discussed later.

```
Z_syn_data_schema_def keeps syn_def, dt_def, dt_constant,
                            dt_constructor, check_schema_id,
                            scope_to_schema_type
```

CHAPTER 13

OPERATOR AND GENERIC SET DEFINITIONS

```
Z_datatypes :Module
Z_scopes :Module
Z_type_norm :Module
```

Declaration of operator symbols

At various points within the syntax it is possible to indicate the syntactic status of the identifier being defined so that it becomes an infix, postfix or prefix operator symbol or relation, with a consequential constraint on its type. The syntactic status is dealt with immediately after encountering the identifier in the definition while the compatibility of the type with the arity of the symbol is checked on completion of the declaration.

```
global_id =
    id,
    id underline <global_sym-1>,
    underline id <global_sym-2>,
    id underline id <global_sym-5>,
    lpar underline id underline rpar <global_sym-3>,
    underline id underline <global_sym-4>,
    id underline id underline <global_sym-6>,
    underline id underline id <global_sym-7>;
```

```
_ global_sym ____________________________________________
 ΔLexState
 n? : 1..7
_________________________________________________________
 hd idlist' = μ Id; Id'
              | θId = hd idlist
               name' = name ∧ version' = version ∧ att' = att
                 synstat' = syn
               where
                  | syn : Synstatus
                  |_____________________________________
                  | n? = 1 ➡ syn = preop
                  | n? = 2 ➡ syn = postop
                  | n? = 3 ➡ syn = op
                  | n? = 4 ➡ syn = rel
                  | n? = 5 ➡ syn = encop(idlist 2).name
                  | n? = 6 ➡ syn = distpreop(idlist 2).name
                  | n? = 7 ➡ syn = distinop(idlist 2).name
              • θId'
 n? > 4 ⇒ tl idlist' = tl(tl idlist)
 n? ≤ 4 ⇒ tl idlist' = tl idlist
_________________________________________________________
```

Syntactic definition of generic operators

During a generic syntactic definition, the syntactic status of an identifier may also be

defined, but in this case the parameter positions are indicated by the presence of generic types. In this one pass system, the syntactic status of the identifier is established at the definition itself, so the various sorts of syntactic definition all appear to be a succession of up to three identifiers. The syntax below provides the semantic functions to enable this to be sorted out.

```
syn_def_ids =
    id id <prepostsymbol--idlist> def term <syn_def-pidlist-ptype>,
    id id id <insetsymbol--idlist>
       def term <syn_def-pidlist-ptype>;
```

Prefix and postfix generic set definitions are syntactically equivalent to `id id` but may be distinguished semantically by whether the identifiers have been declared and whether they are generic types or not (one should be undeclared, one should have generic type). This is done by the `prepostsymbol` compiling function which delivers an identifier list containing a single identifier which is the generic set. The first case is with the first parameter the generic type and the second a postfixed generic set.

```
check_genpar1
ΞEnv
ΔLexState
idlist! : seq Id
---------------------------------------------
idlist(1) ∈ dom(find θEnv)
  ty ∈ rng powerset
  powerset⁻¹ ty ∈ rng generic
where
  ty ≙ find θEnv (idlist(1))
idlist(2) ∉ dom(find θEnv)
idlist! = ⟨μ Id; Id'
            | θId = idlist(2)
              name' = name ∧ version' = version
              att' = att ∧ synstat' = postset (idlist(1)).name
            • θId'
           ⟩
idlist' = tl(tl idlist)
```

In the second case, the first identifier is a prefix generic set and the second the generic type:

```
check_genpar2
ΞEnv
ΔLexState
idlist! : seq Id
-------------------------------------------
idlist(1) ∉ dom(find θEnv)
idlist(2) ∈ dom(find θEnv)
  ty ∈ rng powerset
  powerset⁻¹ ty ∈ rng generic
where
  ty ≙ find θEnv (idlist(2))
idlist! = (μ Id; Id'
          | θId = idlist(1)
            name' = name ∧ version' = version
            att' = att ∧ synstat' = preset (idlist(2)).name
          • θId'
          )
idlist' = tl(tl idlist)
```

For the error case, the identifier list is constructed arbitrarily using the first identifier.

```
genpar_error
ΔLexState
idlist! : seq Id
rep! : seq Char
-------------------------------------------
idlist! = ⟨idlist(1)⟩
rep! = "Incorrect operator definition"
idlist' = tl(tl idlist)
```

```
prepostsymbol ≙ genpar_error ● (check_genpar1 ∨ check_genpar2)
```

For infixed operators, a succession of 3 ids, the generic parameters must be the outermost identifiers. These are checked to be generic and the middle identifier made into an infixed generic set.

```
pars_ok
ΞEnv
ΔLexState
--------------------------------------------
idlist(1) ∈ dom(find θEnv)
  ty ∈ rng powerset
  powerset⁻¹ ty ∈ rng generic
where
  ty ≙ find θEnv (idlist(1))
idlist(3) ∈ dom(find θEnv)
  ty ∈ rng powerset
  powerset⁻¹ ty ∈ rng generic
where
  ty ≙ find θEnv (idlist(3))
idlist(2) ∉ dom(find θEnv)
```

```
pars_not_ok
ΞEnv
ΔLexState
rep! : seq Char
--------------------------------------------
¬pars_ok
rep! = "Incorrect identifier for parameter of generic"
       " operator"
```

In either case the middle identifier is made into an infixed generic set, constructed from the parameter names. These are used when the generic set is instantiated (see below).

```
make_inset
ΔLexState
idlist! : seq Id
--------------------------------------------
idlist! = ⟨μ Id; Id'
           | θId = idlist(2)
             name' = name ∧ version' = version ∧ att' = att
             synstat' = inset⟨(idlist(1)).name,
                              (idlist(3)).name⟩
           • θId'
           ⟩
idlist' = tl(tl(tl idlist))
```

```
insetsymbol ≙ (pars_ok ∨ pars_not_ok) ∧ make_inset
```

Instantiation of generic sets

```
Z_mk_instantiation :Module
Z_named_inst :Module
```

```
formula =
   form1 inset <setop--id> formula <inset-pid-ptype-ptype--type>,
   form1;

form1 =
   ...
   form2;

form2 =
   ...
   form3;

form3 =
   ...
   preset <setop--id> form3 <set_inst1-pid-ptype--type>,
   form4;

form4 =
   ...
   form4 postset <set_inst2-ptype--type>,
   aform;
```

The compiling function setop is simply required to stack the name of the identifier:

```
┌─ setop ──────────────────────────────
│ ΔLexState
│ ident! : Id
├──────────────────────────────────────
│ ident! = hd idlist ∧ idlist' = tl idlist
└──────────────────────────────────────
```

To carry out the instantiation, a binding list instantiation is made up using the name stored with the syntactic status of the generic identifier:

```
┌─ set_inst ───────────────────────────
│ inst! : Instantiation
│ ty?, ty! : Type
│ ident? : Id
│ ΞEnv
├──────────────────────────────────────
│   inst! = binding_list{ident ↦ ty?}
│   ty! = find θEnv ident?
│ where
│   ident ≙ μ Id
│           | name = preset⁻¹ ident?.synstat
│             version = noname
│             att = noatt ∧ synstat = ident
│           • θId
└──────────────────────────────────────
```

and the compiling functions are given by

```
set_inst1 ≙ Check_type ⨠ set_inst ⨠ inst
set_inst2 ≙ setop ⨠ set_inst1
```

For the infixed generic sets, two sets have to be instantiated

```
┌─ inset_inst ──────────────────────────────
│ inst! : Instantiation
│ tyl?, tyr?, ty! : Type
│ ident? : Id
│ ΞEnv
├───────────────────────────────────────────
│   inst! = binding_list{id1 ↦ tyl?, id2 ↦ tyr?}
│   ty! = find θEnv ident?
│ where
│   id1 ≙ μ Id
│         | name = (inset⁻¹ ident?.synstat) 1
│           version = noname
│           att = noatt ∧ synstat = ident
│         • θId
│   id2 ≙ μ Id
│         | name = (inset⁻¹ ident?.synstat) 2
│           version = noname
│           att = noatt ∧ synstat = ident
│         • θId
└───────────────────────────────────────────
```

```
inset ≙ Check_type[tyl?/ty?, tyl!/ty!]
        ∧ Check_type[tyr?/ty?, tyr!/ty!]
        ⨠ inset_inst ⨠ inst
Z_op_def keeps global_sym, prepostsymbol, insetsymbol,
               setop, set_inst1, set_inst2, inset
```

CHAPTER 14

PRIMITIVE TYPES

Z_datatypes :Module

Z_type_unify :Module

Z_scopes :Module

Explicit constructions

```
explicit_constr =
    tuple,
    set eset <empty_set--type>,
    explicit_set termlist1 eset <explicit_set-ptype--type>,
    lseq rseq <empty_list--type>,
    lseq termlist1 rseq <explicit_list-ptype--type>;
```

In the syntax above, explicit_set is a terminal symbol inserted by a look-ahead function in the lexical analyser to resolve the problem of disentangling {a, b, c} from {a, b, c : T...}. The compiling functions required are relatively trivial; for an empty set the type required is powerset of variable:

```
__empty_set ____________________________
| ΔTypeState
| ty! : Type
|_______________________________________
|   ty! = powerset(variable n)
|   tenv' = tenv ∧ valid_names' = {n} ∪ valid_names
| where
|   n : TName | n ∉ valid_names
|_______________________________________
```

and similarly for an empty list:

```
__empty_list ___________________________
| ΔTypeState
| ty! : Type
|_______________________________________
|   ty! = powerset(tuple(Ztype, variable n))
|   tenv' = tenv ∧ valid_names' = {n} ∪ valid_names
| where
|   n : TName | n ∉ valid_names
|_______________________________________
```

An explicit set is a powerset of its elements:

```
__explicit_set ______
| ty!, ty? : Type
|____________________
| ty! = powerset ty?
|____________________
```

and similarly for an explicit list:

```
explicit_list
  ty!, ty? : Type
  ------------------------------
  ty! = powerset(tuple(Ztype, ty?))
```

Both sets and lists must be made up of elements of the same type:

```
termlist1 = # terms of the same type #
   term,
   term comma termlist1a;

   termlist1a =
      term <check_tys_same-ptype-ptype--type>,
      term <check_tys_same-ptype-ptype--type> comma termlist1a;
```

An arbitrary choice is made to deliver the first type in the series:

```
check_tys_same
  TypeCheck[ty1?/ty1, ty2?/ty2]
  ty! : Type
  ------------------------------
  ty! = ty2?
```

Special constants

The various constants appear as members of the atomic formulae:

```
aform =
   <nat--type> nat,
   <char--type> char,
   <sconst-lv--type> sconst,
   ...
```

The special constants are the numbers and strings, distinguished by the type of the lexical value delivered by the lexical analyser.

```
sconst
  lv? : LexVal; ty! : Type
  ------------------------------
  lv? ∈ rng num ∧ ty! = Ztype
  ∨
  lv? ∈ rng char ∧ ty! = Chartype
  ∨
  lv? ∈ rng string ∧ ty! = powerset(tuple(Ztype, Chartype))
```

The terminal symbols nat and char are Z and Char respectively; they are built into the syntax in this way in order to prohibit their re-definition.

```
┌─ nat ──────────────────────
│ ty! : Type
├────────────────────────────
│ ty! = powerset Ztype
└────────────────────────────
```

```
┌─ char ─────────────────────
│ ty! : Type
├────────────────────────────
│ ty! = powerset Chartype
└────────────────────────────
```

Projections

```
aform =
   ...
   aform proj id <proj-ptype--type>,
   ...
```

A projection may only be applied to a schema type and must identifiy a member of the schema's signature. In the error cases the type is left unchanged.

```
┌─ proj_ok ──────────────────
│ ΔLexState
│ ty?, ty! : Type
├────────────────────────────
│ ty? ∈ rng schema_type
│   hd idlist ∈ dom idmap
│   ty! = idmap(hd idlist)
│ where
│   idmap ≙ schema_type⁻¹ ty?
│ idlist' = tl idlist
└────────────────────────────
```

```
┌─ not_schema ───────────────
│ ΔLexState
│ ty?, ty! : Type
│ rep! : seq Char
├────────────────────────────
│ ty? ∉ rng schema_type
│ idlist' = tl idlist
│ ty! = ty?
│ rep! = "Projection may only be applied to schemas"
└────────────────────────────
```

```
┌─ id_not_in_sig ────────────
│ ΔLexState
│ ty?, ty! : Type
│ rep! : seq Char
├────────────────────────────
│ ty? ∈ rng schema_type
│ hd idlist ∉ dom(schema_type⁻¹ ty?)
│ ty! = ty?
│ idlist' = tl idlist
│ rep! = "Identifier not defined in schema"
└────────────────────────────
```

```
proj ≜ not_schema v id_not_in_sig v proj_ok
Z_primitives keeps empty_set, empty_list, explicit_set,
                   explicit_list, check_tys_same,
                   nat, char, sconst, proj
```

CHAPTER 15

TUPLES, PRODUCTS, THETA TERMS AND COMPREHENSIONS

```
Z_datatypes :Module
Z_scopes :Module
Z_type_unify :Module
Z_primitives :Module
```

Syntax

```
tuple =
   lpar term comma termlist2 rpar,
   theta reference <theta-ptype--type>;
   # the reference is to a schema #

   termlist2 = #terms for a tuple #
      term <first_tuple-ptype--type>,
      term <first_tuple-ptype--type> comma termlist2a;

      termlist2a =
         term <next_tuple-ptype-ptype--type>,
         term <next_tuple-ptype-ptype--type> comma termlist2a;
```

Tuples

An explicit tuple is easily compiled as it is simply a matter of combining the types into a list:

```
first_tuple
  ty?, ty! : Type
  ----------------
  ty! = tuple(ty?)
```

```
next_tuple
  ty?, tuple?, ty! : Type
  ----------------
  ty! = tuple((tuple⁻¹tuple?) snoc ty?)
```

Theta expressions

A theta expression forms a tuple, having a schema type, from identifiers defined within the current environment and members of the schema referenced. Unlike a simple schema reference the type of the delivered result is an element, not a set. As the theta expression only involves a reference, rather than an expression it is not necessary to use type unification.

```
theta_schema
  Schema_ok
  ty! : Type
  ----------------
  ty! = powerset⁻¹ ty?
```

For the error cases, the type is passed through unchanged:

```
┌─ not_schema ──────────────────
│ ty?, ty! : Type
│ rep! : seq Char
├───────────────────────────────
│ ¬Schema
│ ty! = ty?
│ rep! = "Schema required here"
└───────────────────────────────
```

```
┌─ schema_undefined ─────────────────────────────────────
│ ΞEnv
│ ty?, ty! : Type
│ rep! : seq Char
├────────────────────────────────────────────────────────
│ Schema ∧ ¬Schema_ok
│ ty! = ty?
│ rep! = "Schema identifiers not present in this environment"
└────────────────────────────────────────────────────────
```

```
theta ≙ not_schema ∨ schema_undefined ∨ theta_schema
```

Products

```
product = term <first_prod-ptype--type> prod product1
          <end_prod-ptype--type>;

    product1 =
        term <next_prod-ptype-ptype--type>,
        term <next_prod-ptype-ptype--type> prod product1;
```

A Cartesian product is formed from sets and forms a set of tuples of the constituent types. The compiling functions are similar to those for tuples with the additional complication of removing powerset constructors: to avoid problems with type variables, this has to be done by type checking against a type consisting of a powerset of a new variable type. The powerset constructor for the tuple is added at the end of the product.

```
┌─ prod ─────────────────────────────────
│ TypeCheck[ty?/ty1, ptype/ty2]
│ ty! : Type
├────────────────────────────────────────
│    ty! = variable n
│    valid_names' = valid_names ∪ {n}
│ where
│   │ n : TName
│   ├──────────────
│   │ n ∉ valid_names
│ ptype = powerset ty!
└────────────────────────────────────────
```

```
┌─ first_member ──────────
│ ty?, ty! : Type
├─────────────────────────
│ ty! = tuple(ty?)
└─────────────────────────
```

```
first_prod ≙ prod\(ptype) ≫ first_member
```

The same procedure is applied to subsequent members of the product, with the result being added to the end of the list:

```
┌─ next_member ──────────────────────────────
│ prod[this_prod/ty!]
│ prod?, ty! : Type
├────────────────────────────────────────────
│ ty! = tuple((tuple^-1 prod?) snoc this_prod)
└────────────────────────────────────────────
```

```
next_prod ≙ next_member\(ptype, this_prod)
```

Finally, the powerset constructor is added at the end of the product:

```
┌─ end_prod ──────────
│ ty?, ty! : Type
├─────────────────────
│ ty! = powerset ty?
└─────────────────────
```

<u>Comprehension terms</u>

```
comprehension =
   schema,
   set <new_scope> dec_list comp_set eset,
   lambda <new_lambda_scope>
   dec_list lambda_set <lambda-ptype--type>,
   mu <new_scope> dec_list lambda_set <end_scope>;

   comp_set =
      <scope_to_tuple--type>,
      cbar pred <unstack_pred-ptype><scope_to_tuple--type>,
      dot term <set-ptype--type>,
      cbar pred <unstack_pred-ptype> dot term <set-ptype--type>;

   lambda_set =
      dot formula,
      cbar pred <unstack_pred-ptype> dot formula;
```

The standard set comprehension is defined to deliver a set of tuples, formed either from the characteristic tuple of the declarations in the comprehension or as provided by the example term. A new scope is created on entry to the comprehension and converted into a tuple using the function below.

```
┌─ tuple_of_scope ──────────────────────────
│ ΔEnv
│ ty! : Type
├───────────────────────────────────────────
│ #ctuple > 1 ⇒ ty! = powerset(tuple ctuple)
│ #ctuple = 1 ⇒ ty! = powerset(hd ctuple)
└───────────────────────────────────────────
```

```
scope_to_tuple ≙ tuple_of_scope ⨾ end_scope
```

The set function is called when an example term is provided and adds a powerset to

its input type and ends the current scope. Consequently it is a composition of previously defined operations:

```
set ≙ explicit_set ⨾ end_scope
```

The λ comprehension uses the scope for the parameter type and the example term for the result type:

```
┌─ lambda1 ─────────────────────────────────
│ scope_to_tuple[typar/ty!]
│ ty!, ty? : Type
├───────────────────────────────────────────
│ ty! = powerset(tuple(powerset⁻¹ typar, ty?))
└───────────────────────────────────────────

lambda ≙ lambda1\(typar)
Z_tuples keeps first_tuple, next_tuple, theta,
                first_prod, next_prod, end_prod,
                scope_to_tuple, set, lambda
```

CHAPTER 16

FUNCTION APPLICATION AND PARTIAL APPLICATION

```
Z_datatypes :Module
Z_scopes :Module
Z_type_unify :Module
Z_references :Module
Z_anon_inst :Module
```

Type checking for function application

Most of the syntax for function application is concerned with indicating the binding of the various forms of operator, and then, for the infixed forms, assembling the parameters into tuples.

```
formula =
   ...
   form1;

form1 =
   form1 op <funop--type> form2 <infix-ptype-ptype-ptype--type>,
   form2;

form2 =
   form2 form3 <funapp-ptype-ptype--type>,
   form3;

form3 =
   preop <funop--type> form3 <funapp-ptype-ptype--type>,
   ...
   distpreop <funop--type>
      term eop form3 <distpreop-ptype-ptype-ptype--type>,
   powerset form3 <powerset_fn-ptype--type>,
   form4;

form4 =
   form4 distinop <funop--type>
      term eop <infix-ptype-ptype-ptype--type>,
   form4 postop <postapp-ptype--type>,
   ...
   aform;
```

The specification for the basic type checking of function application is, loosely, that given a supposed function, of type t1, an argument of type t2, one creates a new type t3 which will be the type of the delivered result. The type checking consists in the unification of t1 with t2 -> t3.

```
┌─ funapp1 ──────────────────────────────
│ TypeCheck[fun?/ty1, fun/ty2]
│ par?, ty! : Type
├──────────────────────────────────────
│   ty! = variable n
│   valid_names' = valid_names ∪ {n}
│ where
│  │ n : TName
│  ├──────────────
│  │ n ∉ valid_names
│ fun = powerset(tuple (par?, ty!))
└──────────────────────────────────────
```

funapp2 ≙ funapp1\(fun)

Note that this is not a complete specification for the compiling function funapp as it is necessary to take account of the possibility of term term being a set membership predicate. This is dealt with later.

The function funop is used when an operator symbol has been recognised, and is equivalent to an identifier reference followed by anonymous instantiation.

funop ≙ reference ⨾ anon_inst

For infixed application it is necessary to calculate the parameter type:

```
┌─ infix1 ──────────────────────
│ funapp2
│ rhpar?, lhpar? : Type
├──────────────────────────────
│ par? = tuple (lhpar?, rhpar?)
└──────────────────────────────
```

infix ≙ infix1\(par?)

and postfixed application is a combination of an operator symbol and function application.

postapp ≙ $\text{funop}_{[\text{fun!}/\text{ty!}]}$ ≫ funapp2

The specification for distpreop is the same as that for infix: the implementation differs only in the order of the parameters, which is determined by the order in which the types are stacked which differs in the two cases.

The powerset function is completely trivial:

```
┌─ powerset_fn ─────────
│ ty?, ty! : Type
├──────────────────────
│ ty! = powerset ty?
└──────────────────────
```

Type checking for set membership

This is syntactically the same as function application (term term), but indicates a predicate rather than a term. Type checking of this phrase assumes function application; if this fails, type checking for set membership is tried; if this fails the reply for function application is delivered.

```
┌─ funapp_ok ──────┐
│ funapp2          │
├──────────────────┤
│ rep! = "OK"      │
└──────────────────┘
```

```
┌─ setmem ───────────────────────────────┐
│ TypeCheck[set?/ty1, set/ty2]           │
│ mem?, ty! : Type                       │
├────────────────────────────────────────┤
│ set = powerset mem?                    │
│ ty! = predicate                        │
│ rep! = "OK"                            │
└────────────────────────────────────────┘
```

funapp ≙ (funapp2 ● (setmem\(set))) ● funapp_ok

Partial application

Partial application consists of operator or relation symbols considered as a term in their own right, and infixed operators with one parameter supplied. The syntax is fairly complicated to take into account the various forms of operator, but only a small number of compiling functions are required.

```
partials =
   underline rel underline <partrel--type>,
   underline op <funop--type> form2 <partop1-ptype-ptype--type>,
   aform op underline <partop2-ptype--type>,
   underline op underline <funop--type>,
   underline distinop <funop--type>
      term eop <partop1-ptype-ptype--type>,
   aform distinop underline eop <partop2-ptype--type>,
   underline distinop underline eop <funop--type>,
   distpreop underline eop underline <funop--type>,
   distpreop <funop--type>
      term eop underline <partop1-ptype-ptype--type>,
   distpreop underline eop <funop--type>
      form3 <partop1-ptype-ptype--type>,
   encop underline eop <funop--type>,
   preop underline <funop--type>,
   underline postop <funop--type>;
```

If no parameters are supplied for operator symbols then the type of the symbol is all that is required; however, for a relation it is necessary to remove the predicate result from the type. This can be done directly as it is derived from the identifier look-up.

```
┌─ reltype ──────────────────────────────┐
│ ty?, ty! : Type                        │
├────────────────────────────────────────┤
│   ty! = powerset(hd tys)               │
│ where                                  │
│   tys ≙ tuple⁻¹(powerset⁻¹ ty?)        │
└────────────────────────────────────────┘
```

```
partrel ≙ (reference >> reltype) ⨾ anon_inst
```

For partial application proper, a variable type is supplied for the missing parameter, and then the function infix is used to calculate what the result type would be. This then gives the type for the partial application as a function from the variable type to the result type. When the left hand parameter is supplied this is

```
┌─ partop ──────────────────────────────
│ infix[tyres/ty!]
│ ty! : Type
├───────────────────────────────────────
│   lhpar? = variable n
│   valid_names' = valid_names ∪ {n}
│ where
│   │ n : TName
│   ├──────────────────
│   │ n ∉ valid_names
│ ty! = powerset(tuple(lhpar?, tyres))
└───────────────────────────────────────

partop1 ≙ partop\(tyres, lhpar?)
```

The right hand parameter case is a simple variation on this.

```
partop2 ≙ funop[fun!/ty!] >>
          partop[lhpar?/rhpar?, rhpar?/lhpar?]\(tyres, rhpar?)
Z_funapps keeps funop, infix, postapp, powerset, funapp,
                partrel, partop1, partop2
```

CHAPTER 17

RELATIONS AND PREDICATES

```
Z_datatypes :Module
Z_scopes :Module
Z_type_unify :Module
Z_type_norm :Module
Z_funapps :Module
```

Relations

Relations involve a straightforward variation on type checking for function application. The syntax is

```
rel_exp =
   term member term <member-ptype-ptype--type>,
   term equals equals_tail <to_pred-ptype--type>,
   term rel <funop--type> rel_tail <to_pred-ptype--type>,
   apred;

   equals_tail =
      term <equals-ptype-ptype--type> tail;

      tail =
         $,
         rel <funop--type> rel_tail,
         equals equals_tail;

   rel_tail =
      term <rel-ptype-ptype-ptype--type> tail;
```

The specification for set membership is:

```
┌─ member1 ─────────────────         member ≙ member1\(set)
│ TypeCheck[set?/ty1, set/ty2]
│ mem?, ty! : Type
├───────────────────────────
│ set = powerset mem?
│ ty! = predicate
└───────────────────────────
```

while that for equality allows for the continued form and delivers the type of the right hand operand:

```
┌─ equals ──────────────────
│ TypeCheck[ty1?/ty1, ty2?/ty2]
│ ty! : Type
├───────────────────────────
│ ty! = ty2?
└───────────────────────────
```

A relation is similar to an infixed function application, and like equality delivers the type of the right hand operand for the continued form.

```
┌─rel1 ──────────┐     rel ≙ rel1 \(pred)
│ infix[pred/ty!]
│ ty! : Type
├────────────────
│ ty! = rhpar?
└────────────────
```

On completion of a relation or equality, a predicate is delivered:

```
┌─to_pred ──────
│ ty?, ty! : Type
├────────────────
│ ty! = predicate
└────────────────
```

Note that with this specification, the terms need not be completely defined, although the predicate result is. This allows an expression such as ∅ ∈ dom ⟨⟩ to type check correctly, even though it is still generic. This is allowed because the actual type may only be fixed as a result of the type checking of a complicated predicate involving several relations.

Predicates

In order to resolve various syntactic ambiguities, both predicates and terms are produced as a result of the expansion of the syntax rule for pred. In effect a predicate is formed by combining terms using the loosely binding operators of the predicate calculus. Once it has been established that a term is destined to be a predicate there are three allowable possibilities for the type: it may be a predicate, undefined or a schema. The last case breaks down into two according to whether the predicate is a schema inclusion in disguise and destined for the hypothesis part of a theorem or a predicate at any other position: in the former case the signature is merged into the current scope, in the latter it must be present within the current scope. The latter case is detected syntactically and checked using the functions check_pred and unstack_pred which occur throughout the syntax in situations such as the following:

```
log_exp =
   log_exp1,
   log_exp <unstack_pred-ptype> iff log_exp1
   <check_pred1-ptype--type>;
```

```
┌─check_predn ─────────────────────────────
│ ty?, ty! : Type
│ rep! : seq Char
├──────────────────────────────────────────
│ ¬(ty? = predicate ∨ ty? = type_undefined ∨ Schema)
│     ⇒ rep! = "Predicate required here"
└──────────────────────────────────────────
```

```
┌─ schema_wrong ──────────────────────────────────────────────
│ ΞEnv
│ ty?, ty! : Type
│ rep! : seq Char
├─────────────────────────────────────────────────────────────
│ Schema ∧ ¬Schema_ok
│ ty! = predicate
│ rep! = "Schema identifiers not present in this environment"
└─────────────────────────────────────────────────────────────
```

```
check_pred ≙ Normalise ⨠ check_predn ∨ schema_wrong ∨ Schema_ok
```

The other function, unstack_pred, is not required to deliver a type:

```
unstack_pred ≙ check_pred\(ty!)
Z_preds keeps member, equals, rel, to_pred,
              check_pred, unstack_pred
```

CHAPTER 18

SCHEMA EXPRESSIONS

```
Z_datatypes :Module
Z_scopes :Module
```

Quantified schema terms

```
quant_sexp = squant <new_scope> dec_list
             dot schema_term <subtract_scope-ptype--type>;
```

A new scope is formed for the quantified identifiers, which must all be present and with the correct type within the schema type. The new schema type is the difference between the two, assuming this is not empty.

```
subtract_scope1
ΞEnv
ty?, ty! : Type
rep! : seq Char
--------------------------------------------
ty! = powerset(schema_type ids')
dom good_ids = dom ids ⇒
   rep! = "All identifiers quantified"
¬(dom good_ids ⊆ dom ids) ⇒
   rep! = "Identifier to be quantified not present"
          "in schema"
bad_ids ≠ {} ⇒
   rep! = "Quantified identifier has inconsistent type"
where
 ids ≙ schema_type⁻¹(powerset⁻¹ ty?)
 quants ≙ (hd blocks).ids
 bad_ids ≙ quants inconsistent ids
 good_ids ≙ bad_ids ⩤ quants
 ids' ≙ dom good_ids ⩤ ids
```

```
subtract_scope ≙ subtract_scope1 ⨾ end_scope
```

Logical schema expressions

The infixed operators all have a similar form, exemplified by:

```
log_sexp =
   log_sexp1,
   log_sexp ziff log_sexp1 <stype2-ptype-ptype--type>;
```

The two schemas may be combined if their signatures are consistent.

```
stype2
  ty1?, ty2?, ty! : Type
  rep! : seq Char
  ------------------------------------------------
    ty! = powerset(schema_type(ids1 ∪ ids2'))
    bad_ids ≠ {} ⟶ rep! = "Schema terms inconsistent"
  where
    ids1 ≙ schema_type⁻¹(powerset⁻¹ ty1?)
    ids2 ≙ schema_type⁻¹(powerset⁻¹ ty2?)
    bad_ids ≙ ids1 inconsistent ids2
    ids2' ≙ bad ids ⩤ ids2
```

The special purpose schema expressions

```
spec_sexp =
   spec_sexp zhide lpar id_list rpar <hideids-pidlist-ptype--type>,
   spec_sexp zhide reference <hideref-ptype-ptype--type>,
   spec_sexp zcmp spec_sexp1 <scompose-ptype-ptype--type>,
   spec_sexp zpipe spec_sexp1 <pipe-ptype-ptype--type>,
   spec_sexp zovr spec_sexp1 <soverride-ptype-ptype--type>,
   spec_sexp1;
```

For hiding it is simply necessary to check that the identifiers are present in the schema type, and then remove them.

```
hideids
  idlist? : seq Id; ty?, ty! : Type
  rep! : seq Char
  ------------------------------------------------
    rng idlist? = dom ids ⟶
       rep! = "All identifiers hidden"
    ¬(rng idlist? ⊆ dom ids) ⟶
       rep! = "Identifier to be hidden not present in schema"
    ty! = powerset(schema_type ids')
  where
    ids ≙ schema_type⁻¹(powerset⁻¹ ty?)
    ids' ≙ rng idlist? ⩤ ids
```

For hiding with a schema it is necessary to check that the name is indeed that of a schema and that it is compatible with the schema to be hidden.

```
_hideref_ok ______________________________________________
| tyref?, tyschema?, ty! : Type
| rep! : seq Char
|_________________________________________________________
| Schema[tyref?/ty?]
|   ty! = powerset(schema_type ids')
|   bad_ids ≠ {} ⇒ rep! = "Schemas inconsistent"
|   ids' = {} ⇒
|      rep! = "All identifiers hidden"
|   ¬(good_ids ⊆ ids) ⇒
|      rep! = "Identifier to be hidden not present in schema"
| where
|   idsref ≙ schema_type⁻¹(powerset⁻¹ tyref?)
|   ids ≙ schema_type⁻¹(powerset⁻¹ tyschema?)
|   bad_ids ≙ idsref inconsistent ids
|   good_ids ≙ bad_ids ◀ idsref
|   ids' ≙ ids \ good_ids
|_________________________________________________________
```

```
_hideref_wrong ___________________________________
| tyref?, tyschema?, ty! : Type
| rep! : seq Char
|_________________________________________________
| ¬Schema[tyref?/ty?] ⇒
|    rep! = "Only schemas may be used for hiding"
| ty! = tyschema?
|_________________________________________________
```

```
hideref ≙ hideref_ok ∨ hideref_wrong
```

For the other schema operations, a few extra functions on sets of identifiers are needed. First of all, ids_with_decor delivers that part of a look-up function where the identifiers have a given decoration.

```
ids_with_decor ≙
  λ decor : Att
  • λ ids : Id ⇸ Type
    • {ident : dom ids | ident.att = decor} ◁ ids
```

Next, ids_with_basename delivers that part of a look-up function such that the identifiers have no attribute, have the same base name and version in the decorated function and deliver the same type.

```
ids_with_basename ≙
   λ ids, decids : Id ⇸ Type
   • {ident : dom ids
     | ident.att = noatt
       ∧ (∃ ident' : dom decids
           • ident'.name = ident.name
           ∧ ident'.version = ident.version
           ∧ decids ident' = ids ident)
     • ident ↦ ids ident
     }
```

For schema composition, find the set of identifiers present in both schemas in primed

and unprimed forms and take the intersection: this should be non-empty for schemas to be composed. The resulting type is simply that of the merged schemas.

```
── scompose ──────────────────────────────────────────
│ ty1?, ty2?, ty! : Type
│ rep! : seq Char
├──────────────────────────────────────────────────────
│   ty! = powerset(schema_type (ids2 ∪ good_ids))
│   undashed_ids1 ∩ undashed_ids2 = {} ⟹
│      rep! = "Schemas cannot be composed"
│   bad_ids ≠ {} ⟹ rep! = "Schemas inconsistent"
│ where
│   ids1 ≙ schema_type⁻¹(powerset⁻¹ ty1?)
│   ids2 ≙ schema_type⁻¹(powerset⁻¹ ty2?)
│   undashed_ids1 ≙ ids_with_basename(ids1,
│                        (ids_with_decor (dashes 1)) ids1)
│   undashed_ids2 ≙ ids_with_basename(ids2,
│                        (ids_with_decor (dashes 1)) ids2)
│   bad_ids ≙ ids1 inconsistent ids2
│   good_ids ≙ bad_ids ⩤ ids1
└──────────────────────────────────────────────────────
```

For piping it is necessary to find identifiers in one schema which have the same base name and version as those in another schema:

```
same_base ≙
   λ ids1, ids2 : Id ⇸ Type
   • {ident : dom ids1
     | ∃ ident' : dom ids2
       • ident'.name = ident.name
       ∧ ident'.version = ident.version
       ∧ ids1 ident = ids2 ident'
     • ident ↦ ids1 ident
     }
```

```
── pipe ──────────────────────────────────────────────
│ ty1?, ty2?, ty! : Type
│ rep! : seq Char
├──────────────────────────────────────────────────────
│   ty! = powerset(schema_type (ids2' ∪ good_ids))
│   bad_ids ≠ {} ⟹ rep! = "Schemas inconsistent"
│   piped_outputs = {} ⟹ rep! = "Schemas cannot be piped"
│ where
│   ids1 ≙ schema_type⁻¹(powerset⁻¹ ty1?)
│   ids2 ≙ schema_type⁻¹(powerset⁻¹ ty2?)
│   outputs ≙ ids_with_decor bang ids1
│   inputs ≙ ids_with_decor query ids2
│   piped_outputs ≙ same_base(outputs, inputs)
│   piped_inputs ≙ same_base(inputs, outputs)
│   ids1' ≙ ids1 \ piped_outputs
│   ids2' ≙ ids2 \ piped_inputs
│   bad_ids ≙ ids1' inconsistent ids2'
│   good_ids ≙ bad_ids ⩤ ids1'
└──────────────────────────────────────────────────────
```

The override function is equivalent to a logical operation as far as type checking is concerned:

```
soverride ≙ stype2
```

The pre condition is simply another variation on hiding:

```
spec_sexp1 =
    spec_sexp2,
    pre spec_sexp2 <pre-ptype--type>;
```

```
__pre ____________________________________________
 ty?, ty! : Type
 rep! : seq Char
__________________________________________________
   ty! = powerset(schema_type (ids \ preids))
   preids = {} ⇒
      rep! = "Schema not suitable for pre-condition"
 where
   ids ≙ schema_type^-1(powerset^-1 ty?)
   afterids ≙ ids_with_decor (dashes 1) ids
   preids ≙ afterids ∪ ids_with_decor bang ids
__________________________________________________
```

Finally, all the above operations presuppose an input type made up from a schema; this is checked at schema reference:

```
spec_sexp2 =
    lpar schema_term rpar,
    lpar schema_term rpar rename,
    reference <check_schema-ptype--type>,
    schema;
```

```
__check_schema ______________________________________________
 ty?, ty! : Type
 rep! : seq Char
_____________________________________________________________
 ¬Schema ⇒
        ty! = powerset(schema_type {ident ↦ type_undefined})
        rep! = "Not a schema type"
      where
        ident ≙ μ Id
                  | name = noname ∧ version = noname
                  ∧ att = noatt
                  • θId
 Schema ⇒ ty! = ty?
_____________________________________________________________
```

```
Z_schema_ops keeps subtract_scope, stype2, hideids, hideref,
                   scompose, pipe, soverride, pre, check_schema
```

REFERENCES

Currie I F. Private communication, 1984. Most of the information on SID is stored as on-line documentation in the Flex computing systems.

Foster J M. "A Syntax Improving Program", Computer J, vol 11, no 1, pp 31 - 34, May 1968.

Foster J M, Currie I F, Edwards P W. "Flex: a working computer with an architecture based on procedure values", Proc International Workshop on High Level Language Computer Architectures, Fort Lauderdale, Maryland, USA, 1982.

King S, Sorensen I H, Woodcock J. "Z: Grammar and Concrete and Abstract Syntaxes", version 1.1, July 28 1987, Programming Research Group, University of Oxford, 1987.

Milner R. "A Theory of Type Polymorphism in Programming", J Comp and System Science vol 17, p 348, 1978.

Robinson J A. "A machine-oriented logic based on the resolution principle" J Assoc Comp Mach, vol 12, pp 23 - 41, 1965.

Spivey J M. "Understanding Z: a Specification Language and its Formal Semantics", D Phil thesis, Programming Research Group, University of Oxford, 1985.

Sufrin B A. "Z Handbook, Draft 1.1", Programming Research Group, University of Oxford, 1986.

APPENDIX

THE Z SYNTAX

BASICS

```
id                  # as provided by lexical analysis, including
                      decoration#
document            # a specification module #
decor               #?,!,' or decor #

# General brackets and separators #

endz                # end of Z picture#
nl                  # hard new line#
semi                #;#
lpar                #(#
rpar                #)#
comma               #, or comma#
lsqb                #[ or lsqb#
ilsqb               # [ in a version = instantiation#
rsqb                #] or rsqb#
si                  # start indentation#
ei                  # end indentation#
keep                # export indicator #
finish              # end of file#

# Declarations and definitions #

colon               #: or colon#
cbar                #| constraint bar#
def                 #≙ or def syntactic equivalence for terms#
sdef                #≙ or sdef syntactic equivalence for schema terms#
becomes             # ::= for datatype definitions#
bbar                #| (branch separator)#
lang                #« (left angled bracket for disjoint union)#
rang                #» (right   "      "       "      "      )#
sr                  # start vertical rule#
er                  # end vertical rule#
ge                  # unique (generic) definition #

# Identifiers #

dlr                 #$#
for                 #/ (renaming)#
underline           #_ or underline (place holder for renaming)#
inset               # infixed generic sets #
preset              # prefixed generic sets #
postset             # postfixed generic sets #
op                  # infix operator#
encop               # lhs of enclosed operator#
distinop            # lhs of distributed infix operator#
distpreop           # lhs of distributed prefix operator#
eop                 # delimiter of two part operators#
preop               # prefix operator#
postop              # postfix operator#
sconst              # numbers and such #

# Theorems #

turnstile           # ⊢ (theorem)#
th                  # start theorem#
eth                 # end theorem#

# Predicate Notation #
```

```
all                  #∀#
exi                  #∃ or exi#
exi1                 # unique ∃#
where1               # artificial where markers#
where2               # #
endwhere             # #
dot                  #. (such that)#
equals               #=#
member               #∈#
rel                  # relational operator#
iff                  #⇔#
implies              #⇒#
and                  #∧#
or                   #∨#
not                  #¬#

# Term notation - for sets and objects #

set                  #{ or setbra#
eset                 #}#
explicit_set         # used to one-track explicit sets and
                       comprehensions#
lambda               #λ#
mu                   #μ#
lseq                 #⟨#
rseq                 #⟩#
proj                 #. (projection)#
theta                #θ tuple constructor#
prod                 #× cartesian product#
powerset             #P or powerset#
nat                  #Z#
char                 # Char#

# Schema notation #

zexi                 #∃#
zall                 #∀#
ziff                 #⇔#
zimplies             #⇒#
zand                 #∧#
zor                  #∨#
znot                 #¬#
zhide                #\#
pre                  #  pre#
zcmp                 #; (bold ;) schema composition #
zpipe                #> piping operator #
zovr                 #● (bold ●) schema over-ride #
sch                  #[ (start schema bracket)#
esch                 #] (end schema bracket)#
sb                   # start schema box (after name)#
st                   # middle line of schema box#
esb                  # end schema box#

RULES

z_text =
   finish <return-1>,
   z_phrase finish <return-1>,
   z_phrase z_sep z_text;

z_sep = list_sep, endz;

list_sep =
   semi,
   nl;
```

```
z_phrase = <store_mon-mon> zphrase1;

zphrase1 =
   given_set_def,
   definition,
   constraint,
   theorem,
   import,
   export;

# Given Set Declaration #

given_set_def = lsqb given_ids rsqb;

   given_ids = id <given_set_def> given_ids1;

      given_ids1 = $, comma given_ids;

# Definition #

definition =
   axiomatic_def,
   syntactic_def,
   datatype_def,
   schema_def;

# Global Constraint #

constraint = pred;

# Theorems #

theorem =
   turnstile pred <unstack_pred-ptype>,
   th th1 turnstile pred_list <unstack_pred-ptype><end_scope> eth;

#don't understand sb and eb at this point in Oxford syntax #

   th1 =
      <new_scope>,
      gen_params,
      <new_scope> hyps,
      gen_params hyps;

# a scope for the declarations in the theorem is always created,
  even if there aren't any.  If there are any generic parameters,
  the scope created for that is used, otherwise one is explicitly
  created.
#

hyps =
   hyp,
   hyp list_sep hyps;

# NB schema_term omitted because of ambiguities with schema
  reference in pred in hyp below
#

hyp =
   pred <check_pred_schema-ptype>,
   dec,
   dec cbar pred <unstack_pred-ptype>;

# pred on its own includes schema_reference #

# Import #
```

```
import = document <newdoc--docmap> import1 <adddoc-pdocmap>;

   import1 =
      $,
      decor <decdoc-pdocmap--docmap>,
      instantiation <instdoc-pinst-pdocmap--docmap>,
      decor <decdoc-pdocmap--docmap>
      instantiation <instdoc-pinst-pdocmap--docmap>;

# Export #

export = id <keep> keep idslist <return-2>;

   ids = id, inset, preset, postset, op, rel, encop eop, distinop eop,
         distpreop eop, preop, postop;

   idslist = ids <keep_id> idslist1;

      idslist1 = $, comma idslist;


# identifiers, names and references #

reference =
   id <reference--type> ref2,
   id dlr <check_no_att> id <doc_reference--type> ref2;

   ref2 =
      <anon_inst-ptype--type>,
      instantiation <id_inst-ptype-pinst--type>;

instantiation = ilsqb inst_list rsqb;

inst_list =  inst_term_list, binding_list, rename_list;
#gathered together to resolve various one-track problems#

inst_term_list =
   term <tml1-ptype--inst>,
   term <tml1-ptype--inst> comma inst_termlist1;

   inst_termlist1 =
      term <tml2-ptype-pinst--inst>
      inst_termlist2;

      inst_termlist2 = $, comma inst_termlist1;

binding_list = bl1, bl1 comma binding_list1;

   bl1 =
      id equals <bl1--id> term <bl2-pid-ptype--inst>;

   binding_list1 =
      bl2, bl2 comma binding_list1;

      bl2 =
         id equals <bl1--id>
         term <bl3-pid-ptype-pinst--inst>;

rename_list =
   id for id <rnl1--inst>,
   id for id <rnl1--inst> comma rename_list1;

   rename_list1 =
      id for id <rnl2-pinst--inst> rename_list2;

      rename_list2 = $, comma rename_list1;
```

```
* Axiomatic definition *

axiomatic_def =
   liberal_def,
   unique_def,
   generic_def;

liberal_def =
   global_dec,
   global_dec cbar pred <unstack_pred-ptype>,
   sr def_body er;

def_body =
   global_dec_list,
   global_dec_list st pred_list <unstack_pred-ptype>;

unique_def = ge def_body er;

generic_def =
   global_id <start_idlist--idlist> gen_params <new_scope>
   colon term <dec_ids-pidlist-ptype>
   cbar pred <unstack_pred-ptype><end_gen_def>,
   ge gen_params <new_scope>  def_body er <end_gen_def>;

gen_params = lsqb <new_scope> given_ids rsqb;

global_dec_list =
   global_dec,
   global_dec list_sep global_dec_list;

global_dec =
   global_id_list colon term <dec_ids-pidlist-ptype>;

global_id_list =
   global_id <start_idlist--idlist>,
   global_id <start_idlist--idlist> comma global_id_list1;

   global_id_list1 =
      $,
      global_id <stack_idlist-pidlist--idlist>,
      global_id <stack_idlist-pidlist--idlist> comma global_id_list1;


global_id =
   id,
   id underline <global_sym-1>,
   underline id <global_sym-2>,
   id underline id <global_sym-5>,
   lpar underline id underline rpar <global_sym-3>,
   underline id underline <global_sym-4>,
   id underline id underline <global_sym-6>,
   underline id underline id <global_sym-7>;

* Syntactic definition *

syntactic_def =
   syn_def_id,
   global_id <start_idlist--idlist> gen_params <new_scope>
   def term <syn_def-pidlist-ptype> <end_gen_def>,
   ge gen_params <new_scope> syn_def_list er <end_gen_def>;

syn_def_id =
   global_id <start_idlist--idlist> def term <syn_def-pidlist-ptype>;

syn_def_list =
   syn_def,
```

```
   syn_def list_sep syn_def_list;

syn_def =
   syn_def_id,
   syn_def_ids;

syn_def_ids =
   id id <prepostsymbol--idlist> def term <syn_def-pidlist-ptype>,
   id id id <insetsymbol--idlist>
      def term <syn_def-pidlist-ptype>;

# various sorts of pre and post generic set definition, depending
  on whether the ids occur in the generic parameters or not #

# Data type definition #

datatype_def = id <dt_def--id> becomes branches <unstack_id-pid>;

branch =
   id <dt_constant-qid>,
   id <dt_const_id--id> lang term rang <dt_constructor-ptype-pid-qid>;

branches = branch, branch bbar branches;

# Schema definition #

schema_def =
   id <start_idlist--idlist>
   schema_definition <syn_def-pidlist-ptype>,
   id <start_idlist--idlist> gen_params
   schema_definition <end_scope><syn_def-pidlist-ptype>;

schema_definition =
   <check_id_schema-pidlist--idlist> sdef schema_term,
   <check_id_schema-pidlist--idlist> schema;

# only boxed forms should really be allowed, but this is not checked#

# Schemas #

schema =
   sb <new_scope> dec_list <scope_to_schema_type--type> esb,
   sb <new_scope> dec_list
   st pred_list <unstack_pred-ptype><scope_to_schema_type--type> esb,
   sch <new_scope> dec_list <scope_to_schema_type--type> esch,
   sch <new_scope> dec_list
   cbar pred <unstack_pred-ptype><scope_to_schema_type--type> esch;

# Lists of predicates and declarations #

pred_list =
   pred,
   pred <unstack_pred-ptype> list_sep pred_list;

dec_list =
   dec,
   dec list_sep dec_list,
   inclusion,
   inclusion list_sep dec_list;

dec = id_list colon term <dec_ids-pidlist-ptype>;

id_list =
   id <start_idlist--idlist>,
   id <start_idlist--idlist> comma idlist1;
```

```
   idlist1 =
      id <stack_idlist-pidlist--idlist>,
      id <stack_idlist-pidlist--idlist> comma idlist1;

inclusion = reference <open_schema-ptype>;

# and check reference is to a schema_term #

# Explicit construction terms #

explicit_constr =
   tuple,
   explicit_set eset <empty_set--type>,
   explicit_set termlist1 eset <explicit_set-ptype--type>,
   lseq rseq <empty_list--type>,
   lseq termlist1 rseq <explicit_list-ptype--type>;

# explicit_set above is a pseudo terminal symbol inserted by a
look-ahead function to resolve the problem of disentangling {a, b, c}
from {a, b, c: T}. The look-ahead function looks ahead while
encountering id comma: if terminated by anything other than colon, the
explicit_set symbol is delivered instead of set. #

termlist1 = # terms of the same type #
   term,
   term comma termlist1a;

   termlist1a =
      term <check_tys_same-ptype-ptype--type>,
      term <check_tys_same-ptype-ptype--type> comma termlist1a;

tuple =
   lpar term <first_tuple-ptype--type> comma termlist2 rpar,

   theta reference <theta-ptype--type>;
   # the reference is to a schema #

   termlist2 = #terms for a tuple #
      term <next_tuple-ptype-ptype--type> termlist2a;

      termlist2a =
         $,
         comma termlist2;

# Closed terms #

aform =
   <nat--type> nat,
   <char--type> char,
   <sconst-lv--type> sconst,
   reference,
   aform proj id <proj-ptype--type>,
   lpar product rpar,
   explicit_constr,
   set <new_scope> dec_list comp_set eset,
   lpar partials rpar,
   encop term eop <funapp-ptype-ptype--type>,
   where1 <new_scope> ax_dec_list <unstack_pred-ptype>
   where2 pred_list <end_scope><check_pred-ptype--type> endwhere,
   where1 <new_scope> syn_def_list
   where2 pred_list <end_scope><check_pred-ptype--type> endwhere,
   lpar pred rpar;
   # allows bracketted predicates...#

   product =
      term <first_prod-ptype--type> prod
```

```
      product1 <end_prod-ptype--type>;

      product1 =
         term <next_prod-ptype-ptype--type>,
         term <next_prod-ptype-ptype--type> prod product1;

   comp_set =
      <scope_to_tuple--type>,
      cbar pred <unstack_pred-ptype><scope_to_tuple--type>,
      dot term <set-ptype--type>,
      cbar pred <unstack_pred-ptype> dot term <set-ptype--type>;

   ax_dec_list =
      dec_list cbar pred,
      sr dec_list st pred_list er;
      # sr and er because I like it that way...#

partials =
   underline rel underline <partrel--type>,
   underline op <funop--type> form2 <partop1-ptype-ptype--type>,
   aform op underline <partop2-ptype--type>,
   underline op underline <funop--type>,
   underline distinop <funop--type>
      term eop <partop1-ptype-ptype--type>,
   aform distinop underline eop <partop2-ptype--type>,
   underline distinop underline eop <funop--type>,
   distpreop underline eop underline <funop--type>,
   distpreop <funop--type>
      term eop underline <partop1-ptype-ptype--type>,
   distpreop underline eop <funop--type>
      form3 <partop1-ptype-ptype--type>,
   encop underline eop <funop--type>,
   preop underline <funop--type>,
   underline postop <funop--type>;

# Formulae #

formula =
   form1 inset <setop--id> formula <inset-pid-ptype-ptype--type>,
   form1;

form1 =
   form1 op <funop--type> form2 <infix-ptype-ptype-ptype--type>,
   form2;

form2 =
   form2 form3 <funapp-ptype-ptype--type>,
   form3;

form3 =
   preop <funop--type> form3 <funapp-ptype-ptype--type>,
   preset <setop--id> form3 <set_inst1-pid-ptype--type>,
   distpreop <funop--type>
      term eop form3 <distpreop-ptype-ptype-ptype--type>,
   powerset form3 <powerset-ptype--type>,
   form4;

form4 =
   form4 distinop <funop--type>
      term eop <infix-ptype-ptype-ptype--type>,
   form4 postop <postapp-ptype--type>,
   form4 postset <set_inst2-ptype--type>,
   aform;

# Comprehension terms #
```

```
comprehension =
   lambda <new_lambda_scope> dec_list lambda_set <lambda-ptype--type>,
   mu <new_scope> dec_list lambda_set <end_scope>;

   lambda_set =
      dot term,
      cbar pred <unstack_pred-ptype> dot term;

# Terms #

term = comprehension,
       formula;

# Atomic predicates #

apred =
   si pred_list ei,
   term;

# term includes term term (set membership), schema reference and
  bracketted predicate
#

# Relations #

rel_exp =
   term member term <member-ptype-ptype--type>,
   term equals equals_tail <to_pred-ptype--type>,
   term rel <funop--type> rel_tail <to_pred-ptype--type>,
   apred;

   equals_tail =
      term <equals-ptype-ptype--type> tail;

      tail =
         $,
         rel <funop--type> rel_tail,
         equals equals_tail;

   rel_tail =
      term <rel-ptype-ptype-ptype--type> tail;


# Logical expressions #

log_exp =
   log_exp1,
   log_exp <unstack_pred-ptype> iff log_exp1
   <check_pred-ptype--type>;

log_exp1 =
   log_exp2,
   log_exp1 <unstack_pred-ptype> implies log_exp2
   <check_pred-ptype--type>;

log_exp2 =
   log_exp3,
   log_exp2 <unstack_pred-ptype> or log_exp3
   <check_pred-ptype--type>;

log_exp3 =
   log_exp4,
   log_exp3 <unstack_pred-ptype> and log_exp4
   <check_pred-ptype--type>;

log_exp4 =
```

```
   rel_exp,
   not log_exp4 <check_pred-ptype--type>;

# Quantified expressions #

quant_exp =
   quant <new_scope> dec_list
   dot pred <end_scope><check_pred-ptype--type>,
   quant <new_scope> dec_list cbar pred <unstack_pred-ptype>
   dot pred <end_scope><check_pred-ptype--type>;

   quant = exi, exil, all;

# Predicates  #

pred =
   quant_exp,
   log_exp;

# Schema terms #

schema_term =
   quant_sexp,
   log_sexp;

quant_sexp = squant <new_scope> dec_list
             dot schema_term <subtract_scope-ptype--type>;

squant = zexi, zall;

# Logical schema expressions #

log_sexp =
   log_sexp1,
   log_sexp ziff log_sexp1 <stype2-ptype-ptype--type>;

log_sexp1 =
   log_sexp2,
   log_sexp1 zimplies log_sexp2 <stype2-ptype-ptype--type>;

log_sexp2 =
   log_sexp3,
   log_sexp2 zor log_sexp3 <stype2-ptype-ptype--type>;

log_sexp3 =
   log_sexp4,
.  log_sexp3 zand log_sexp4 <stype2-ptype-ptype--type>;

log_sexp4 =
   spec_sexp,
   znot log_sexp4;

# Special-purpose schema expressions #

spec_sexp =
   spec_sexp zhide lpar id_list rpar <hideids-pidlist-ptype--type>,
   spec_sexp zhide reference <hideref-ptype-ptype--type>,
   spec_sexp zcmp spec_sexp1 <scompose-ptype-ptype--type>,
   spec_sexp zpipe spec_sexp1 <pipe-ptype-ptype--type>,
   spec_sexp zovr spec_sexp1 <soverride-ptype-ptype--type>,
   spec_sexp1;

spec_sexp1 =
   spec_sexp2,
   pre spec_sexp2 <pre-ptype--type>;
```

```
rename =
    lsqb rename_list rsqb <id_inst-ptype-pinst--type>,
    decor <decorate-ptype--type>;

spec_sexp2 =
    lpar schema_term rpar,
    lpar schema_term rpar rename,
    reference <check_schema-ptype--type>,
    schema;
```

DOCUMENT CONTROL SHEET

Overall security classification of sheet UNCLASSIFIED

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference	3. Agency Reference	4. Report Security Classification
	REPORT 87017		UNCLASSIFIED

5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location
778400	RSRE, SAINT ANDREWS ROAD, MALVERN, WORCS WR14 3PS
5a. Sponsoring Agency's Code (if known)	**6a. Sponsoring Agency (Contract Authority) Name and Location**

7. Title

REVIEW OF TYPE CHECKING AND SCOPE RULES OF THE SPECIFICATION LANGUAGE Z

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials	9(a) Author 2	9(b) Authors 3,4...	10. Date	pc. ref.
SENNETT C T			1987.11	84

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution statement

Descriptors (or keywords)

continue on separate piece of paper

Abstract

This report gives the detailed type checking and scope rules for the specification language Z in the form of an implementation specification for a type checking tool for Z, written in Z itself.

S80/48

END

DATE
FILMED

8 8