An Algorithm for Type-Checking Z

A Z Specification

J. N. Reed
J. E. Sinclair

Abstract

We present an outline of an algorithm for type-checking Z specifications and determining appropriate error messages. The algorithm understands an abstract syntax of Z as given by J.M. Spivey, and is similar to the one implemented in the Forsite prototype specification support environment. The outline presented here is intended to serve as a brief introductory overview to implementing a Z type checker, and to elucidate important and subtle details involved in type checking Z. We do not discuss user interface or performance issues such as display of error messages or representation of data structures. The outline is itself described in Z.

Oxford University Computing Laboratory
Programming Research Group
8–11 Keble Road
Oxford OX1 3QD
England

Contents

1 Introduction

We present an outline of an algorithm for type-checking Z specifications. The purpose of this outline is to specify an overview of the algorithm for those planning to implement a Z type checker. The specification describes the task of checking an abstract syntax tree of a Z specification for type correctness and determining appropriate error messages.

Over the past ten years or so, Z has matured and is now a popular language used for software specification. A number of software development groups who use Z have done so without automated tool support (other than editors equipped with Z symbols and simple cross referencing). While it might be argued that the standardisation required for any kind of tool support for Z would render the language inflexible and hinder its further development, it is widely believed that tools such as parsers and type checkers are invaluable aids in many cases.

The first automated tool set for Z to provide both syntax and type checking was the Forsite prototype developed in 1986. The Forsite project began in 1985 under Alvey sponsorship with four collaborators: Racal Research, Systems Designers, PRG Oxford, and Surrey University. The objective was to develop a specification support environment for Z which provided automated syntax checking, type checking, and proof assistance. The prototype, which did not offer any support for theorem proving but provided substantial syntax and type checking, was distributed to a small number of development groups for β evaluation. The general opinion from these groups was that while certain improvements (e.g., performance) were necessary, such a tool, particularly the type checker, was essential for most industrial developmental work. At the completion of the Forsite project in March 1989 a first attempt at an assistant for proof work had been made and the toolset had been upgraded to take account of users' comments. The environment is now available for commercial development.

A recent trend among groups using Z is to build their own customised editors, parsers, and type checkers. Aiding these efforts has been the distribution of BNF forms of a Z syntax definition [KSW88,SPI88b]. We would like to further ease the difficulty of implementing a type checker for Z by providing a specification of an algorithm for checking an abstract syntax.

A complete specification would by its nature be so extensive and detailed as to be extremely difficult to understand as a first introduction to the problem. Thus we have decided to produce an overview, rather than a complete specification. We have developed this specification with *folding* in mind - the outline given here is the top layer, with further details folded below. We hope to impart a clear description of the operational model for the algorithm. The details which are omitted are straightforward variations of those which are discussed.

The style of presentation is functional, rather than state-based. The functional style is natural for translating the type semantics given by J.M. Spivey [SPI88a]. Indeed, many of the functions are "transliterations" of the corresponding definitions in [SPI88a]. The specification presented here describes an abstract operational model, directly implementable with a functional language but suitable for implementation with an imperative language.

We first present some notational conventions, followed by a definition of the abstract syntax understood by the algorithm. Section 4 describes the way in which errors are reported, and Section 5 introduces the idea of types and signatures. Section 6 describes *environments*, which serve as symbol tables. Section 7 discusses the concept of normalised declarations, the driving force for Z type checking. Sections 8-11 give descriptions of checking various groups of syntactic constructs, culminating in a function which checks a complete abstract syntax tree for a Z document. Included is a description of a unification algorithm used to infer types of expressions containing implicitly instantiated generic objects. Finally we present a brief comparison with similar work and general conclusions.

2 Notation

The notation that we use is conventional Z with certain syntactic differences, introduced simply as a shorthand. These can be easily (automatically) textually expanded into conventional Z.

Many of the functions we define are conditional in that their value depends on a boolean expression. We use the following more recognisable form:

$$\begin{aligned} value = \ & \textbf{if}\ \beta \\ & \qquad \textbf{then}\ X \\ & \qquad \textbf{else}\ Y \end{aligned}$$

to mean

$$(\beta \ \wedge \ \mathit{value} = X)$$
$$\vee$$
$$(\neg\beta \ \wedge \ \mathit{value} = Y)$$

We use ellipses (. . .) within a declaration or definition to indicate that there would be additional information in the complete specification. While we freely omit information from the body (predicate) of definitions, we do give complete signatures of all referenced terms. We use the data type *seq X* so frequently that we adopt the convention that for arbitrary *X*, *Xs* is shorthand for *seq X*. For example, *SIGs* is defined to be *seq SIG*.

3 Abstract Syntax

Our algorithm operates on an abstract syntax tree whose structure accords with the following syntax description. This is, with minor differences, the abstract syntax proposed by J. M. Spivey.

$[\mathit{STRING}]$

$\mathit{word} == \mathit{STRING}$
$\mathit{decor} == \mathit{seq}\ \mathit{STRING}$

$\mathit{ident} ::= \mathit{Ident} \ll \mathit{word} \times \mathit{decor} \gg$

$\mathit{rename} ::= \mathit{Rename} \ll \mathit{ident} \times \mathit{decor} \gg$

$\mathit{decl} ::= \mathit{Decl} \ll (\mathit{seq}\ \mathit{ident}) \times \mathit{expr} \gg$
$\quad\quad | \ \mathit{Include} \ll \mathit{sdes} \gg$

$$
\begin{array}{rl}
\mathit{pred} ::= & \mathit{Equal} \langle\!\langle \mathit{expr} \times \mathit{expr} \rangle\!\rangle \\
\mid & \mathit{Member} \langle\!\langle \mathit{expr} \times \mathit{expr} \rangle\!\rangle \\
\mid & \mathit{Truth} \\
\mid & \mathit{Falsity} \\
\mid & \mathit{Not} \langle\!\langle \mathit{pred} \rangle\!\rangle \\
\mid & \mathit{And} \langle\!\langle \mathit{pred} \times \mathit{pred} \rangle\!\rangle \\
\mid & \mathit{Or} \langle\!\langle \mathit{pred} \times \mathit{pred} \rangle\!\rangle \\
\mid & \mathit{Implies} \langle\!\langle \mathit{pred} \times \mathit{pred} \rangle\!\rangle \\
\mid & \mathit{Exists} \langle\!\langle \mathit{schema} \times \mathit{pred} \rangle\!\rangle \\
\mid & \mathit{Exists}1 \langle\!\langle \mathit{schema} \times \mathit{pred} \rangle\!\rangle \\
\mid & \mathit{Forall} \langle\!\langle \mathit{schema} \times \mathit{pred} \rangle\!\rangle \\
\mid & \mathit{Spred} \langle\!\langle \mathit{sdes} \rangle\!\rangle
\end{array}
$$

$$
\mathit{schema} ::= \mathit{Schema} \langle\!\langle (\mathit{seq}_1\ \mathit{decl}) \times \mathit{pred} \rangle\!\rangle
$$

$$
\begin{array}{rl}
\mathit{optionalexpr} ::= & \mathit{Justexpr} \langle\!\langle \mathit{expr} \rangle\!\rangle \\
\mid & \mathit{Noexpr}
\end{array}
$$

$$
\begin{array}{rl}
\mathit{expr} ::= & \mathit{Ref} \langle\!\langle \mathit{ident} \times \mathit{seq}\ \mathit{expr} \rangle\!\rangle \\
\mid & \mathit{Number} \langle\!\langle \mathit{word} \rangle\!\rangle \\
\mid & \mathit{Sexpr} \langle\!\langle \mathit{sdes} \rangle\!\rangle \\
\mid & \mathit{Ext} \langle\!\langle \mathit{seq}\ \mathit{expr} \rangle\!\rangle \\
\mid & \mathit{Comp} \langle\!\langle \mathit{schema} \times \mathit{optionalexpr} \rangle\!\rangle \\
\mid & \mathit{Power} \langle\!\langle \mathit{expr} \rangle\!\rangle \\
\mid & \mathit{Tuple} \langle\!\langle \mathit{seq}\ \mathit{expr} \rangle\!\rangle \\
\mid & \mathit{Seq} \langle\!\langle \mathit{seq}\ \mathit{expr} \rangle\!\rangle \\
\mid & \mathit{Product} \langle\!\langle \mathit{seq}\ \mathit{expr} \rangle\!\rangle \\
\mid & \mathit{Theta} \langle\!\langle \mathit{word} \times \mathit{decor} \rangle\!\rangle \\
\mid & \mathit{Select} \langle\!\langle \mathit{expr} \times \mathit{ident} \rangle\!\rangle \\
\mid & \mathit{Apply} \langle\!\langle \mathit{expr} \times \mathit{expr} \rangle\!\rangle \\
\mid & \mathit{Lambda} \langle\!\langle \mathit{schema} \times \mathit{expr} \rangle\!\rangle \\
\mid & \mathit{Mu} \langle\!\langle \mathit{schema} \times \mathit{optionalexpr} \rangle\!\rangle
\end{array}
$$

$$
\mathit{sdes} ::= \mathit{Sdes} \langle\!\langle \mathit{word} \times \mathit{decor} \times (\mathit{seq}\ \mathit{expr}) \times (\mathit{seq}\ \mathit{rename}) \rangle\!\rangle
$$

$$\begin{aligned}
sexp ::= {} & Text \ll schema \gg \\
| {} & Sref \ll sdes \gg \\
| {} & Snot \ll sexp \gg \\
| {} & Spre \ll sexp \gg \\
| {} & Sand \ll sexp \times sexp \gg \\
| {} & Sor \ll sexp \times sexp \gg \\
| {} & Simplies \ll sexp \times sexp \gg \\
| {} & Sequiv \ll sexp \times sexp \gg \\
| {} & Project \ll sexp \times sexp \gg \\
| {} & Hide \ll sexp \times seq\ sexp \gg \\
| {} & Fatsemi \ll sexp \times sexp \gg \\
| {} & Sexists \ll schema \times sexp \gg \\
| {} & Sforall \ll schema \times sexp \gg
\end{aligned}$$

$$arm ::= Arm \ll ident \times optionalexpr \gg$$

$$lhs ::= Lhs \ll ident \times seq\ ident \gg$$

$$\begin{aligned}
para ::= {} & Given \ll seq\ ident \gg \\
| {} & Let \ll schema \gg \\
| {} & Sdef \ll word \times (seq\ ident) \times sexp \gg \\
| {} & Pred \ll pred \gg \\
| {} & Define \ll (seq\ ident) \times schema \gg \\
| {} & Eqeq \ll lhs \times expr \gg \\
| {} & Data \ll seq\ (ident \times seq\ arm) \gg \\
| {} & Theorem \ll (seq\ ident) \times (seq\ expr) \times pred \gg
\end{aligned}$$

$$spec == seq\ para$$

4 Exceptions and Error Messages

Intuitively, we would like to construct functions which, when dealing with correct Z will calculate some value, but otherwise supply an appropriate error message. We make the following generic definition to capture this:

$$Result[X] == X \times ERROR$$

When an error is encountered, a default value of the correct type is supplied with the error message. This allows type checking to continue whilst trapping and dealing with errors in an appropriate way. The error reports used in this paper are described by the following data type:

$$\begin{array}{llllll} ERROR ::= & idnotfound & \mid notvalidschema & \mid notpowertype & \mid badsubstitution \\ \mid & idnotdeclared & \mid clear & \mid badunification & \mid badapplication \\ \mid & typevarsinpred & & & \end{array}$$

Use of the *Result* mechanism is described further in section 7.

5 Types and Signatures

We model the concept of a Z *TYPE* with the following datatype:

$$\begin{array}{ll} TYPE ::= identty \langle\!\langle ident \rangle\!\rangle & \text{– given set type} \\ \quad \mid powerty \langle\!\langle TYPE \rangle\!\rangle & \text{– power type} \\ \quad \mid productty \langle\!\langle TYPEs \rangle\!\rangle & \text{– cartesian product type} \\ \quad \mid schematy \langle\!\langle ident \twoheadrightarrow TYPE) \rangle\!\rangle & \text{– schema type} \\ \quad \mid unity & \text{– error type} \end{array}$$

We can think of proper Z types for a specification as its given sets, power sets of types, cartesian products of types, and "schema bindings" between identifiers and types. We introduce the *unity* type as a "univeral error type" for expressions which cannot be assigned a proper type because of errors in the specification. This type is useful for reducing cascading of error messages.

User-defined data types are not included as they may be viewed as derivable from other Z constructions (see [SPI88b]).

A major task of the type checker is to calculate a *signature* for each identifier (including generic ones), which associates the identifier (and its generic parameters) with its type:

$$GENTYPE == idents \times TYPE$$

$$SIG == ident \times TYPE$$

$$GENSIG == ident \times (seq\ ident) \times TYPE$$

Signatures make up *environments*, which are used to determine the variables which are in scope for a given expression.

6 Dictionaries and Environments

The type of an expression in a specification depends on the definitions in scope for that expression. Visibility of Z definitions is modeled using *environments* which play the role of symbol tables. Environments contain signatures which are grouped into dictionaries. A *dictionary* contains a list of *generic signatures*. Entries with *null* sequences of generic parameters represent nongeneric axiomatic definitions.

$$DICT == ident \text{⇻} GENTYPE$$

$$\begin{array}{|l}
nulldict : DICT \\
\hline
nulldict = \varnothing
\end{array}$$

The use of *Result* is illustrated below in the definition of *lookup*, which produces the generic parameters and type of any identifier stored in a given dictionary, and a default value plus error message for any not found.

$$\begin{array}{|l}
lookup : ident \rightarrow DICT \rightarrow Result[GENTYPE] \\
\hline
\forall\ id : ident;\ dict : DICT \bullet \\
\qquad lookup\ id\ dict = \\
\qquad\qquad \textbf{if}\ id \in \mathrm{dom}\ dict\ \textbf{then}\ (dict\ id, clear) \\
\qquad\qquad \textbf{else}\ ((<>, unity), idnotfound)
\end{array}$$

To project out the first element of a *Result* (that is, its type) we define the function *value*:

$$value == first\ [GENTYPE, ERROR]$$

Another useful dictionary operation is *addsig*, which adds a simple signature to a dictionary.

$$\begin{array}{|l}
addsig : SIG \rightarrow DICT \rightarrow DICT \\
\hline
\forall\ id : ident;\ t : TYPE;\ dict : DICT \bullet \\
\qquad addsig\ (i, t)\ d = d \oplus \{i \mapsto (<>, t)\}
\end{array}$$

An *environment* consists of an *axiomatic dictionary* and a *schema dictionary* (distinguishing between axiomatic and schema signatures is convenient but not necessary):

$$ENV == DICT \times DICT$$

Given sets are added to an *environment* simply by recording that their types are power sets of themselves:

$$\begin{array}{|l}
installgiven : ident \rightarrow ENV \rightarrow ENV \\
\hline
\forall\ givenset : ident;\ axdct, schemadct : DICT \bullet \\
\quad installgiven\ givenset\ (axdct, schemadct) = \\
\qquad (addsig(givenset, powerty(identty\ givenset))axdct, schemadct)
\end{array}$$

We name some other useful environment operations, omitting the full definitions:

$$\begin{array}{|l}
axdict : ENV \rightarrow DICT \\
schemadict : ENV \rightarrow DICT \\
installgenschemasig : ENV \rightarrow GENSIG \rightarrow ENV \\
installgivens : ENV \rightarrow \mathbb{P}\ ident \rightarrow ENV \\
installsigs : ENV \rightarrow \mathbb{P}\ SIG \rightarrow ENV \\
installgensigs : ENV \rightarrow \mathbb{P}\ GENSIG \rightarrow ENV \\
\ldots
\end{array}$$

The first two project the axiomatic and schema dictionaries respectively from an environment. The function *installgenschemasig* adds a generic signature to the schema dictionary. The others add collections of signatures or given sets to an environment and would be defined in much the same way as *installgiven*.

7 Normalised Declarations

The type checker must check each declaration with respect to its "current" environment, and update this environment accordingly. It does this by transforming, i.e., *normalising*, definitions from the Z specification into signatures which it adds to the environment. Normalising a simple declaration produces a signature which associates the declared variable with its type, and for a schema name used as a declaration produces a list of signatures which associate the schema variables with their types.

We first define some useful "pseudo inverse" functions. Intuitively, these functions behave as inverse functions provided that they are applied to values in the range of their counterpart, otherwise they "except" supplying an appropriate error message.

Often we need to extract the signature list from a schema type. Recall that *schematy* is a constructor for the *TYPE* datatype, and is therefore an injection with a functional inverse. We extend this inverse function to give a total function on *TYPE*:

$$\begin{array}{|l}
\mathit{invschematy} : \mathit{TYPE} \rightarrow \mathbb{P}\,\mathit{SIG} \\
\hline
\forall\, \mathit{ty} : \mathit{TYPE} \bullet \\
\qquad (\mathit{ty} \in \operatorname{ran} \mathit{schematy} \;\wedge\; \mathit{invschematy}\ \mathit{ty} = \mathit{schematy}^{-1}\ \mathit{ty}) \;\vee \\
\qquad (\mathit{ty} = \mathit{unity} \;\wedge\; \mathit{invschematy}\ \mathit{ty} = \varnothing) \;\vee \\
\qquad (\mathit{ty} \notin (\mathit{unity} \cup \operatorname{ran} \mathit{schematy}) \;\wedge\; \mathit{invschematy}\ \mathit{ty} = \mathit{value}(\varnothing, \mathit{notvalidschema}))
\end{array}$$

Important Notes :

1. When the type checker meets the default type *unity*, it is the case that an error has previously been discovered and an error message generated. To limit cascading of error messages initiated from a single error, no further error messages are supplied.

2. Notice that in the above function definition, since we extract only the *value* component from the *Result*, the error message is logically superflous. We have chosen to think of this function as returning values of its range type $\mathbb{P}\,SIG$ when things go smoothly, while excepting with a message together with a (default) value of the range type when things go wrong.We could think of the function, *value* as having some side-effect which deals with error messages in an appropriate way. To be fully formal, we could have such functions return the complete *Result*, rather than just the *value*. To simplify subsequent references to *invschematy* , we have chosen to express it as having range type that of the *value*, i.e., $\mathbb{P}\,SIG$. The logically superflous information contained in the above predicate is intended to guide the implementor. We use this convention for all functions which possibly generate error messages.

Very similar to *invschematy* is the function *invpowerty*, which strips off the $\mathbb{P}$ from a *powerty*, and excepts with *unity* and the error 'notpowertype':

$$
\begin{array}{|l}
invpowerty : TYPE \rightarrow TYPE \\
\hline
\forall\, ty : TYPE \bullet \\
\quad (ty \in \operatorname{ran} powerty \;\wedge\; invpowerty\ ty = powerty^{-1}\ ty) \;\vee \\
\quad (ty = unity \;\wedge\; invpowerty\ ty = unity \;\vee \\
\quad (ty \notin (unity \cup \operatorname{ran} powerty) \;\wedge\; invpowerty\ ty = value(unity, notpowertype))
\end{array}
$$

For expressions representing sets, it is useful to compnte the type which contains that set as a subset, e.g., intuitively, $\{x : \mathbb{N} \mid x \geq 5\}$ is a subset of $\mathbb{N}$, given that $\mathbb{N}$ is a type. (Note that the type of this expression is $\mathbb{P}\,\mathbb{N}$.) To compute this "superset" type, all that needs doing is to find the type of the expression and then strip off the "$\mathbb{P}$" :

$$
\begin{array}{|l}
supertype : ENV \rightarrow expr \rightarrow TYPE \\
\hline
\forall\, env : ENV;\; exp : expr \bullet \\
\quad supertype\ env\ exp = invpowerty\,(typeof\ env\ cxp)
\end{array}
$$

Note that the expression on the right might "except", producing an error message and returning *unity* as the *supertype*. This is as intended, but also as intended we need not concern ourselves in this function with the resulting error message - rather we proceed as if a proper type was calculated.

Important Note - The function *typeof* above, yet to be defined, calculates the type of an expression with respect to the current *environment*. In a more complete presentation of this algorithm, we would combine *supertype*, *typeof*, and various other function definitions into one mutually recursive set of axiomatic definitions. So that we can individually explain each definition, we present them here as separate definitions. The type of the function *typeof* is the same as that of *supertype*.

Simple declarations

To normalise a simple declaration consisting of "**variables : expression**" such as
"**a,b : X**", the type checker simply builds signatures associating each variable with the *supertype* of the expressiou on the right of the colon:

$$
\begin{array}{|l}
\mathit{normDecl} : \mathit{ENV} \rightarrow (\operatorname{ran} \mathit{Decl}) \rightarrow \mathbb{P}\,\mathit{SIG} \\
\hline
\forall\, \mathit{env} : \mathit{ENV};\ \mathit{idlist} : \mathit{idents};\ \mathit{exp} : \mathit{expr} \bullet \\
\qquad \mathit{normDecl}\ \mathit{env}\ (\mathit{Decl}(\mathit{idlist}, \mathit{exp})) = \\
\qquad\qquad \operatorname{ran}(\mathit{map}(\lambda\, \mathit{id} : \mathit{ident} \bullet (\mathit{id}, \mathit{supertype}\ \mathit{env}\ \mathit{exp})\mathit{idlist}))
\end{array}
$$

where *map* is the usual function used in functional programming which may be defined:

$$
\begin{array}{|l}
\hline
[I, X, Y] \\
\mathit{map} : (X \rightarrow Y) \rightarrow (I \twoheadrightarrow X) \rightarrow (I \twoheadrightarrow Y) \\
\hline
\forall\, \mathit{fn} : X \rightarrow Y;\ \mathit{sx} : I \twoheadrightarrow X;\ i : I;\ y : Y \bullet \\
\qquad (i, y) \in (\mathit{map}\ \mathit{fn}\ \mathit{sx}) \Leftrightarrow (\exists\, x : X \bullet (i, x) \in \mathit{sx} \wedge \mathit{fn}\ x = y) \\
\hline
\end{array}
$$

Named schemas

Normalising a declaration consisting of a named schema with possibly some actual parameters requires "unraveling" the schema to its normalised component signatures, and then instantiating the generic parameters with the actuals. We first define functions for the instantiation, which will also be useful for instantiating generic function applications, both implied as well as explicit.

The function *instant* takes a list of substitutions, each indicating that a generic parameter should be replaced by an actual type, together with a type, and returns this type with the indicated substitutions. For example, *instant* with the substitution $\{(\alpha, \mathbb{P}\,\mathbb{N}), (\beta, \mathbb{N})\}$ and target type $(\alpha \times \beta)$ yields $(\mathbb{P}\,\mathbb{N}) \times \mathbb{N}$. The definition of *instant* is recursive on the structure of the target type. The base cases are: *(i).* the null substitution, in which case the target type is left unchanged, *(ii).* the target type is unity - unity is returned, and *(iii).* a simple identifier target type, in which case the target is replaced by the indicated substitution if the identifier appears in the substitution list, or left unchanged if the identifier does nor appear in the list.

$GENPARAM == ident$

$$
\begin{array}{|l}
instant : (GENPARAM \mathrel{⇸} TYPE) \rightarrow TYPE \rightarrow TYPE \\
\hline
\forall subst : GENPARAM \mathrel{⇸} TYPE;\ targetty : TYPE; \\
\qquad\qquad targetid : ident;\ tys : TYPEs;\ binding : ident \mathrel{⇸} TYPE \bullet \\
\quad instant\ \{\}\ targetty = targetty \\
\land \\
\quad instant\ subst\ (identty\ targetid) = \\
\qquad \mathbf{if}\ targetid \in \operatorname{dom} subst\ \mathbf{then}\ subst\ targetid\ \mathbf{else}\ identty\ targetid \\
\land \\
\quad instant\ subst\ (powerty\ targetty) = powerty\ (instant\ subst\ targetty) \\
\land \\
\quad instant\ subst\ (productty\ tys) = productty\ (map\ (instant\ subst)\ tys) \\
\land \\
\quad instant\ subst\ (schematy\ binding) = schematy\ (map\ (instant\ subst)\ binding) \\
\land \\
\quad instant\ subst\ unity = unity
\end{array}
$$

A declaration consisting of an included schema name may contain a decoration and some renames. The following two functions are useful for handling these. The function *decorvars* decorates all the variables of a signature list with a given decoration. The function *renamevars* renames all the variables within a schema type according to a given list of renames. (It excepts, leaving the schema type unchanged, if a new variable collides with an unchanged original, or if a variable to be replaced does not appear in the original schema type.) We omit the complete definitions for these two functions:

$$
\begin{array}{|l}
decorvars : \mathbb{P}\, SIG \rightarrow decor \rightarrow \mathbb{P}\, SIG \\
renamevars : renames \rightarrow TYPE \rightarrow TYPE
\end{array}
$$

The function *mksubst* constructs a substitution list of generic parameters paired with instantiations, to be used by *instant*. Actual parameters are expressions, but *mksubst* constructs the substitution list using their *supertypes*. For example, if the generic parameter is β and the actual parameter is $\{x : \mathbb{N} \mid x \geq 5\}$, we treat β as being instantiated with $\mathbb{N}$. The function excepts if the numbers of the generics and the actuals supplied to it are not equal, or if there is a repeat in the generic list.

```
$mksubst : ENV \rightarrow GENPARAMs \rightarrow exprs \rightarrow (GENPARAM \nrightarrow TYPE)$
----------
$\forall\, env : ENV;\ gens : GENPARAMs;\ actuals : exprs \bullet$
    $mksubst\ env\ gens\ actuals =$
        $\mathbf{if}\,(\#actuals = \#gens) \wedge (\#gens = \#\operatorname{ran} gens)$
        $\mathbf{then}\ \ gens^{-1}\ ;\ (map\,(supertyenv)\ actuals)$
        $\mathbf{else}\ \ value(\{\}, badsubstitution)$
```

We now explain how to normalise a declaration which is a *schema designator*, consisting of a name of a previously declared schema, together with optional decoration, actual parameters, and list of renames. The type checker must look up the undecorated schema name in the schema dictionary, decorate the variables in the stored signature with the decoration from the included schema name, rename the variables as indicated, make a substitution list associating the supertypes of the actual parameter expressions with the generic types in the schema signature, and finally, replace the generic types in the signature with their corresponding actual types. If the schema name is not located in the schema dictionary, the function excepts.

For example, suppose the schema S is :

```
$S[X, Y]$
$a : X;$
$b : \mathbb{P}\, Y$
----------
$\ldots$
```

and the type checker encounters the declaration:

$$S'[\{1, 2\}, \mathbb{P}\,\mathbb{N}][s \backslash a]$$

The following steps should be taken:

- Look up S in the schema dictionary. This should give the generic type, $(< X, Y >, schematy\{(a, X), (b, \mathbb{P}\ Y)\})$.
- Decorate the schema variables giving $\{(a', X), (b', \mathbb{P}\ Y)\}$.
- Rename the schema variables as directed making the schematype $\{(s, X), (b', \mathbb{P}\ Y)\}$.
- Make a substitution list: $X \mapsto \mathbb{N}$, $Y \mapsto \mathbb{P}\,\mathbb{N}$.

- Apply the substitution, yielding schematy $\{(s,\mathbb{N}),(b',\mathbb{P}\,\mathbb{N})\}$

This is the task of the function *NormInclude*. It uses a function:

$$assignvars : GENTYPE \rightarrow TYPE$$

to assign type variables to any generic parameters which have not been instantiated. This is described further in the section on unification.

$$
\begin{array}{l}
normInclude : ENV \rightarrow (\mathrm{ran}\ Include) \rightarrow \mathbb{P}\,SIG \\
\hline
\forall\, env : ENV : wd : word;\ dcr : decor : actuals : exprs;\ newnames : renames\ \bullet \\
\quad (normInclude\ env\ Include(Sdes\ (wd\ dcr\ actuals\ newnames)) = \\
\qquad \mathbf{if}\ (exception \neq clear)\ \mathbf{then}\ value(\emptyset, schemanotdeclared)) \\
\qquad \mathbf{else}\ \ assignvars\ (gens, invschematy\ (instant\ subst\ renamedty)) \\
\quad \mathbf{where} \\
\qquad\quad gens : GENPARAMs;\ genty, decoratedty, renamedty\ :\ TYPE; \\
\qquad\quad exception : ERROR;\ subst : seq\ (GENPARAM \times TYPE) \\
\qquad \mid\ ((gens, genty), exception) = (lookup\ (ident(wd, <>))\ (schemadict\ env)) \\
\qquad \wedge\ decorty = schematy(decorvars\ (invschematy\ genty)\ dcr) \\
\qquad \wedge\ renamedty = renamevars\ newnames\ decoratcdty \\
\qquad \wedge\ subst = mksubst\ env\ gens\ actuals \\
\quad)
\end{array}
$$

The function *normInclude* defined above is quite useful - we shall see it again when we deal with schemas as ordinary expressions.

We end this section by giving the function *normdecl*, which normalises an arbitrary declaration. From the previously defined functions, we see that for a simple declaration, *normdecl* produces a list associating each variable on the left with the supertype of the expression on the right, and for an included schema name it produces a list consisting of the unraveled schema component signatures, properly decorated, renamed and instantiated:

$$
\begin{array}{l}
normdecl : ENV \rightarrow decl \rightarrow \mathbb{P}\,SIG \\
\hline
\forall\, env : ENV;\ d : decl\ \bullet \\
\quad (d \in (\mathrm{ran}\ Decl)\ \wedge\ normdecl\ env\ d = normDecl\ env\ d) \\
\vee\ \ (d \in (\mathrm{ran}\ Include)\ \wedge\ normdecl\ env\ d = normInclude\ env\ d)
\end{array}
$$

8 Types of Expressions

Much of the work of the type checker consists of calculating the types of expressions with respect to a current environment. We examine in detail the most interesting and/or obstruse of these calculations - that of a simple reference to an identifier, a schema expression, a set comprehension, and a theta term. We devote the next section to describing the type of a function application and the unification involved in inferring implicit actual parameters for generic functions.

A reference to an identifier ($Ref \ll ident \times seq\ expr \gg$) consists of its name and a possibly empty list of actual parameters. The type of the identifier is the type found in the axiomatic dictionary for the name, with the generic parameters replaced by the supertypes of the actual parameters. The function *typeofRef* given below calculates the type of a reference to an *ident*, with a sequence of *expr*, possibly empty, as actual parameters. (If the actual parameters are not explicitly given, the type returned is the original generic type for the identifier. This is sorted out elsewhere by the type checker. As the *Ref* will be part of a larger syntactic structure type inference may be possible.) The function excepts if the identifier is not found in the axiomatic dictionary of the current environment. Exceptions may also be generated by *mksubst* if the actual and generic parameters are not consistent.

```
| $typeofRef : ENV \rightarrow ident \rightarrow exprs \rightarrow TYPE$
|------
| $\forall env : ENV;\ id : ident;\ exps : exprs \bullet$
|     $typeofRef\ env\ id\ exps =$
|         if $exception \neq clear$ then $value(unity, idnotdeclared)$
|         else $assignvars\ (gens, instant\ (mksubst\ env\ gens\ exps)\ gentype)$
|       where
|           $gens : GENPARAMs;\ gentype : TYPE;\ exception : ERROR$
|         $\mid ((genparams, gentype), exception) = lookup\ id\ (axdict\ env)$
```

A schema designator may appear as an expression ($Sexpr \ll sdes \gg$). For example, if **TABLE** were defined as a (generic) schema, then its occurrence in the declaration, **tab: TABLE[SYMBOL]** is as a schema designator with actual parameter **SYMBOL**. The type of such a schema expression is intuitively the powerset of the schema type of the normalised signatures of the designated schema components, properly instantiated, renamed and decorated. The function *normInclude*, defined previously, handles these matters. Thus *typeofSexp* defined below is nicely concise:

$$\begin{array}{|l}
typeofSexp : ENV \rightarrow sdes \rightarrow TYPE \\
\hline
\forall env : ENV;\ sd : sdes \bullet \\
\quad typeofSexp\ env\ sd = \\
\quad\quad powerty\ (schematy\ (normInclude\ env\ (Include\ sd)))
\end{array}$$

In order to describe how to calculate the type of a set comprehension expression
Comp $\ll$ *schema* $\times$ *optionalexpr* $\gg$, we must first introduce the notion of adapting an environment with new declarations. The function ***adapt*** updates an environment by installing normalised signatures of declarations. It uses ***installsigs*** (a function introduced in section 6) to add the generalised union of the normalised signatures of declarations to an environment:

$$\begin{array}{|l}
adapt : ENV \rightarrow decls \rightarrow ENV \\
\hline
\forall env : ENV;\ decs : decls \bullet \\
\quad adapt\ env\ decs \\
\quad\quad installsigs\ env\ \mathrm{ran}(\bigcup(map\ (normdecl\ env)\ decs))
\end{array}$$

The type of a set comprehension expression is the powerset of the type of its "defining term" with respect to the current environment adapted with the contained declarations. If the defining term is not explicitly given, it is taken to be the charactaristic tuple of the variables in the declaration. The way in which this characteristic tuple is built is described in [SPI88b].We do not give the full definition of a tuple-building function here. However, its declaration is:

$$\begin{array}{|l}
mkchartuple : decls \rightarrow expr
\end{array}$$

Since the function *mkchartuple* can be used to make the defining term of a comprehension, when determining the type of a set comprehension we need only consider the case where the optional expression is present. This is done by the function *typeofComp*:

$$
\begin{array}{|l}
\mathit{typeofComp} : \mathit{ENV} \rightarrow (\mathit{schema} \times \mathit{expr}) \rightarrow \mathit{TYPE} \\
\hline
\forall\, \mathit{env} : \mathit{ENV};\ \mathit{decs} : \mathit{decls} : \mathit{pre} : \mathit{pred};\ \mathit{exp} : \mathit{expr} \bullet \\
\quad (\ \mathit{typeofComp}\ \mathit{env}\ (\mathit{Schema}(\mathit{decs}, \mathit{pre}),\ \mathit{exp}) = \mathit{powerty}\ (\mathit{typeof}\ \mathit{newenv}\ \mathit{exp}) \\
\quad\ \wedge\ (\mathit{pre}, \mathit{newenv}) \in \mathit{groundpreds} \\
\quad \textbf{where} \\
\qquad \mathit{newenv} : \mathit{ENV}\ |\ \mathit{newenv} = \mathit{adapt}\ \mathit{env}\ \mathit{decs} \\
\quad)
\end{array}
$$

Important Note - The set *groundpred*, discussed in section 10, describes a notion of predicates being type correct with respect to an environment. We just note here that the predicate above about *groundpreds* is "universally true", but error messages may be generated through its "side effects".

A theta expression ($\mathit{Theta} \ll \mathit{word} \times \mathit{decor} \gg$) consists of a name and a decoration. To determine the type of a theta expression, the type checker retrieves the variables from the schema type stored in the schema dictionary for the undecorated schema name, decorates these variables with the given decoration, determines the types of these decorated variables for the current environment (hence these decorated variables must be in scope), forms normalised signatures associating the undecorated variables with the types of their decorated version, and finally, returns the schema type over these normalised signatures.

This has the consequence, described in [SPI88b], that the types come from the current environment and *not* from the schema. So, for a schema S, there is no guarantee that $\theta S \in S$. For a fuller description, see [SPI88b].

Decorating a single variable is straightforward:

$$
\begin{array}{|l}
\mathit{decvar} : \mathit{decor} \rightarrow \mathit{ident} \rightarrow \mathit{ident} \\
\hline
\forall\, \mathit{olddcr}, \mathit{newdcr} : \mathit{decor};\ \mathit{name} : \mathit{word} \bullet \\
\quad \mathit{decvar}\ \mathit{newdcr}\ (\mathit{Ident}(\mathit{name}, \mathit{olddcr})) = \mathit{Ident}(\mathit{name}, \mathit{olddcr} \frown \mathit{newder})
\end{array}
$$

Retrieving the variables from the undecorated schema name can be achieved by treating the schema name as if it were an included declaration, and extracting them from the normalised signature. If the types of any of the decorated versions of the variables is *unity*, the function initiates an exception. Again we see *normInclude* used in the definition of *typeofTheta*:

```
typeofTheta : ENV → (word × decor) → TYPE
──────────────────
∀ env : ENV; name : word; dcr : decor •
    typeofTheta env (name, dcr) =
        if unity ∈ newsig then value(unity, notinscope)
        else schematy newsig
    where newsig : SIG; origvars : idents
        | origvars =
            dom(normInclude env Include(Sdes(name, <>, <>, <>)))
        ∧ newsig =
            (λ var : ident • (var, typeof env (decvar dcr var))) ⦇origvars⦈
    )
```

Moving now from specific kinds of expressions to expressions in general, the function *typeof* defined below produces the type of any arbitrary expression with respect to a given environment. Note that the function *typeof* would be mutually recursive with *groundpreds* and *superty*:

```
typeof : ENV → expr → TYPE
──────────────────
...
    ∧ typeof env (Ref(id, actuals)) = typeofRef env id actuals
    ∧ typeof env (Sexp(sd)) = typeofSexp env sd
    ∧ typeof env (Comp(schematerm, opexp)) =
            typeofComp env (schematerm, opexp)
    ∧ typeof env (Theta(name, dcr)) = typeofTheta env (name, dcr)
...
```

This may be extended to cover all other Z expressions in the same manner as those described here.

9 Schema Expressions

A schema expression (*sexp*) consists of either a set of declarations together with a predicate (typically expressed with the box notation), or some "logical" combination from the schema calculus of schema expressions (e.g., S ∧ **T**). The type checker unravels such schema expressions, producing normalised signatures associating component variables with their types.

For a schema expression consisting of a set of declarations together with a predicate (*Text* $\ll$ *schema* $\gg$), the function *normText* normalises the declarations and checks the predicate with respect to the current environment adapted with the normalised declarations:

$$\begin{array}{|l}
normText : ENV \rightarrow schema \rightarrow \mathbb{P}\, SIG \\
\hline
\forall\, decs : decls;\ pre : pred \bullet \\
\qquad normText\ env\ (Schema(decs, pre)) = newsigs \\
\quad\ \land\ (pre, installsigs\ env\ newsigs) \in groundpred \\
\textbf{where}\ newsigs : SIGS \mid newsigs = map\ normdecl\ decs
\end{array}$$

A schema expression can reference another schema (for example, the right hand side of $T = S[a\backslash x;\ b\backslash y]$ is a reference to schema S with x and y renamed to a and b). Normalising such a schema expression (*Sref* $\ll$ *sdes* $\gg$) simply involves treating the reference as an included schema declaration:

$$\begin{array}{|l}
normSref : ENV \rightarrow sdes \rightarrow SIGS \\
\hline
\forall\, env : ENV;\ sd : sdes \bullet \\
\qquad normSref\ env\ sd = normInclude\ env\ sd
\end{array}$$

In order to normalise logical combinations of schema expressions the type checker simply groups together the component normalised signatures. Checking that the resulting list of signatures contains no collisions is left to the function *normsexp*, which is mutually recursive with *normText*, *normSref*, and the other specific schema normalising functions (see below).

A representative logical combination of schema expressions is "schema and" (*Sand* $\ll$ *sexp* $\times$ *sexp* $\gg$):

$$\begin{array}{|l}
normSand : ENV \rightarrow (sexp \times sexp) \rightarrow \mathbb{P}\, SIG \\
\hline
\forall\, env : ENV;\ sexp_1, sexp_2 : sexp \bullet \\
\qquad normSand\ env\ sexp_1\ sexp_2 = \\
\qquad\qquad (normsexp\ env\ sexp_1) \cup (normsexp\ env\ sexp_2)
\end{array}$$

The definitions for the other schema expression normalising functions, which are all variants of the ones given above. The general function *normsexp* given below takes an arbitrary schema expression and produces a normalised signature list. It must ensure that there are no "collisions", i.e., redeclared variables in this list. The function *rmcollisions* checks a set of signatures for collisions, generating an appropriate error message for each one. Any variables with colliding signatures will be assigned the type *unity* to allow type checking to continue. We omit complete definitions for these functions:

$$rmcollisions : SIGs \rightarrow SIGs$$

$$normsexp : ENV \rightarrow sexp \rightarrow SIGs$$

$$\begin{array}{l}
\ldots \\
\quad \land\ normsexp\ (Text(schematext)) = rmcollisions(normText\ env\ schematext) \\
\quad \land\ normsexp\ env\ (Sref(schemades)) = rmcollisions(normSref\ env\ schemades) \\
\quad \land\ normsexp\ env\ (Sand(sexp_1, sexp_2)) = rmcollisions(normSand\ env\ (sexp_1, sexp_2)) \\
\ldots
\end{array}$$

Here, we do not give details of checking all schema operations. The general approach is the same, with the following guidelines. Schema quantification, hiding, projection and precondition all have the effect of hiding some of the components of their argument schemas. The components being hidden must occur in the argument schema and have the same type as in the schema. The type of the schematerm is the type of the original schema, but without the bindings of the hidden components.

The sequential composition, $S \fatsemi T$, is well-typed when the dashed variables of S match exactly the undashed variables of T. The resulting schema has a type consisting of the bindings of all the undashed variables of S and the dashed variables of T.

10 Unification

The type system of Z depends only on the signatures in the environment and not on any of the constraints, and it is therefore decidable[1]. However, because of the presence of generic definitions which may be used without explicit instantiation, type expressions may require unification to see whether terms are correctly typed. The process of unification takes two (possibly

[1]The following account of unification in Z and the unification algorithm itself are derived from the work of Mike Spivey. Thanks.

generic) types and discovers which (if any) instantiations of the generic paramaters would make these types match exactly. Z requires that we must be able to find exactly one such instantiation, and that no generic parameters be left uninstantiated.

For instance,

$$\emptyset = 3$$

is badly typed. The empty set is a generic constant having generic type $[X]\,\mathbb{P}\,X$. Since it is set-valued, there is no value for X that can unify the type of $\emptyset$ with $\mathbb{N}$, the type of the right-hand side. In this case, no instantiation would work.

Spivey[SPI88b] gives examples using the function *first* which gives the first of a pair of objects. *first* has generic type $[X, Y]\,\mathbb{P}((X \times Y) \times X)$. The expression:

$$\mathit{first}(\emptyset, 3) \in \mathbb{F}\,\mathbb{N}$$

is correctly typed since we can determine that *first* must be instantiated with $\mathbb{P}\,\mathbb{N}$ and $\mathbb{N}$, and the empty set with $\mathbb{N}$. However, the expression:

$$\mathit{first}(3, \emptyset) = 3$$

is incorrectly typed. This time the problem is that there are too many possible unifiers - the types of the occurances of $\emptyset$ and *first* cannot be uniquely determined. This situation can always be resolved by explicit instantiation of the unknown parameters.

The type checker must report an error if either (i) the types assigned so far indicate that a conflict has arisen, or (ii) at a time when all generic types should have been assigned actual parameters some of them remain uninstantiated. The time for deciding (ii) is when an "$=$" or "$\in$", or any relational operator is encountered. It is not possible to accrue type information over several expressions containing different occurances of some generic object. For instance, the expression:

$$(\mathit{first}(\emptyset, 3) = 3) \land (\emptyset \in \mathbb{F}\,\mathbb{N})$$

does not determine the type of *first*, or indeed of the first occurance of $\emptyset$. This is because the two occurances of the empty set are treated as separate instances and both must be completely instantiated.

Unification works by assigning type variables to generic types and trying to calculate a unique instantiation of these variables. We extend our definition of type to reflect this:

$TYPE ::= identty \ll ident \gg$	– given set type
$\mid\ powerty \ll TYPE \gg$	– power type
$\mid\ productty \ll TYPEs \gg$	– cartesian product type
$\mid\ schematy \ll ident \twoheadrightarrow TYPE) \gg$	– schema type
$\mid\ varty \ll name \gg$	– type variable
$\mid\ unity$	– error type

Example

As an example, consider how the following expression would be type checked:

$$\varnothing = \{S : \mathbb{P}\,\mathbb{N} \mid S = \varnothing \bullet \varnothing\}$$

As noted above, the three occurances of $\varnothing$ are all different and here we number them just to emphasize the point:

$$\varnothing_1 = \{S : \mathbb{P}\,\mathbb{N} \mid S = \varnothing_2 \bullet \varnothing_3\}$$

The generic type of $\varnothing$ is $[X]\ \mathbb{P}\,X$. For each occurance of the empty set we form its type using a type variable, taking care to use fresh variables each time. Choosing the type variable α we can say that the type of the left-hand side is $\mathbb{P}\alpha$:

$$\varnothing_1 : \mathbb{P}\,\alpha$$

We must unify this with the type of the set comprehension term. So now consider the term $\{S : \mathbb{P}\,\mathbb{N} \mid S = \varnothing_2 \bullet \varnothing_3\}$. To type check a set we add its declarations (in this case, $S : \mathbb{P}\,\mathbb{N}$) to the environment. In this extended environment we must check the predicate $S = \varnothing_2$, which involves unifying the type of S with the type of $\varnothing_2$. We will need a new type variable, β, with which to represent the type of the second occurance of the empty set. With this we know that:

$$S : \mathbb{P}\,\mathbb{N}$$

and

$$\varnothing_2 : \mathbb{P}\,\beta$$

Our aim is to unify the two types. This is easily achieved with β equal to $\mathbb{N}$. With this substitution the types of both sides of the relational expression $S = \varnothing$ are completely and uniquely defined as required.

The defining term of the set expression is $\varnothing_3$ and using another new variable we represent its type as $\mathbb{P}\,\gamma$. And so for the whole set comprehension,

$$\{S : \mathbb{P}\,\mathbb{N} \mid S = \varnothing_2 \bullet \varnothing_3\} : \mathbb{P}\,\mathbb{P}\,\gamma$$

and it is $\mathbb{P}\,\mathbb{P}\,\gamma$ that we must finally unify with $\mathbb{P}\,\alpha$. This tells us that α must be equal to $\mathbb{P}\,\gamma$, but there is no unique way to give values to α and γ. So the expression is incorrectly typed because it does not contain enough type information.

Functions

As shown above, unification may be necessary for any two terms related by some relational operator. At the beginning of this section we gave an example where a function application gave rise to a type error. Function application is another form of expression where unification may be required. Here we consider how the type checker should deal with a function application.

A function in Z is just a set of ordered pairs, and so will have type $\mathbb{P}(X \times Y)$ for some X and Y, possibly containing generic parameters. A function application is a term consisting of the function name and the argument to which it is applied. The whole expression is well typed if the argument has a type unifiable to X, and its type is Y (possibly with suitable instantiation).

Examples of possible situations arising from function application:

- $\mathit{first}(3, 1)$ This is a well-typed term with all generic parameters fully instantiated. This instance of the function *first* has type $\mathbb{P}((\mathbb{N} \times \mathbb{N}) \times \mathbb{N})$. The argument has type $\mathbb{N} \times \mathbb{N}$ and the type of the whole term is $\mathbb{N}$.

- $\mathit{first}(\varnothing, 3)$ Using a type variable, α, we can represent the type of this occurrance of *first* as $\mathbb{P}(((\mathbb{P}\,\alpha) \times \mathbb{N}) \times \alpha)$. The argument has type $(\mathbb{P}\,\alpha) \times \mathbb{N}$ and the type of the whole term is $(\mathbb{P}\,\alpha)$. The presence of a type variable is acceptable at this stage because in a wider context the function application term may well be related to some other term which gives us more information. For instance, if the context were $\mathit{first}(\varnothing, 3) \in \mathbb{F}\,\mathbb{N}$ then we could unify α with $\mathbb{N}$ and all would be well.

- *first*$(3, \varnothing)$ Using type variable, β, *first* has type $\mathbb{P}(\mathbb{N} \times \mathbb{P}\,\beta) \times \mathbb{N})$ and the argument has type $\mathbb{N} \times \mathbb{P}\,\beta$. Tbe only possibility for the type of the whole term is $\mathbb{N}$. But this has no reference to the type variable, β, which would get left behind, forever uninstantiated. So we can tell that, whatever its context, the function application cannot be correctly typed.

To deal with all these possibilities the type checker can behave in the following way when dealing with a function application. First, find the type of the function (which is possibly generic and of the form $\mathbb{P}(X \times Y)$). Find the type, Z, of the argument and unify this with X. Then use the information gained from the unification to instantiate X and Y. There may be type variables left in both X and Y. Type variables left in Y are permissible at this stage because they may be given values by the wider context. However, type variables left in X which do not also appear in Y have no possibility of instantiation and a type error should be reported.

An algorithm for unification

A successful unification will return a (possibly empty) set of substitutions assigning actual types to type variables. A substitution is represented as a partial injection:

$$SUBST == word \rightarrowtail\!\!\!\!\!+ \; TYPE$$

In fact, the result is defined as belonging to the following type:

$$\begin{aligned} OPTSUBST ::= &\; just \ll SUBST \gg \\ | &\; nothing \end{aligned}$$

which allows *nothing* to be returned when a type conflict is discovered. The function *unopt* projects the substitution from an *OPTSUBST*:

$$\begin{array}{|l} unopt : OPTSUBST \nrightarrow SUBST \\ \hline \forall\, s : SUBST \bullet unopt\,(just\; s) = s \end{array}$$

To aid the description of the unification process we declare, but do not fully define the following useful functions:

$$\begin{array}{|l} applysub : SUBST \to TYPE \to TYPE \end{array}$$

Given a substitution and a type the function *applysub* applies the substitution to the type, yielding a new type.

| $tyvars : TYPE \rightarrow \mathbb{P}\, word$

The function *tyvars* gives the set of type variable names which occur within a given type.

| $order : TYPE \rightarrow TYPE$

The function *order* is used when dealing with schema types. It forms a sequence of all the types bound within the schema type, the order being the lexicographical order of the identifier names. Finally, we have already used the function

| $assignvars : GENTYPE \rightarrow TYPE$

which, for each uninstantiated generic parameter in a generic type assigns a fresh type variable, thus converting a generic type to a type with type variables which were not previously in use.

The following function, *unify*, finds the unifying substitution (if any).

```
unify : SUBST → (GENTYPE × GENTYPE) → OPTSUBST
─────────
∀ s : SUBST; n : word; t, u : TYPE; i : IDENT; tl, ul : TYPEs •
    (unify s (varty n, t) =
        if n ∈ dom s then (unify s ((s n), t))
        else (if n ∈ uu then nothing
              else just((s; (applysub{n ↦ uu})) ∪ {n ↦ uu})
              where uu = applysub s t
        )
∧
    unify s (t, varty n) =
        if n ∈ dom s then (unify s (t, (s n)))
        else (if n ∈ uu then nothing
              else just((s; (applysub{n ↦ uu})) ∪ {n ↦ uu})
              where uu = applysub s t
        )
∧
    unify s (identty i, identty i) = just s
∧
    unify s (powerty t, powerty u) = unify s t u
∧
    unify s (productty {}, productty {}) = just s
∧
    unify s (productty < t > ⁀tl, productty < u > ⁀ul) =
        if ss = nothing then nothing
        else unify (unopt ss) (productty tl)(productty ul)
        where ss = unify s t u
∧
    unify s (schematy t, sehematy u) =
        if (dom t) = (dom u) then unify s (order (schematy t), order (schematy u))
        else nothing
∧
    unify s (unity, t) = s
∧
    unify s (t, unity) = s
∧
         . . . for all other cases return nothing
    )
```

With this function we can type check a function application term:

```
$typeofApply : ENV \rightarrow (expr \times expr) \rightarrow TYPE$
----------------------------------------------------------------
$\forall\, env : ENV;\ s, t : expr \bullet$
      $typeofApply\ env\ (s, t) =$
          if $(sub = nothing)$ then $value(unity, badunification)$
          else (if $(tyvars\ ss) \subseteq (tyvars$
$tt)$ then $tt$
              else $value(unity, badapplication)$
              where $ss = applysub\ (unopt\ sub)\ stype \wedge$
                     $tt = applysub\ (unopt\ sub)\ ttype$
              )
          where $stype, ttype : TYPE;\ sub : SUBST\ |$
                 $stype = typeof\ env\ s\ \wedge$
                 $ttype = typeof\ env\ t\ \wedge$
                 $sub = unify\ \varnothing\ (stype, ttype)$
```

11 Predicates

The type checker must make sure that all predicates are correctly typed. We have represented this by the requirement:

$$(p, env) \in groundpreds$$

for a predicate, p, and environment, env.

A predicate is correctly typed with respect to an environment if all its constituent terms and predicates are correctly typed. As discussed in the previous section, unification may be necessary, and all type variables must at this stage be assigned actual types by the unification process. The following function to apply unification and check for the presence of type variables will be of use:

$$
\begin{array}{l}
\mathit{unifypred} : (\mathit{TYPE} \times \mathit{TYPE}) \rightarrow \mathit{TYPE} \\
\hline
\forall s, t : \mathit{TYPE} \bullet \\
\quad \mathit{unifypred}(s, t) = \\
\qquad \textbf{if } (\mathit{tyvars}\ ss) = (\mathit{tyvars}\ tt) = \varnothing \textbf{ then } ss \\
\qquad \textbf{else } \mathit{value}\ (\mathit{typevarsinpred}, \mathit{unity}) \\
\qquad \textbf{where } \mathit{sub} : \mathit{SUBST};\ ss, tt : \mathit{TYPE} \mid \\
\qquad\quad \mathit{sub} = \mathit{unify}\ \{\}\ (s, t) \land \\
\qquad\quad ss = \mathit{applysub}\ \mathit{sub}\ s \land \\
\qquad\quad tt = \mathit{applysub}\ \mathit{sub}\ t
\end{array}
$$

We can now define the set *groundpreds*:

$$
\begin{array}{l}
\mathit{groundpreds} : \mathbb{P}(\mathit{pred} \times \mathit{ENV}) \\
\hline
\forall t1, t2 : \mathit{expr};\ p1, p2 : \mathit{pred};\ \mathit{env} : \mathit{ENV} \bullet \\
\quad (\mathit{Equal}(t1, t2), \mathit{env}) \in \mathit{groundpreds} \Leftrightarrow \\
\qquad\quad \mathit{unifypred}((\mathit{typeof}\ t1\ \mathit{env}), (\mathit{typeof}\ t2\ \mathit{env})) \in \mathit{TYPE} \\
\land \\
\quad (\mathit{Member}(t1, t2), \mathit{env}) \in \mathit{groundpreds} \Leftrightarrow \\
\qquad\quad \mathit{unifypred}(\mathit{powerty}\ (\mathit{typeof}\ t1\ \mathit{env}), (\mathit{typeof}\ t2\ \mathit{env})) \in \mathit{TYPE} \\
\land \\
\quad (\mathit{Implies}(p1, p2), \mathit{env}) \in \mathit{groundpreds} \Leftrightarrow \\
\qquad ((p1), \mathit{env}) \in \mathit{groundpreds}\ \land\ (p2), \mathit{env}) \in \mathit{groundpreds}) \\
\land \\
\quad (\mathit{Truth}, \mathit{env}) \in \mathit{groundpreds}) \\
\land \quad \ldots
\end{array}
$$

The idea is that the test for membership of *groundpreds* is always satisfied (since *unifypred* always returns a type), but performing the check would have the side effect of generating appropriate error messages where necessary.

For a quantification, the predicates are checked in an environment updated with the signatures of the quantified variables. Eg:

$$
\begin{array}{l}
\hline
\forall \mathit{decs} : \mathit{decls};\ p1, p2 : \mathit{pred};\ \mathit{env} : \mathit{ENV} \bullet \\
\quad (\mathit{Forall}(\mathit{Schema}(d, p1), p2), \mathit{env}) \in \mathit{groundpreds} \Leftrightarrow \\
\qquad ((p1, \mathit{newenv}) \in \mathit{groundpreds}\ \land\ (p2, \mathit{newenv}) \in \mathit{groundpreds}) \\
\quad \textbf{where} \\
\qquad \mathit{newenv} : \mathit{env} \mid \mathit{newenv} = \mathit{installsigs}\ \mathit{env} \bigcup(\mathrm{ran}(\mathit{map}\ \mathit{normdecl}\ \mathit{decs}))
\end{array}
$$

A schema reference used as a predicate is correctly typed if all the components of the schema are currently in scope and have the same types as in the schema. The predicate part of the schema paired witb the current environment must also be a memher of *groundpreds*.

12 The Document

A Z specification is structured as a sequence of *paragraphs*, each being a declaration (of a schema, given sets, axiomatically defined constants, syntactic equivalences, or data types), a predicate (indicating a constraint), or theorem. To check a specification, the type checker checks each paragraph with respect to a current environment, adapting this current environment for those paragraphs introducing declarations. The first paragraph is checked with respect to a *primitive environment* corresponding to the Z library. The final result of the type checker after checking a type correct document is an environment containing the definitions which are in scope at the top level of the document.

We examine here paragraphs introducing simple axiomatic definitions, schema definitions, and syntactic equivalences. Finally, we produce the function which pulls everything together by checking the entire document.

A simple nongeneric axiomatic definition consists of a set of declarations together with a predicate. The function *installLet* returns the current environment adapted with normalised declarations, and checks the predicate with respect to this adapted environment:

$$
\begin{array}{|l}
installLet : ENV \rightarrow (decls \times pred) \rightarrow ENV \\
\hline
\forall\, env : ENV;\ decs : decls;\ pre : pred \bullet \\
\qquad installLet\ env\ (decs, pre) = newenv \\
\quad\ \ \wedge\ (pre, newenv) \in groundpred \\
\mathbf{where}\ newenv : ENV \mid newenv = adapt\ env\ decs
\end{array}
$$

A syntactic equivalence definition, which may be generic, equates an identifier with an expression. The function *installEqeq* determines the type of an expression with respect to the current environment with the generic parameters added as given sets, and installs the resulting generic signature associating the identifier with the type of the expression into the current environment:

$$\begin{array}{|l}
installEqeq : ENV \rightarrow ((ident \times GENPARAMs) \times expr) \rightarrow ENV \\
\hline
\forall env : ENV;\ id : ident;\ gens : GENPARAMs;\ exp : expr \bullet \\
\quad installEqeq\ env\ ((id, gens), exp) = \\
\quad\quad installgensig\ env\ (id,\ gens,\ (typeof\ (installgivens\ env\ gens)\ exp))
\end{array}$$

A schema definition, which may be generic, consists of a name which is an undecorated identifier, possibly some generic parameters, and a body which is a schema expression. The function *installSdef* normalises a schema expression with respect to the current environment to which has been added the generic parameters as given sets, and then installs the resulting generic schema signature associating the given name with the schema type of the normalised signature into the current environment:

$$\begin{array}{|l}
installSdef : ENV \rightarrow (word \times GENPARAMs \times sexp) \rightarrow ENV \\
\hline
\forall env : ENV;\ wd : word;\ gens : GENPARAMS;\ sexpr : sexp \bullet \\
\quad installSdef\ env\ (wd, gens, sexpr) = installgenschemasig\ env \\
\quad\quad (Ident(wd, <>),\ gens,\ normSexp\ (installgivens\ env\ gens)\ sexpr)
\end{array}$$

Given functions to check each particular sort of paragraph we can define *installpara* which handles an arbitrary paragraph:

$$\begin{array}{|l}
installpara : ENV \rightarrow para \rightarrow ENV \\
\hline
\ldots \\
\quad \wedge\ installpara\ env\ (Given(ids)) = installgivens\ env\ (\mathrm{ran}\ ids) \\
\quad \wedge\ installpara\ env\ (Let(Schema(decs, pre))) = installLet\ env\ (decs, pre) \\
\quad \wedge\ installpara\ env\ (Eqeq(Lhs(id, gens), exp)) = installEqeq\ env\ ((id, gens), exp) \\
\quad \wedge\ installpara\ env\ (Sdef(wd, gens, sexpr)) = installSdef\ env\ (wd, gens, sexpr) \\
\ldots
\end{array}$$

Checking the entire document

To check the entire document, the type checker checks each paragraph in turn - with the first paragraph checked with respect to a primitive environment corresponding to a library. The result of this checking (ignoring side effect error messages) is an environment of signatures which has been incrementally constructed after each paragraph. We are now ready to define a function which checks the syntax tree for an entire Z document. Its primitive environment would typically be one corresponding to the conventional Z library, but it could be arbitrary (indeed, the Z library environment itself could be generated with the *null* environment). The function *checkspec* checks a document with respect to an arbitrary (library) environment:

$$\begin{array}{|l}
checkspec : ENV \rightarrow spec \rightarrow ENV \\
\hline
\forall\, primenv : ENV;\ doc : seq_1\, para \bullet \\
\quad checkspec\ primenv\ \varnothing = primenv \land \\
\quad checkspec\ primenv\ doc = checkspec\ (installpara\ primenv\ (head\, doc))\ (tail\ doc)
\end{array}$$

13 Related Work

Peter Hancock has defined a type checker for the functional language Miranda in Miranda itself[HAN87]. He represents success and failure with a defined type *reply*. This type is used for functions which may succeed or fail, returning a "proper value" upon success or a special value *FAILURE* upon failure. However, in recognition that error-handling code tends to obscure the code for correct cases, in [HAN87] Hancock has chosen to give an abridged version which does not provide any error messages indicating the reason for failure. In Miranda, the only object that can appear as the right half of a declaration is a type, so there is no notion of *normalising* a declaration - a major task for a Z type checker.

C. Sennett has produced a Z specification of a Z type checker, which has been implemented at RSRE[SEN87]. He presents a model for a type checker which operates in parallel with a one pass parser. The complete specification consists of a set of schema operations defined for syntactic constructs individually presented to the type checker as they are parsed. For certain constructs (e.g., the θ term), his model deviates from the type semantics given in [SPI88a]. This is in contrast to our abridged specifiation of a type checker which checks a complete abstract syntax tree according to the type semantics of [SPI88a].

Two other type checking systems have evolved from the Programming Research Group, Oxford. One has been produced by Mike Spivey and is known as *Fuzz*. Fuzz obtains its input by extracting the formal text from a LATEX input file. Spivey provides a set of LATEX macros with which to write Z text. It uses the type system described in [SPI88a] for type matching, but also uses type abbreviations when computing the type of expressions. This allows more meaningful error reporting, reflecting the way in which objects have been defined. Fuzz initially loads a "prelude" containing the mathematical tool-kit definitions of [SPI88b], which can be extended or substituted if required. Fuzz is written in C, can work on PCs and is commercially available.

Bernard Sufrin has produced an ML parser and type checker for Z known as *zebra*. He is currently producing a modular ML system known as *hippo* which can be used as a front end to processing Z in many different applications. The programs use ascii input (which can be supplied directly or translated from other forms, such as QED output). They can take input from a file, or be used interactively. The current environment can be interrogated to find the types of particular identifiers or expressions. A standard Z library database is provided for zebra. Again, this can be altered or other databases used as required. The syntax understood by the systems is different from (in general, more permissive than) that of [SPI88b]. For instance, generic data type definitions are permitted. Also, the type system is somewhat different with overloading supported and objects treated in [SPI88b] as generic symbols here viewed as functions. For example: following [SPI88b] the relation symbol $\leftrightarrow$ would be an infix generic symbol with type:

$$\_ \leftrightarrow \_ : [X, Y]\ \mathbb{P}\mathbb{P}(X \times Y)$$

zebra gives the type as:

$$\_ \leftrightarrow \_ : [X, Y]\ \mathbb{P}(((\mathbb{P}\, X) \times (\mathbb{P}\, Y)) \times (\mathbb{P}\mathbb{P}(X \times Y)))$$

The use of "pseudotypes" allows the user to nominate certain sets to be treated as much like types as possible, making reported types more recognisable to the user.

14 Conclusions

The model of a Z type checker that we present can be summarised as follows:

For a Z abstract syntax tree, the type checker produces an environment of signatures. These signatures indicate the generic parameters (if any) and type of each variable in scope at the top most level of the tree. The type checker builds the environment incrementally, starting with an environment corresponding to a predefined library, and then checking each paragraph node of the tree in turn to extend the environment accordingly.

The key to incorporating new declarations into an environment is a notion of normalising declarations into signatures associating variables with their types. For a simple declaration introducing a variable drawn from an expression representing a set, a signature is formed by associating the variable with the type of the expression with the $\mathbb{P}$ *removed. For a schema name used as a declaration, the normalised signature list consists of the normalised component signatures.*

The determination of the type of an expression is recursive on the structure of the expression. The type of a schema expression, also recursively calculated from its structure, involves normalising the schema expression into its component signatures. Each of these calculations may require checking a predicate with respect to an environment adapted with new declarations.

Error diagnostics are generated as side effects as the type checker visits each node.

Because the complete algorithm is very diverse with a high degree of mutual recursion, a single fully comprehensive specification would by its complex nature not reveal a clear introductory overview of the approach. Specifying how to check every possible form of expression would involve a good deal of repetition of the techniques used. We have therefore chosen to present an abridged version directed at those wanting an introduction to the problem of implementing a Z type checker. The various expressions chosen for explanation are intended to form a representative sample, covering the basic functions of the type checker, and some of the less obvious details too. Thus we have presented an "underspecification", with details included either because they are essential for presenting the model (e.g., normalising declarations), or because they are interesting in their own right (e.g., the θ term).

Our original implementation of the type checker for the Forsite prototype represented a transliteration of the formal definitions of [SPI88a] into the language ML, using functions very similar to those described here. Since ours was the first effort to build a type checker for Z based on its denotational semantics, we chose to focus on functional correctness. Little attention was paid to implementation issues such as performance, but even so, the prototype system provided a useable type checker which indicated what could be achieved if the prototype system were to be developed into a carefully engineered product.

References

[KSW88] KING, SORENSEN & WOODCOCK *Z: Grammar and Concrete and Abstract Syntaxes,* Oxford University Programmimg Research Group Technical Monograph, PRG 68.

[SPI88a] J. M. SPIVEY *Understanding Z,* Cambridge Tracts in Theoretical Computer Science, Cambridge University Press (1988).

[SPI88b] J. M. SPIVEY *The Z Notation: A Reference Manual,* Prentice-Hall International (1988).

[HAN87] P. HANCOCK *A Type-Checker,* **The Implementation of Functional Programming languages,** ed. S.L.P. Jones, Prentice-Hall International (1987).

[SEN87] C. T. SENNETT *Review of Type Checking and Scope Rules of the Specification Language Z,* RSRE Report No. 87017 (1987).