Typechecking Z

Ian Toyn[1], Samuel H. Valentine[1], Susan Stepney[2], and Steve King[1]

[1] Department of Computer Science, University of York, Heslington, York, YO10 5DD, UK.
{ian,sam,king}@cs.york.ac.uk
[2] Logica UK Ltd, Betjeman House, 104 Hills Road, Cambridge, CB2 1LQ, UK.
stepneys@logica.com

Abstract. This paper presents some of our requirements for a Z typechecker: that the typechecker accept all well-typeable formulations, however contrived; that it gather information about uses of declarations as needed to support interactive browsing and formal reasoning; that it fit the description given by draft standard Z; and that it be able to check some particular extensions to Z that are intended to allow explicit definitions of schema calculus operators. The paper presents a specification of such a Z typechecker, which we have implemented.

1 Introduction

Algorithms for typechecking polymorphic functional languages, as explained by Cardelli [1] and by Hancock [2], are readily adaptable to typecheck Z specifications and their generic constructs. They are based around Milner's unification algorithm [6]. Spivey and Sufrin gave an account of typechecking Z [12], focusing on the inference of implicit generic instantiations. They deliberately omitted any discussion of the typechecking of schemas. We have found some schemas that are awkward to typecheck but could be well-typed. An investigation of the typechecking of schemas is particularly important in view of the merging of schemas with expressions in draft standard Z [15]. Our work has involved the construction of a Z typechecker within the CADiZ toolset [17, 18], replacing a previous inferior algorithm.[1] Some other requirements on the design of the new typechecker are also discussed in this paper, namely keeping track of uses of declarations for the purposes of interactive browsing and formal reasoning, and the typechecking of extensions to the Z notation to permit explicit definition of schema calculus operators [21].

2 Types

Each Z type corresponds to a set of values known as its *carrier set*. The type system excludes combinations of expressions whose values are related in ways

[1] This new typechecker has approved the formal Z in this paper.

that are inappropriate based on their types. It is unlikely that such combinations of expressions could be given the intended meanings. A typechecker implementing the type system can decide automatically whether to accept or reject any combination of expressions. Some of the goals that arise in formal reasoning are properties that the typechecker has already decided, and so another advantage of the type system is that it allows such goals to be discharged automatically. There are also disadvantages in having a type system. For example, the rejection of combinations of expressions that could have had sensible meanings makes the language less expressive, and explicit injections may inconveniently be needed to cast values between types. Also, the kinds of goals that are decided by the typechecker are likely to be ones that a theorem prover could decide anyway. Lamport and Paulson have discussed the advantages and disadvantages of decidable and undecidable type systems for specification languages [5]. We take the Z type system as a given—our aim is to provide a specification of how to enforce it. An implementation of this specification should reject all ill-typed Z specifications while accepting all well-typed Z specifications.

The various kinds of types in the Z type system are illustrated by the following examples. *Given types* are introduced by given set paragraphs.

$$[\mathit{PERSON}, \mathit{NAME}, \mathit{AGE}]$$

Using the notation of draft standard Z [16], the types introduced by this paragraph are denoted by $\mathtt{GIVEN}\ \mathit{PERSON}$, $\mathtt{GIVEN}\ \mathit{NAME}$ and $\mathtt{GIVEN}\ \mathit{AGE}$. The members of their carrier sets are as yet unspecified; they may be constrained by subsequent paragraphs.

Types can be assembled in three ways to form larger types. First, the set of all subsets of a type is itself a type—a *powerset type*. For example, a team comprises a set of persons.

$$\mathit{team} : \mathbb{P}\,\mathit{PERSON}$$

The type of *team* is $\mathbb{P}(\mathtt{GIVEN}\ \mathit{PERSON})$. Second, the set of all tuples of a certain size of values from other types is itself a type—a *Cartesian product type*. For example, personal details can be represented as a tuple.

$$\mathit{personal\_details}1 : \mathit{PERSON} \times \mathit{NAME} \times \mathit{AGE}$$

The type of the triple $\mathit{personal\_details}1$ is $\mathtt{GIVEN}\ \mathit{PERSON} \times \mathtt{GIVEN}\ \mathit{NAME} \times \mathtt{GIVEN}\ \mathit{AGE}$. Third, a type can be a product type but with labels on the components—a *schema type*. For example, personal details can be represented by a binding from a schema.

$$\mathit{personal\_details}2 : [\mathit{person} : \mathit{PERSON};\ \mathit{name} : \mathit{NAME};\ \mathit{age} : \mathit{AGE}]$$

The type of the binding $\mathit{personal\_details}2$ is $[\mathit{person} : \mathtt{GIVEN}\ \mathit{PERSON};\ \mathit{name} : \mathtt{GIVEN}\ \mathit{NAME};\ \mathit{age} : \mathtt{GIVEN}\ \mathit{AGE}]$. The association of names and types within the square brackets is called a *signature*.

Generic definitions require additional type notation. An example of a generic definition is that of the empty set.

$\varnothing[X] == \{x : X \mid \mathit{false}\}$

The type of $\varnothing$ is generic in X, and is written as the *generic type* $[X]\,\mathbb{P}(\mathtt{GENTYPE}\,X)$, the type of the reference expression X within the set comprehension being $\mathbb{P}(\mathtt{GENTYPE}\,X)$. Such generic types are used only in describing the types of generic definitions, and so never appear within other types.

The notation used for types is similar to that used in the expressions that denote their carrier sets, with the difference that given and generic types are distinguished.

Z's free type notation does not require any additional type notation: free types are abbreviations for given sets with constraints on their members [11, 16, 20].

3 Requirements on the Typechecker

This section discusses some issues for the design of our typechecker.

3.1 Schemas

Schemas have signatures that influence the environment in which formulæ are typechecked. For example, in the set comprehension $\{n : \mathbb{N} \mid n \geq 1 \bullet 2 * n\}$ there is the schema $n : \mathbb{N} \mid n \geq 1$ which declares an n of numeric type that is referenced from both the $\mid$ part of the schema itself and from the $\bullet$ part of the set comprehension. In the majority of schemas that occur in real specifications, their signatures can be determined by checking their declarations alone. In the exceptional cases, it is preferable not to demand that the specifier reformulate in a way with which the typechecker can cope. Indeed, the formulations may have arisen not from being written by hand but as the results of semantically-valid inferences in a tool.[2] We contrive some exceptional cases below.

We use draft standard Z notation, including its toolkit.

section *contrivedExamples* parents *standard_toolkit*

[2] Special provisions are needed with some inference rules to avoid variable capture. For example, the predicate $m \in \{n : \mathbb{N} \mid n \geq 1 \bullet 2 * n\}$ is equivalent to the predicate $\exists\, n : \mathbb{N} \mid n \geq 1 \bullet m = 2 * n$ unless the name m had been n, in which case it becomes captured by the local declaration of n. Inference rules also must be careful with implicit instantiations. For example, applying the one-point inference rule [7] to the predicate $\exists\, x, y : \mathbb{P}\,\mathbb{A} \mid x = \varnothing \land y = \varnothing \bullet x = y$ produces $\varnothing = \varnothing$, which without explicit instantiations is type erroneous. A common provision to guard against these potential errors is to apply the typechecker and reject the inferences if any errors are detected. An alternative, as used in CADi$\mathbb{Z}$, is to make the inference rules inherit types and instantiations from their operands onto their results, and to automatically rename variables to avoid variable capture.

The first example also conforms to the notation of the Z reference manual [11].

$$\begin{array}{|l|}
\multicolumn{1}{l}{S1} \\ \hline
x : \mathbb{P}\,\varnothing \\ \hline
x = \varnothing[\mathbb{Z}] \\ \hline
\end{array}$$

In this example, the named schema paragraph has a single declaration and a single predicate. By considering only the declaration, the signature can be seen to contain the single name x, but the type of that name is not completely determined. The type of x as determined by the declaration may be expressed as $\mathbb{P}\,\alpha$, where α is a *variable type* (or *type variable*). Consideration of the predicate part of the schema constrains this type to $\mathbb{P}\,\mathbb{Z}$, assuming that this Z is written according to [11], or to $\mathbb{P}\,\mathbb{A}$ if it is viewed as draft standard Z [16].[3] Almost all Z typecheckers accept this example as being well-typed by this means, e.g. Hippo [14] and ZTC [4].

In draft standard Z notation, as well as the requirement to cope with signatures in which the types of some components are incompletely determined, there is also a requirement to cope with signatures in which the names of the components are incompletely determined.

$$g[X] == X$$

g is a generic definition that will be referred to without an explicit instantiation.

$$\begin{array}{|l|}
\multicolumn{1}{l}{S2} \\ \hline
s == g \\ \hline
s = [x, y, z : \mathbb{Z}] \\
\{s \mid x = y\} = \varnothing \\ \hline
\end{array}$$

In example $S2$, the reference to g is in the declaration of s,[4] where its type (and hence its implicit instantiation) is not constrained at all. The first conjunct constrains s, and hence g, to be a schema with a particular signature.[5] This signature could not be determined from the declaration alone. Yet this paragraph could be well-typed, and so a typechecker ought not to complain.

$$\begin{array}{|l|}
\multicolumn{1}{l}{S3} \\ \hline
s == g \\ \hline
\{s \mid x = y\} = \varnothing \\
s = [x, y, z : \mathbb{Z}] \\ \hline
\end{array}$$

Example $S3$ is similar to $S2$, differing only in that the conjuncts have been swapped. Again, the signature of the schema cannot be determined without

[3] Draft standard Z uses $\mathbb{A}$ as the type of all numbers, including the integers $\mathbb{Z}$.
[4] Draft standard Z allows use of the $==$ notation in local as well as global declarations.
[5] This is an example of a schema being used as an expression.

consideration of the schema's predicate part. In this case, the first conjunct uses s as an inclusion declaration, which constrains its type to be that of a schema, but without constraining its signature. The environment in which the equality $x = y$ is to be typechecked is consequently not yet known, but it can be determined by consideration of the second conjunct. A typechecker that considers conjuncts in one particular order could not cope with both $S2$ and $S3$. The $S3$ paragraph could be well-typed, and a typechecker ought not to complain.

$S4$

$s == g$
$t == g$

$\{s \mid x = y \lor t = [z : \mathbb{P}\, x]\} = \varnothing$
$s = [x, y : \mathbb{P}\, \mathbb{Z};\ z : \mathbb{P}\, t]$

Example $S4$ is like $S2$ and $S3$, except that instead of one conjunct providing information to help typecheck the other, information has to flow both ways. The types of x and y are determined by the second conjunct, then the type of t can be determined by the first conjunct, then the type of s and its z component can be determined from the second conjunct, and only then can remaining constraints within the first conjunct be checked.

Examples can be contrived in which the mutual dependencies between conjuncts are such that the constraints cannot be solved. We consider those to be ill-typed.

The requirements arising from these examples are that a typechecker should not insist on solving constraints in any particular order, except as necessitated by dependencies between the constraints themselves, and that it must be able to cope with constraints in which are signatures whose names are unknown. This requires *variable signatures*, analogous to variable types. A recursive descent of the phrase tree of a specification checking constraints along the way, e.g. as in [9], does not satisfy the first requirement.

3.2 Browsing

A Z browser is a tool that presents a view of a Z specification and allows the user to select formulæ and ask questions about them [13, 17]. An example is the selection of a reference expression and the question "where is the referenced variable's declaration?". This question is not so easy to answer as might at first appear. One problem is that a Z schema text can have several declarations of the same name: so long as they have the same types, these declarations are regarded as being merged into a single declaration. Another problem arises from schema inclusion declarations. A schema inclusion declaration declares variables of the same names and types as those of the included schema. Constraints imposed on the included declarations (such as the chained relation in the following example) do not constrain the original schema's variables. Hence uses of the included declarations should not be regarded as uses of the original schema's components.

The following contrived specification illustrates these problems, as explained below.

$$schema == [a, b : \mathbb{A}]$$

$$\models ?\ \forall\, schema;\ \ b, c, d : \mathbb{A};\ \ c : \mathbb{A} \bullet a = b = c = d$$

When asking to see the declaration of a reference to a variable, a browser should (at least) direct attention to the schema text where that variable is declared. In the case of the above example's reference to b, there are two merged declarations, one from the inclusion of *schema*, and one explicit, so directing attention to the whole schema text will have to suffice (assuming attention is directed to a single formula). In the case of the reference to d, there is only a single explicit declaration, so attention can be directed to that specific declaration. The variable c has two explicit declarations, so directing attention to the whole schema text is appropriate. The variable a has only one declaration, arising from the inclusion of *schema*, and so it is appropriate to direct attention to that inclusion declaration.

Browsing is a concern for a typechecker because the typechecker is clearly in the best position to determine the declarations referred to by reference expressions. But to succeed, it must be concerned not merely with the names and types that appear in signatures as introduced above, but with specific declarations and schema texts.

As well as mapping reference expressions to variable declarations, a browser may map variable declarations to uses of those variables. Given the assumption that attention is directed to a single formula, this can be achieved using questions such as "where is the first use?", "where is the next use?", etc. The uses of a variable are not just explicit reference expressions. There may also be uses of variables in the implicit instantiations of references to generic definitions, and uses implicit in binding construction (*theta*) expressions (these being equivalent to binding extensions involving reference expressions) and schema predicates (these being defined in terms of theta expressions). A browser might wish to draw attention to expressions that contain such implicit uses of variable declarations.

The knowledge of where a variable is used is relevant not only to interactive browsing, but also to formal reasoning. For example, the `one-point` rule must find all uses of a variable in replacing them by an expression of equal value [7].

A browser might also allow inspection of the types of expressions and the signatures of schemas. This is relatively easy for a typechecker to support: it just has to note that which it infers. This information is especially useful to specifiers attempting to understand type errors, as well as to implementors of typecheckers.

3.3 Draft Standard Z

In draft standard Z, the type system is given, as input, an annotated syntax tree in which some formulæ already have type annotations expressing constraints between their types. As well as determining whether the given specification is well-typed, the type system is required to assign type annotations to formulæ for use later in defining the semantics of formulæ such as schema negations.

The type system as presented in draft standard Z has been subject to some criticism concerning the overloading of τ as a meta-variable and as a variable type. A requirement on the type system presented below is to avoid that criticism, while at the same time being presented in a way that satisfies the requirements of draft standard Z.

3.4 Type-Constrained Generics

A companion paper to this one proposes some extensions to Z that would enable explicit definitions of schema calculus operators [21]. Such definitions would be similar to Z's existing generic definitions, except that whereas the parameters of existing generic definitions can be instantiated with any sets, the parameters of schema calculus operators should be instantiated with schemas, usually with constraints between their signatures. For example, schema projection takes two schemas and returns the schema that is the set of bindings of just the names present in the right operand but subject to the constraints of both schemas.

$$\text{function } 32 \text{ leftassoc } (\_ \mathit{schProj} \_)$$

$$\_ \mathit{schProj} \_[\dagger X, Y] == \lambda S : \mathbb{P} X;\ T : \mathbb{P} Y \bullet \{S;\ T \bullet \theta Y\}$$

The $\dagger$ symbol separates generic parameters to its left (none in this example) from parameters to its right that should be instantiated with schemas (X, Y). The schema $S;\ T$ imposes constraints on X and Y that they be compatible schemas. If the $\dagger$ had been omitted, this would have been regarded as an error, as the types of generic parameters may not be constrained. With the $\dagger$, we want the definition to be well-typed, even though the signature of the schema $S;\ T$ is unknown. Given the lack of precise knowledge of signatures, how is the expression θY to be typechecked? What if the instantiation of X or Y was a schema with Y as one of its components? We would not want such a component to capture the reference to Y, as then the definition would not have the desired semantics of schema projection. We shall need to achieve the effect of the names in the signatures of the instantiating schemas being in a different name-space from the names declared explicitly in the definition.

A further extension to Z proposed in the companion paper [21] is undecoration expressions, which are needed for schema calculus operators whose definitions depend on decorations.

4 Specification of the Typechecker

The specification of the typechecker takes the form of a formal system, for reasons as given by Cardelli [1].

"A typechecking algorithm, in some sense, implements a formal system, by providing a procedure for proving theorems in that system. The formal system is essentially simpler and more fundamental than any algorithm, so that the simplest presentation of a typechecking algorithm is the formal system it implements. Also, when looking for a typechecking algorithm, it is better to first define a formal system for it."

The specification is presented in bottom-up order, first introducing the notations to be used, then presenting the individual type inference rules, and finally explaining how these are composed in forming the whole typechecker. Type-constrained generics are addressed separately, having first presented a type-checker for draft standard Z. We also explain how implicit instantiations are determined, as that has to be revised to cope with type-constrained generics.

4.1 Notations

Phrases The definition of the syntax of Z phrases is assumed. Defined here is the syntax of notation for types and signatures (as exemplified earlier), and notation for environments to be used during typechecking. Phrases of this syntax denote values in the type universe.[6]

```
Type = 'GIVEN' , NAME                                              (* given type *)
     | 'GENTYPE' , NAME                                (* generic parameter type *)
     | 'ℙ' , Type                                               (* powerset type *)
     | Type , '×' , Type , { '×' , Type }             (* Cartesian product type *)
     | '[' , Sig , ']'                                            (* schema type *)
     | '[' , NAME , { ',' , NAME } , ']' , Type , [ ',' , Type ]  (* generic type *)
     | 'α' , { STROKE }                                         (* variable type *)
     | '(' , Type , ')'                                    (* parenthesized type *)
     ;
Sig  = [ NAME , ':' , Type , { '; ' , NAME , ':' , Type } ]
     | 'β' , { STROKE }                                    (* variable signature *)
     | 'ε'                                                    (* empty signature *)
     ;
Env  = Sig ;
     | Sig , '⊕' , Sig                                (* overridden environment *)
     ;
```

Generic types never occur within other types, despite this syntax allowing that possibility. The need for the optional second type within a generic type is explained in the context of the type inference rule for reference expression on page

[6] In this syntax, quotes enclose terminal symbols, comma concatenates phrases, square brackets enclose optional phrases, braces enclose phrases to be repeated zero or more times, and vertical bar separates alternatives [3].

13. Variable types and variable signatures denote unknown values that will be determined by solving the constraints in which they appear. Similar variables are needed for `NAME`s, for which we use (subscripted) $\imath$ and $\jmath$. An empty signature could be written as nothing, but writing ε is clearer. There is also an annotation operator $⨾$ that allows types and signatures to be associated with Z phrases.

Metavariables Metavariables appear in patterns that, when matched against existing known phrases, become associated with existing known values. Metavariables are named according to the type of phrase that they can match, as listed in Table 1. Where a pattern has to match several phrases of the same types, the names of the metavariables are given distinct numeric subscripts. For example, the pattern $p_1 \wedge p_2$ matches any conjunction predicate, associating p_1 with the left operand and p_2 with the right operand.

Table 1. Metavariables

Symbol	**Definition**
d	matches a `Paragraph` phrase (d for definition/description).
de	matches a `Declaration` phrase.
e	matches an `Expression` phrase.
i, j	match `NAME` tokens or `DeclName` or `RefName` phrases (i for identifier).
p	matches a `Predicate` phrase.
s	matches a `Section` phrase.
t	matches a `SchemaText` phrase (t for text).
τ	matches a `Type` phrase.
σ	matches a `Sig` phrase.
Σ	matches an arbitrary type environment.
$^+$	matches a `STROKE` token.
*	matches a { `STROKE` } phrase.
$\ldots$	matches elision of repetitions of surrounding phrases, the total number of repetitions depending on syntax.

Having matched a metavariable with a phrase, we will use that metavariable as denoting the value of that phrase, for example σ denotes a function from `NAME` to `Type`.

Type Sequents We write type sequents using the $\vdash$ symbol, to assert the well-typedness of the possibly-annotated phrase to the right of that symbol in the environment to its left. This notation is similar to that used by Spivey [10]. We superscript each $\vdash$ with a mnemonic letter to distinguish the syntax of the phrase appearing to its right — see Table 2.

Type Inference Rules Each type inference rule is written in the following form,

Table 2. Type sequents

Formula	**Definition**
$\overset{\mathcal{Z}}{\vdash}\ z$	a type sequent asserting that specification z is well-typed.
$\Lambda\ \overset{\mathcal{S}}{\vdash}\ s \mathbin{⨾} \Gamma$	a type sequent asserting that, in the context of section environment Λ, section s has section-type environment Γ.
$\Sigma\ \overset{\mathcal{D}}{\vdash}\ d \mathbin{⨾} \sigma$	a type sequent asserting that, in the context of type environment Σ, paragraph d has signature σ.
$\Sigma\ \overset{\mathcal{P}}{\vdash}\ p$	a type sequent asserting that, in the context of type environment Σ, predicate p is well-typed.
$\Sigma\ \overset{\mathcal{E}}{\vdash}\ e \mathbin{⨾} \tau$	a type sequent asserting that, in the context of type environment Σ, expression e has type τ.
$\Sigma\ \overset{\mathcal{T}}{\vdash}\ t \mathbin{⨾} \sigma$	a type sequent asserting that, in the context of type environment Σ, schema text t has signature σ.
$\Sigma\ \overset{\mathcal{DE}}{\vdash}\ de \mathbin{⨾} \sigma$	a type sequent asserting that, in the context of type environment Σ, declaration de has signature σ.

$$\frac{type\ \ subsequents}{type\ \ sequent}(constraints)$$

or laid out as follows if the preceding form would extend into the right margin.

$$\begin{array}{c}\dfrac{type\ \ subsequents}{type\ \ sequent}\\ (constraints)\end{array}$$

They can be read as: if the type subsequents are valid, and the constraints are true, then the type sequent is valid. Some type inference rules have no type subsequents, and some have no constraints, but all have one type sequent. The constraints are written using set theory notation; they typically express relationships that are required to hold between types or signatures. They refer to metavariables bound by pattern matching and to variables for which each application of a type inference rule uses fresh occurrences. We try to use Z-like syntax for the set theory notation used in constraints, so that no description of its intended meaning is needed here. The only unusual notation is $\approx$ for compatible relations, and $decor\ '\ i$, which denotes the name that is like the name associated with metavariable i but with the stroke $'$ appended to it. (In contrast, i' is a metavariable name, and $i\ '$ is the schema resulting from decoration of the schema associated with metavariable i.)

We have chosen to use these conventional notations for syntactic definitions and type inference rules because of their conciseness and readability. Others have shown that it can all be done in Z [8, 9]. Our discussion of the type inference system in section 4.3 is devoid of formalism due to lack of space.

4.2 Type Inference Rules

Using the notation introduced above, one type inference rule can be presented for each production of the Z syntax. There is space in this paper to present only some of them; a fuller set is available [19].

Specification

Sectioned specification Each section[7] is typechecked in an environment formed from preceding sections, and is annotated with an environment that it establishes. The constraints that establish these environments are omitted here (but are included in the fuller set of rules). From the environment in which a section is typechecked will be extracted just those section environments established by the section's parents. The type subsequent for the prelude section should be omitted if the prelude is one of the explicit sections of the specification.

$$\frac{\{\} \vdash^{\mathcal{S}} s_{prelude} \fatsemi \Gamma_0 \qquad \Lambda_1 \vdash^{\mathcal{S}} s_1 \fatsemi \Gamma_1 \qquad \ldots \qquad \Lambda_n \vdash^{\mathcal{S}} s_n \fatsemi \Gamma_n}{\vdash^{\mathcal{Z}} s_1 \fatsemi \Gamma_1 \ldots s_n \fatsemi \Gamma_n}$$

Section *Rules omitted.*

Paragraph

Given types paragraph The names should all be different.

$$\frac{}{\Sigma \vdash^{\mathcal{D}} [i_1, ..., i_n]\ \mathtt{END} \fatsemi \sigma} \left(\begin{array}{l} \#\,\{i_1,\ ...,\ i_n\} = n \\ \sigma = i_1 : \mathbb{P}(\mathtt{GIVEN}\ i_1);\ ...;\ i_n : \mathbb{P}(\mathtt{GIVEN}\ i_n) \end{array}\right)$$

Axiomatic description paragraph The signature of the paragraph is that of its schema text.

$$\frac{\Sigma \vdash^{\mathcal{T}} t \fatsemi \sigma}{\Sigma \vdash^{\mathcal{D}} \mathtt{AX}\ t \fatsemi \sigma\ \mathtt{END} \fatsemi \sigma}$$

Generic axiomatic description paragraph The parameter names should all be different. The schema text can refer to the parameters. The signature of the paragraph comprises generic forms of the types from the signature of the schema text.

$$\frac{\Sigma \oplus \{i_1 \mapsto \mathbb{P}(\mathtt{GENTYPE}\ i_1),\ ...,\ i_n \mapsto \mathbb{P}(\mathtt{GENTYPE}\ i_n)\} \vdash^{\mathcal{T}} t \fatsemi \sigma_1}{\begin{array}{c} \Sigma \vdash^{\mathcal{D}} \mathtt{GENAX}\ [i_1, ..., i_n]\ t \fatsemi \sigma_1\ \mathtt{END} \fatsemi \sigma_2 \\ \left(\begin{array}{l} \#\,\{i_1,\ ...,\ i_n\} = n \\ \sigma_2 = \lambda\ j : \operatorname{dom} \sigma_1 \bullet [i_1, ..., i_n]\ (\sigma_1\ j) \end{array}\right) \end{array}}$$

[7] Draft standard Z divides specifications into sections, each of which is a named sequence of paragraphs related to other sections [15].

Conjecture paragraph The predicate should be well-typed. The signature of the paragraph is empty.[8]

$$\frac{\Sigma \vdash^{\mathcal{P}} p}{\Sigma \vdash^{\mathcal{D}} \models ?\, p \ \mathtt{END} \ \text{⨾} \ \sigma} (\sigma = \varepsilon)$$

Generic conjecture paragraph The parameter names should all be different. The predicate can refer to the generic parameters. The signature of the paragraph is empty.

$$\frac{\Sigma \oplus \{i_1 \mapsto \mathbb{P}(\mathtt{GENTYPE}\ i_1),\ ...,\ i_n \mapsto \mathbb{P}(\mathtt{GENTYPE}\ i_n)\} \vdash^{\mathcal{P}} p}{\Sigma \vdash^{\mathcal{D}} [i_1, ..., i_n] \models ?\, p \ \mathtt{END} \ \text{⨾} \ \sigma}$$
$$\left(\begin{array}{l} \#\,\{i_1,\ ...,\ i_n\} = n \\ \sigma = \varepsilon \end{array}\right)$$

Predicate

Membership predicate The type of the right operand should be a powerset of the type of the left operand.

$$\frac{\Sigma \vdash^{\mathcal{E}} e_1 \ \text{⨾} \ \tau_1 \qquad \Sigma \vdash^{\mathcal{E}} e_2 \ \text{⨾} \ \tau_2}{\Sigma \vdash^{\mathcal{P}} (e_1 \ \text{⨾} \ \tau_1) \in (e_2 \ \text{⨾} \ \tau_2)} (\tau_2 = \mathbb{P}\,\tau_1)$$

Truth predicate This is always well-typed, hence there are no type subsequents.

$$\frac{}{\Sigma \vdash^{\mathcal{P}} \mathrm{true}}$$

Negation predicate

$$\frac{\Sigma \vdash^{\mathcal{P}} p}{\Sigma \vdash^{\mathcal{P}} \neg\, p}$$

Conjunction predicate

$$\frac{\Sigma \vdash^{\mathcal{P}} p_1 \qquad \Sigma \vdash^{\mathcal{P}} p_2}{\Sigma \vdash^{\mathcal{P}} p_1 \land p_2}$$

Universal quantification predicate The predicate should be well-typed in the environment overridden with the signature of the schema text.

$$\frac{\Sigma \vdash^{\mathcal{T}} t \ \text{⨾} \ \sigma \qquad \Sigma \oplus \sigma \vdash^{\mathcal{P}} p}{\Sigma \vdash^{\mathcal{P}} \forall\, t \ \text{⨾} \ \sigma \bullet p}$$

[8] Conjectures in draft standard Z are introduced by the $\models ?$ keyword.

Expression

Reference expression A reference expression can be a reference to a generic definition in which the instantiation has been left implicit. In that case, for the instantiations to be determined later (once all constraints have been solved), the uninstantiated type has to be remembered as well as the instantiated type. The instantiated type is denoted by juxtaposing the generic type $\Sigma\ i$ with a square bracketed list of variable types $[\alpha_1, ..., \alpha_n]$ that replace instances of corresponding generic parameter types.

$$\frac{}{\Sigma \vdash^{\mathcal{E}} i \mathbin{⦂} \tau}\left(\begin{array}{l} i \in \text{dom } \Sigma \\ \tau = \text{if } \Sigma\ i = [\imath_1, ..., \imath_n]\ \alpha \text{ then } \Sigma\ i, (\Sigma\ i)\ [\alpha_1, ..., \alpha_n] \text{ else } \Sigma\ i \end{array}\right)$$

Generic instantiation expression The name should be in the environment with a generic type. The instantiating expressions should be sets.

$$\frac{\Sigma \vdash^{\mathcal{E}} e_1 \mathbin{⦂} \tau_1 \quad ... \quad \Sigma \vdash^{\mathcal{E}} e_n \mathbin{⦂} \tau_n}{\Sigma \vdash^{\mathcal{E}} i[(e_1 \mathbin{⦂} \tau_1), ..., (e_n \mathbin{⦂} \tau_n)] \mathbin{⦂} \tau}\left(\begin{array}{l} i \in \text{dom } \Sigma \\ \Sigma\ i = [\imath_1, ..., \imath_n]\ \alpha \\ \tau_1 = \mathbb{P}\,\alpha_1 \\ \quad\vdots \\ \tau_n = \mathbb{P}\,\alpha_n \\ \tau = (\Sigma\ i)\ [\alpha_1, ..., \alpha_n] \end{array}\right)$$

Set extension expression The component type of a set can be constrained only if it has any members. Those members should be all of the same type.

$$\frac{\Sigma \vdash^{\mathcal{E}} e_1 \mathbin{⦂} \tau_1 \quad ... \quad \Sigma \vdash^{\mathcal{E}} e_n \mathbin{⦂} \tau_n}{\Sigma \vdash^{\mathcal{E}} \{(e_1 \mathbin{⦂} \tau_1), ..., (e_n \mathbin{⦂} \tau_n)\} \mathbin{⦂} \tau}\left(\begin{array}{l} \text{if } n > 0 \text{ then} \\ \qquad (\tau_1 = \tau_n \\ \qquad\quad \vdots \\ \qquad \tau_{n-1} = \tau_n \\ \qquad \tau = \mathbb{P}\,\tau_1) \\ \text{else } \tau = \mathbb{P}\,\alpha \end{array}\right)$$

Set comprehension expression The expression should be well-typed in the environment overridden with the signature of the schema text.

$$\frac{\Sigma \vdash^{\mathcal{T}} t \mathbin{⦂} \sigma \qquad \Sigma \oplus \sigma \vdash^{\mathcal{E}} e \mathbin{⦂} \tau_1}{\Sigma \vdash^{\mathcal{E}} \{t \mathbin{⦂} \sigma \bullet (e \mathbin{⦂} \tau_1)\} \mathbin{⦂} \tau_2}(\tau_2 = \mathbb{P}\,\tau_1)$$

Binding construction expression The expression should be a schema. Every name and type pair in its signature, with the optional decoration added, should be present in the environment, and the types should not be generic.

$$\frac{\Sigma \vdash^{\mathcal{E}} e \mathbin{⦂} \tau_1}{\Sigma \vdash^{\mathcal{E}} \theta\,(e \mathbin{⦂} \tau_1)\ ^{*} \mathbin{⦂} \tau_2}$$
$$\left(\begin{array}{l} \tau_1 = \mathbb{P}[\beta] \\ \tau_2 = [\beta] \\ \forall\ i : \texttt{NAME} \mid (i, \alpha_1) \in \beta \bullet (\mathit{decor}\ ^{*}\ i, \alpha_1) \in \Sigma \wedge \neg\, \alpha_1 = [\imath_1, ..., \imath_n]\ \alpha_2 \end{array}\right)$$

Schema conjunction expression The two expressions should be schemas with compatible signatures. Those signatures are merged in forming the type of the whole schema conjunction.

$$\frac{\Sigma \vdash^{\mathcal{E}} e_1 \mathbin{⨾} \tau_1 \qquad \Sigma \vdash^{\mathcal{E}} e_2 \mathbin{⨾} \tau_2}{\Sigma \vdash^{\mathcal{E}} (e_1 \mathbin{⨾} \tau_1) \land (e_2 \mathbin{⨾} \tau_2) \mathbin{⨾} \tau_3} \left(\begin{array}{l} \tau_1 = \mathbb{P}[\beta_1] \\ \tau_2 = \mathbb{P}[\beta_2] \\ \beta_1 \approx \beta_2 \\ \tau_3 = \mathbb{P}[\beta_1 \cup \beta_2] \end{array} \right)$$

Schema universal quantification expression The expression should be a schema whose signature is compatible with that of the schema text. Those signatures are subtracted in forming the type of the whole schema universal quantification.

$$\frac{\Sigma \vdash^{\mathcal{T}} t \mathbin{⨾} \sigma \qquad \Sigma \oplus \sigma \vdash^{\mathcal{E}} e \mathbin{⨾} \tau_1}{\Sigma \vdash^{\mathcal{E}} \forall\, t \mathbin{⨾} \sigma \bullet (e \mathbin{⨾} \tau_1) \mathbin{⨾} \tau_2} \left(\begin{array}{l} \tau_1 = \mathbb{P}[\beta] \\ \sigma \approx \beta \\ \tau_2 = \mathbb{P}[\mathrm{dom}\ \sigma \mathbin{⩤} \beta] \end{array} \right)$$

Schema text and declaration

Schema text The declarations should have pairwise compatible signatures. The predicate should be well-typed in the environment overridden by the merging of those signatures. Duplicate declarations of the same names are thus permitted.

$$\frac{\Sigma \vdash^{\mathcal{DE}} de_1 \mathbin{⨾} \sigma_1 \qquad \ldots \qquad \Sigma \vdash^{\mathcal{DE}} de_n \mathbin{⨾} \sigma_n \qquad \Sigma \oplus \sigma \vdash^{\mathcal{P}} p}{\Sigma \vdash^{\mathcal{T}} de_1;\ \ldots;\ de_n \mid p \mathbin{⨾} \sigma} \left(\begin{array}{l} \sigma_1 \approx \sigma_2\ \ldots\ \sigma_1 \approx \sigma_n \\ \vdots \\ \sigma_{n-1} \approx \sigma_n \\ \sigma = \sigma_1 \cup \ldots \cup \sigma_n \end{array} \right)$$

Variable declaration The expression should be a set. The signature of the declaration is formed from the names, amongst which there can be duplicates.

$$\frac{\Sigma \vdash^{\mathcal{E}} e \mathbin{⨾} \tau}{\Sigma \vdash^{\mathcal{DE}} i_1, \ldots, i_n : e \mathbin{⨾} \sigma} \left(\begin{array}{l} \tau = \mathbb{P}\,\alpha \\ \sigma = \{(i_1,\ \alpha)\} \cup \ldots \cup \{(i_n,\ \alpha)\} \end{array} \right)$$

Variable definition

$$\frac{\Sigma \vdash^{\mathcal{E}} e \mathbin{⨾} \tau}{\Sigma \vdash^{\mathcal{DE}} i == e \mathbin{⨾} \sigma} (\sigma = i : \tau)$$

Inclusion declaration The expression should be a schema.

$$\frac{\Sigma \vdash^{\mathcal{E}} e \mathbin{⨾} \tau}{\Sigma \vdash^{\mathcal{DE}} e \mathbin{⨾} \sigma} (\tau = \mathbb{P}[\sigma])$$

4.3 Type Inference System

The type inference system applies type inference rules backwards (relative to the way the notation was described in section 4.1): the type sequent is viewed as a pattern, and the associations of metavariables with values produced by matching that pattern are used to instantiate the type subsequents and constraints.

For the patterns to match, there must already be annotations on all formulæ, excepting predicates as they have none. These annotations can be all distinct variables, except as required by draft standard Z (namely that all instances of expressions duplicated by its transformations of chained relations and comma-separated declarations should have the same types).

Starting with a type sequent for a whole Z specification, the type inference rule for specification is applied to it, producing one type subsequent for each section, and some constraints to determine the environments to be used in type-checking those sections. There is no need to solve the constraints yet. Instead, type inference rules can be applied to the generated type subsequents, each application producing zero or more new type subsequents, until no more type subsequents remain. Termination is guaranteed by the finiteness of the original specification, and the fact that in every type inference rule the type subsequents involve only sub-formulæ of the type sequent's Z phrase.

This leaves a set of constraints to be solved. There are dependencies between constraints: for example, a constraint that checks that a name is declared in an environment cannot be solved until that environment has been determined by other constraints. As another example, references to generics generate a constraint involving the operation of generic type instantiation, which should not be performed until the type of the referenced generic has been determined. This can be ensured by solving the constraints in per-paragraph batches, as generics are defined at top-level and instantiated only in subsequent paragraphs.

Unification is a suitable mechanism for solving constraints. For a well-typed specification, it is possible to solve all the constraints. The resulting unifier provides values for the variables in the constraints. For draft standard Z, every annotation's original variable should be replaced by the value to which it has been constrained. For a specification to be well-typed, no variables should remain within any of those values.

4.4 Implicit Instantiations

Once a paragraph has been typechecked, the instantiations of its uninstantiated references to generics can be made explicit. This can be expressed formally by the following rule, which transforms a reference expression with a pair of annotations to a generic instantiation expression.

$$i \mathbin{⦂} [i_1, \ldots, i_n]\ \tau, \tau' \implies i\ [\mathit{carrier}\ \alpha_1, \ldots, \mathit{carrier}\ \alpha_n] \mathbin{⦂} \tau'$$

$$\mathit{where}\ \tau' = ([i_1, \ldots, i_n]\ \tau)\ [\alpha_1, \ldots, \alpha_n]$$

The instantiating expressions are the carrier sets of the types inferred for the generic parameters. Those types α_1, ..., α_n are determined by comparison of the generic type $[i_1, ..., i_n]\ \tau$ with the instantiation of it τ'.

4.5 Schemas

The well-typed though awkward schemas discussed in the requirements section can all be accepted by a typechecker as specified above. They could not have been accepted if the typechecker had instead attempted to solve constraints during a recursive traversal of Z phrases.

4.6 Browsing

Typechecking is based on signatures, which comprise just names and types, yet a browser needs to know about specific declarations. The names and types in a signature originate from the variable declarations of a schema text. So a set of variable declarations can serve as a representation of a signature. When a name is looked-up in an environment, a declaration can be returned rather than just a type.

The requirement that inclusion declarations introduce new variable declarations distinct from those of the included schema is a complication for this scheme. Our typechecker defers this copying of declarations until after typechecking has finished. When a reference expression is typechecked, as well as noting the declaration to which it is bound, we also note the schema text which put that declaration into scope. A traversal of the specification after typechecking can then find all schema texts, make distinct copies of included declarations, and find all reference expressions and rebind them to the new declarations. To support this, every overriding of an environment by a signature is annotated with the corresponding schema text.

If a typechecker notes the declarations of all uses, including all implicit ones, then a browser has all the information needed to determine the uses of all declarations.

Knowing the declaration referred to by a reference expression helps in the process of filling in implicit instantiations: the original uninstantiated type need not be remembered on the reference expression, as it can be retrieved from the referenced generic definition.

4.7 Draft Standard Z

The requirements of draft standard Z on the specification of the type system have largely been addressed by the above specification. One difference is that we have chosen to give type inference rules for schema texts and declarations, whereas those are transformed away earlier in draft standard Z. The choice made here involves more type inference rules, but generates fewer constraints elsewhere.

4.8 Undecoration Expressions

To support undecoration expressions [21], the following changes to the above specification are needed.

Change to Z syntax Undecoration expressions are written using the *undecor* keyword and specify the stroke of the components to be extracted.

```
Expr = ... all existing productions ...
     | 'undecor' , STROKE , Expr
     ;
```

Change to Z typechecker The new undecoration expressions need a type inference rule.

Undecoration expression The expression should be a schema. Every name and type pair in the schema's signature where the name's last stroke matches the given one, is present in the result with that stroke removed.

$$\frac{\Sigma \vdash^{\mathcal{E}} e \,⨾\, \tau_1}{\Sigma \vdash^{\mathcal{E}} \mathit{undecor}^{\,+}\ (e \,⨾\, \tau_1) \,⨾\, \tau_2} \left(\begin{array}{l} \tau_1 = \mathbb{P}[\beta_1] \\ \tau_2 = \mathbb{P}[\beta_2] \\ \beta_2 = \{ i : \mathtt{NAME} \mid (\mathit{decor}^{\,+}\ i,\ \alpha) \in \beta_1 \bullet (i,\ \alpha) \} \end{array} \right)$$

Semantics of undecoration expressions The semantic value of an undecoration expression is the set of bindings that is like that of the operand schema but without those components whose names do not have the given stroke and with that stroke removed from the retained names.

4.9 Type-Constrained Generics

To support type-constrained generics [21], the following changes to the above specification are needed.

Change to Z syntax Generic parameter lists can have a dagger, which precedes those parameters that are constrained to be schemas.

```
Fmls = [ NAME , { ',' , NAME } ] , [ '†' , NAME , { ',' , NAME } ] ;
```

Although the † notation has been introduced in formal parameter lists, and is introduced below in generic types, we do not introduce it in explicit generic instantiation lists, which just use , (comma) between instantiating expressions.

Changes to Z typechecker The notation for generic types needs to list the names of the new parameters, for use in determining implicit instantiations.

```
Type = '[' , [ NAME , { ',' , NAME } ] , [ '†' , NAME , { ',' , NAME } ] , ']' , Type ,
           [ ',' , Type ]                                    (* generic type *)
     | ... other productions as before ...
     ;
```

There should be at least one `NAME` in a generic type, despite this syntax not requiring that.

Some additional notation is needed for signatures.

```
Sig  = ... all existing productions ...
     | 'GENSIG' , NAME                 (* generic parameter signature *)
     | Sig , '∪' , Sig                          (* merged signature *)
     ;
```

The `GENSIG` notation is somewhat analogous to `GENTYPE`, but with the difference that a generic definition can impose constraints on a `GENSIG`. That difference makes `GENSIG` seem like the variable signature notation, but when these notations appear in environments, they are interpreted differently (see below). We also restrict the constraints on generic parameter signatures: we allow compatibility constraints, but reject unification constraints.

The $\cup$ notation denotes the signature formed by merging two signatures. The $\cup$ symbol that has already been used in some of the above type inference rules was an operator of set theory: the constraints in which $\cup$ was used were regarded as solvable only when its operands were known signatures. Those uses of $\cup$ can be regarded as uses of the new signature notation, allowing some of those constraints to be solved sooner.

All type inference rules concerned with generics need to be revised, as follows.

Generic axiomatic description paragraph Generic parameter signatures are treated much like generic parameter types by this rule, the difference being that they have different types in the environment.

$$\frac{\begin{array}{l}\Sigma \oplus \{i_1 \mapsto \mathbb{P}(\texttt{GENTYPE}\ i_1),\ ...,\ i_n \mapsto \mathbb{P}(\texttt{GENTYPE}\ i_n),\\ \qquad j_1 \mapsto \mathbb{P}[\texttt{GENSIG}\ j_1],\ ...,\ j_m \mapsto \mathbb{P}[\texttt{GENSIG}\ j_m]\} \vdash^{\mathcal{T}} t \mathbin{⨾} \sigma_1\end{array}}{\Sigma \vdash^{\mathcal{D}} \texttt{GENAX}\ [i_1, ..., i_n \dagger j_1, ..., j_m]\ t \mathbin{⨾} \sigma_1\ \texttt{END} \mathbin{⨾} \sigma_2}\left(\begin{array}{l}\#\,\{i_1,\ ...,\ i_n,\ j_1,\ ...,\ j_m\} = n + m\\ \sigma_2 = \lambda\ j : \operatorname{dom}\sigma_1 \bullet [i_1, ..., i_n \dagger j_1, ..., j_m]\ (\sigma_1\ j)\end{array}\right)$$

Generic conjecture paragraph

$$\frac{\begin{array}{l}\Sigma \oplus \{i_1 \mapsto \mathbb{P}(\texttt{GENTYPE}\ i_1),\ ...,\ i_n \mapsto \mathbb{P}(\texttt{GENTYPE}\ i_n),\\ \qquad j_1 \mapsto \mathbb{P}[\texttt{GENSIG}\ j_1],\ ...,\ j_m \mapsto \mathbb{P}[\texttt{GENSIG}\ j_m]\} \vdash^{\mathcal{P}} p\end{array}}{\Sigma \vdash^{\mathcal{D}} [i_1, ..., i_n \dagger j_1, ..., j_m] \models ?\ p\ \texttt{END} \mathbin{⨾} \sigma}\left(\begin{array}{l}\#\,\{i_1,\ ...,\ i_n,\ j_1,\ ...,\ j_m\} = n + m\\ \sigma =\end{array}\right)$$

Generic instantiation expression The instantiations of schema parameters should be schemas.

$$\frac{\Sigma \vdash^{\mathcal{E}} e_1 ⦂ \tau_1 \quad \dots \quad \Sigma \vdash^{\mathcal{E}} e_n ⦂ \tau_n \qquad \Sigma \vdash^{\mathcal{E}} e'_1 ⦂ \tau'_1 \quad \dots \quad \Sigma \vdash^{\mathcal{E}} e'_m ⦂ \tau'_m}{\Sigma \vdash^{\mathcal{E}} i[(e_1 ⦂ \tau_1), \dots, (e_n ⦂ \tau_n), (e'_1 ⦂ \tau'_1), \dots, (e'_m ⦂ \tau'_m)] ⦂ \tau}$$

$$\left(\begin{array}{l} \alpha = \mathit{lookup}\ i\ \Sigma \\ \alpha = [\imath_1, \dots, \imath_n \dagger \jmath_1, \dots, \jmath_m]\ \alpha' \\ \tau_1 = \mathbb{P}\,\alpha_1 \\ \qquad \vdots \\ \tau_n = \mathbb{P}\,\alpha_n \\ \tau'_1 = \mathbb{P}[\beta_1] \\ \qquad \vdots \\ \tau'_m = \mathbb{P}[\beta_m] \\ \tau = \alpha\ [\alpha_1, \dots, \alpha_n, \beta_1, \dots, \beta_m] \end{array}\right)$$

The *lookup* operation is described below.

Reference expression

$$\frac{}{\Sigma \vdash^{\mathcal{E}} i ⦂ \tau}$$

$$\left(\begin{array}{l} \alpha = \mathit{lookup}\ i\ \Sigma \\ \tau = \text{if } \alpha = [\imath_1, \dots, \imath_n \dagger \jmath_1, \dots, \jmath_m]\ \alpha' \text{ then } \alpha, \alpha\ [\alpha_1, \dots, \alpha_n, \beta_1, \dots, \beta_m] \text{ else } \alpha \end{array}\right)$$

The rule for binding construction expression needs to be revised analogously.

For draft standard Z, it is possible to solve all the constraints relating to a paragraph of a (well-typed) specification before proceeding to the next paragraph. With type-constrained generics, some constraints might not be solvable then. A counterexample is the explicit definition of schema conjunction, which imposes a constraint of compatibility between the signatures of its instantiating schemas. Having typechecked a paragraph, any remaining constraints should be noted as an attribute of that paragraph.

These unsolved constraints affect the check that all implicit instantiations are uniquely determined. The check cannot be delayed until the constraints are solved, as the declarations of the paragraph might never be used, and even if they are they might be used in type erroneous ways, so we continue to perform the check after typechecking each paragraph. Where an implicit instantiation is in a paragraph that has some parameters that must be schemas, and there are some unsolved constraints on which the implicit instantiation depends, we have assumed that the implicit instantiation will become uniquely determined when the paragraph's parameters are instantiated.

Constraints that involve looking up a name in an environment viewed the environment as a function, requiring any uses of the $\oplus$ notation in forming the environment to have been evaluated before the constraint doing the look up could be solved. With the extensions, environments can now contain generic parameter signatures, and so cannot all be evaluated. Hence the introduction of the *lookup* operation, which behaves as follows.

If the environment is a known signature, then look up proceeds as before. If the environment is a variable signature, look up cannot succeed, and the constraint doing the look up will have to be solved later. If the environment is a generic parameter signature, look up behaves as if the requested name is not defined in this environment. This solves the problem exemplified by the definition of schema projection in the requirements above. Overridden environments and merged signatures cause look up to recurse appropriately.

This special treatment of generic parameter signatures in the environment is what restricts our type-constrained generics to being schemas. Relaxing that restriction would be nice, but we have been unable to find a way of doing so that also provides a solution to the name-space problem.

The *lookup* operation needs to return not just the inferred type but also any unresolved constraints from typechecking of a generic definition; these constraints are instantiated appropriately and added to the collection of constraints yet to be solved. Those unresolved constraints are also relevant to a browser: when displaying the type of a formula in a constrained generic definition, any unresolved constraints should be revealed.

Changes to implicit instantiations Instantiating expressions are needed not just for the generic parameters but also for the parameters that are constrained to be schemas.

$$i \mathbin{⨾} [i_1,\ ...,\ i_n \dagger j_1,\ ...,\ j_m]\ \tau, \tau'$$

$$\Longrightarrow$$

$$i\ [\mathit{carrier}\ \alpha_1, ..., \mathit{carrier}\ \alpha_n,\ \mathit{carriersig}\ \beta_1, ..., \mathit{carriersig}\ \beta_m] \mathbin{⨾} \tau'$$

$$\mathit{where}\ \tau' = ([i_1,\ ...,\ i_n \dagger j_1,\ ...,\ j_m]\ \tau)\ [\alpha_1,\ ...,\ \alpha_n,\ \beta_1,\ ...,\ \beta_m]$$

The carrier set of a schema type continues to be a schema construction expression, but we can no longer assume that the declarations within it are all variable declarations. Instead we need *carriersig* to generate an appropriate list of declarations. The carrier of a generic parameter signature is an inclusion declaration referring to the generic parameter. The carrier of a merged signature is the concatenation of the declarations that are the carriers of its operands. Beware that a schema construction expression with only one declaration that is an inclusion does not conform to draft standard Z syntax; the square brackets should be dropped in that case.

Semantics of type-constrained generics Draft standard Z's semantic equation for generic axiomatic paragraph creates models for all set-valued instantiations of the generic parameters. It should be extended to consider all schema-valued instantiations of the parameters that should be schemas and that conform to the constraints on those instantiations.

4.10 Diagnosing Type Errors

An aim was to reject all ill-typed Z specifications, but also legible error reports should be provided when mistakes are detected. Mistakes are detected as invalid constraints. Since every constraint arises from the application of a type inference rule to a particular phrase, this allows mistakes to be attributed to corresponding phrases. For each invalid constraint, we provide the specifier with that constraint (paraphrased to some extent), identification of the corresponding phrase, and also the ability to browse the specification to see what the typechecker inferred.

Where constraints are independent of one another, they can be solved in any order. The order in which they are solved affects the phrases to which any mistakes are attributed. It is worth solving constraints that would be difficult to diagnose if invalid before solving independent constraints that would be easier to diagnose if invalid. For example, operator applications are transformed to involve tuples of operands before being typechecked; it is better to diagnose an operator as being of inappropriate arity than to say that the tuple of operands is of inappropriate size, as that tuple is not a separate phrase visible to the specifier.

5 Conclusions

It is possible to typecheck even contrived Z specifications, so long as the implementation of the typechecker does not impose extra constraints, such as on the order in which constraints are expected to be solved. A typechecker can assist browsing and reasoning tools by determining where variables are used. We have given a specification of a typechecker in a form that might be suitable for draft standard Z. A typechecker for draft standard Z can be extended to handle type-constrained generics, and hence explicit definitions of schema calculus operators, without any backwards incompatibilities.

Acknowledgements

Rob Arthan set us thinking about awkward schemas. Funding for this work was provided by EPSRC grant GR/M20723.

References

1. L. Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8(2):147–172, April 1987.
2. P. Hancock. Polymorphic type-checking. In S.L. Peyton Jones, editor, *The Implementation of Functional Programming Languages*, 1987.
3. ISO/IEC 14977:1996(E). *Information Technology—Syntactic Metalanguage—Extended BNF.*
4. Xiaoping Jia. ZTC: A type checker for Z notation, user's guide. Technical Report Version 2.03, Division of Software Engineering, School of Computer Science, Telecommunication, and Information Systems, DePaul University, August 1998.
5. L. Lamport and L.C. Paulson. Should your specification language be typed? *Transactions on Programming Languages and Systems*, 21(3):502–526, May 1999.

6. R. Milner. A theory of type polymorphism in programming languages. *Journal of Computer and System Science*, 17:348–357, 1978.
7. D. Neilson. Machine support for Z: the zedB tool. In *Proceedings of the 5th Z User Meeting*, 1990.
8. J.N. Reed and J.E. Sinclair. An algorithm for type-checking Z. Technical Monograph PRG-81, Oxford University Computing Laboratory, Programming Research Group, March 1990.
9. C.T. Sennett. Review of the type checking and scope rules of the specification language Z. Technical Report 87017, Royal Signals and Radar Establishment, Malvern, November 1987.
10. J.M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics.* Cambridge University Press, 1988.
11. J.M. Spivey. *The Z Notation: A Reference Manual, 2nd editon.* Prentice Hall, 1992.
12. J.M. Spivey and B.A. Sufrin. Type inference in Z. In D. Bjørner, C.A.R. Hoare, and H. Langmaack, editors, *VDM'90: VDM and Z—Formal Methods in Software Development*, LNCS 428, pages 426–451. Springer, 1990.
13. S. Stepney. Formaliser Home Page. http://public.logica.com/~formaliser/.
14. B. Sufrin. Using the Hippo system. Technical report, Oxford University Computing Laboratory, Programming Research Group, June 1989.
15. I. Toyn. Innovations in the notation of standard Z. In *ZUM'98: The Z Formal Specification Notation*, LNCS 1493. Springer, September 1998.
16. I. Toyn, editor. *Z Notation: Final Committee Draft.* http://www.cs.york.ac.uk-/~ian/zstan/fcd.ps, August 1999.
17. I. Toyn. CADiZ web pages. http://www.cs.york.ac.uk/~ian/cadiz/, 2000.
18. I. Toyn and J.A. McDermid. CADiZ: An architecture for Z tools and its implementation. *Software — Practice and Experience*, 25(3):305–330, March 1995.
19. I. Toyn and S.H. Valentine. Type inference rules for Z. ftp://ftp.cs.york.ac.uk/hise_reports/cadiz/ZSTAN/rules.ps, March 2000.
20. I. Toyn, S.H. Valentine, and D.A. Duffy. On mutually recursive free types in Z. In *ZB2000: International Conference of B and Z Users*, 2000.
21. S.H. Valentine, I. Toyn, S. Stepney, and S. King. Type-constrained generics. In *ZB2000: International Conference of B and Z Users*, 2000.