A Formal Description of an Incremental Type-Checker for Z

Alexandre M. L. de Vasconcelos (amlv@di.ufpe.br)

Departamento de Informática, Universidade Federal de Pernambuco

Caixa Postal 7851, 50732-970, Recife, PE, Brasil.

Abstract

In this paper, we describe some of the difficulties that must be tackled when type-checking Z [1] specifications on an incremental basis. We formalise, in Z itself, the possible dependency relationships for each kind of Z definition. Then, we present an extensive list of issues that an incremental type-checking algorithm for Z must deal with, as well as an outline specification of an incremental type-checking algorithm which deals with these issues.

Keywords: Incremental Type-Checking, Formal Specification, Z Notation.

1. INTRODUCTION

Re-typechecking mechanisms are desirable in order to re-evaluate the type environment and the type consistency of the specification when a definition is edited inside a specification.

In specification/programming environments which use batch type-checking algorithms, the amount of re-typechecking (A_r) is proportional to the size (n) of the specification/program (i.e. $A_r \alpha\ n$), because the whole specification/program is processed from scratch. For simplicity, we can say that A_r is given by the result of a function *Retype* applied to the size of the specification/program (i.e. A_r = *Retype(n)*).

Theoretically, the amount of re-typechecking in incremental environments is proportional to the size of the change (i.e. $Retype_{min}(c)$, where c is the size of the change). The actual cost *Retype(c)* will be determined by the complexity of dependencies etc. In some cases, *Retype(c) = Retype(n)* (e.g. redefining a global variable that is used by all the segments of the specification/program will cause re-typechecking from scratch).

The main technical issue, in incremental environments, is identifying what sub-set of the specification/program needs to be re-checked after a change. In practice, the computational costs of the mechanisms for incremental checking (e.g. traversing a dependency graph) may outweigh the benefits.

2. A STRATEGY FOR INCREMENTAL TYPE-CHECKING IN Z

Incremental type-checking algorithms are based on the "observation that if a definition does not use a modified definition, either directly or indirectly, then its type cannot be affected by changes in the type of the modified definition" [2]. Consequently, after modifying a definition, only the definitions which use (i.e. depend on) it need to be re-typechecked. In fact, the dependent definitions do not need to be re-typechecked if the underlying type of the modified definition remains the same, because type errors will not be introduced in these dependent definitions. The incremental type-checking strategy which we propose is more flexible. However, before we present the strategy, we need to introduce some terminology.

2.1 Terminology

We introduce the concepts of "signature" and "sub-signature", which will be used in sub-section 2.2, when explaining our incremental type-checking strategy.

2.1.1 Signature

The signature of a Z definition is the set of identifiers that it introduces, each with its type [1]. Unlike other descriptions of Z, all our Z definitions have signature, not just schema definitions.

2.1.2 Sub-Signature

A signature Σ_1 is a sub-signature of a signature Σ_2 if the identifiers of Σ_1 are also in Σ_2, and the type of each corresponding identifier in both signatures is the same [3]. This notion of "sameness" of types needs to be defined carefully when dealing with generic type parameters, because two generic type parameters may express the same type even if they have different names. For instance, in the following example: if we modify the Sch on the left-hand side to give the version on the right-hand side

```
┌─ Sch [X] ──        ┌─ Sch [Y] ──
│ x : X              │ x : Y
└──────────          └──────────
```

X and Y should be considered to be the same generic type. Hence, a generic type G_1 is the same as another generic type G_2, if there is a substitution S from identifiers to identifiers, so that $G_1 = SG_2$ (i.e. when applying S to G_2 makes G_1 equal to G_2).

2.2 The Strategy

Our incremental type-checking algorithm is based on the observation that when a definition is modified, type errors are not introduced in a specification if the signature of the definition is extended with new identifiers and the underlying types of the identifiers in the previous signature are still the same. For instance, if a given −set[1] definition:

```
[A, B]
```

which has signature $\{A \mapsto \mathbb{P}\, A, B \mapsto \mathbb{P}\, B\}$ is replaced by another given-set definition:

```
[A, B, C]
```

which has signature $\{A \mapsto \mathbb{P}\, A, B \mapsto \mathbb{P}\, B, C \mapsto \mathbb{P}\, C\}$, type errors are not introduced into the specification (although there may be scope errors — e.g., if C is already defined later in the specification), because A and B are still in scope after the definition is modified and they have the same underlying types as before.

The relationship between those two given-set definitions is very similar to the subtype relationship between structure (or record) types in object-oriented systems as presented in [4] (i.e. "a subtype structure can have more, but not fewer fields than the supertype"). Similarly, the signature of the first given-set definition is a sub-signature of the signature of the second given-set definition. Based on this observation, we can define a similar relationship between definitions which we will refer to as *subtype*:

A definition *A* is a *subtype* of a definition *B* iff for every identifier introduced in definition *B* there is an equivalent identifier (with the same type) introduced in definition *A*.

Hence, according to our definition of *subtype*, the second given-set definition [A, B, C], introduced above, is a *subtype* of the first given-set definition [A, B], and so subsequent definitions depending on the first definition do not need to be re-typechecked. But later definitions that already define C do.

3. A FORMAL MODEL OF INCREMENTAL TYPE-CHECKING IN Z

In this section, we describe the most important issues that an incremental type-checking algorithm for Z must deal with. First we present some preliminary concepts and their formal specification in Z itself. Then, we formalise the *subtype* relationship. Finally, we specify the dependencies between definitions of a specification in terms of affected definitions due to the editing (i.e. insertion, modification, deletion, or transference) of a specific definition inside a specification.

Only the aspects related to the internal dependencies of a specification are specified here, as the treatment of external dependencies would imply that a large and complex specification of the mechanisms for version control and configuration management should be given.

We do not specify details of the type-checking of a single definition, since this is dealt with by a ML-like algorithm similar to the one described in [5]. Readers interested in a Z specification of such an algorithm for type-checking Z definitions should refer to [6, 7].

3.1 Preliminaries

We assume the existence of names for distinguishing identifiers, expressions used in declarations of identifiers, definitions given inside a specification and logical predicates.

```
[Name, Exp, Def, Pred]
```

In our model we use the standard decorations for identifiers [1].

```
Decor ::= ? | ! | '
```

Hence, an identifier has a name and one or more optional decorations.

```
┌ Id ──────────────────
│ name : Name
│ decors : seq Decor
│
└──────────────────────
```

3.1.1 The Representation of Types

The type of each identifier may be any of the types described in [1] and defined below. It is interesting to discuss the representation of the types related to polymorphism in Z. A type parameter (parT) corresponds to each occurrence of a generic identifier in a generic type (genT). As the instantiation of the parameters

1 A given-set definition introduces types defined in a non-constructive way (also called "parachuted" types).

of a generic type depends on the order in which the instantiated types are given. we represent a generic type by using a sequence of identifiers (the generic parameters) and the generic type itself. It is also necessary to represent type variables (varT) which are used for anonymous (implicit) instantiations (i.e. they appear in a type substituted for the formal generic parameters when a generic type is implicitly instantiated [6]).

```
Type ::=
    givenT «Id» | powerT «Type» | tupleT «seq Type» | schemaT «Id ⇸ Type» |
    parT «Id» | genT «(seq Id × Type)» | varT «Id»
```

3.1.2 Signature

As explained in sub-section 2.1.1, a signature is a mapping from identifiers to their most general types. Notice that a schema type as specified above is built from the signature of the corresponding schema.

```
Sig == Id ⇸ Type
```

3.1.3 Sub-Signature

In order to specify the concept of sub-signature explained in sub-section 2.1.2. we first specify an identifiers' substitution. which is a mapping from identifiers to identifiers.

```
Subst == Id ⤔ Id
```

Notice that Subst is represented as a partial injection to ensure that an "abnormal" generic type as genT ((⟨ W, W ⟩), powerT(tupleT ⟨ parT W, parT W ⟩)) is not considered the same as the generic type genT ((⟨ X, Y ⟩), powerT(tupleT ⟨ parT X, parT Y ⟩)) through the application of a substitution (e.g. S = [W/X ,W/Y][2] to the second generic type). We also specify an instance of the identity relation which must be applied to identifiers.

```
Ident == id [Id]
```

A function type_subst applies a substitution to a type returning a modified type in which all the identifiers of type parameters have been replaced by their corresponding identifiers in the substitution. The application of type_subst to a type parameter either returns the type parameter itself or another type parameter which is associated with the original type parameter through the substitution. The application of type_subst to a generic type replaces all the type parameters occurring in the generic type by their corresponding type parameters in the substitution. The other cases are defined recursively on the structure of the other types. except the given-set types and the type variables which are not affected by the substitution.

```
| type_subst : Subst → Type → Type
|---------------
| ∀ subst : Subst; i : Id; t : Type; ids : seq Id; ts : seq Type; sig : Sig •
|     ∃ tsub : Type → Type | tsub = type_subst subst •
|         type_subst subst (parT i) = parT ((Ident ⊕ subst) i) ∧
|         type_subst subst (genT (ids, t)) = genT ((Ident ⊕ subst) ∘ ids, tsub t) ∧
|         type_subst subst (givenT i) = givenT i ∧ type_subst subst (varT i) = varT i ∧
|         type_subst subst (powerT t) = powerT (tsub t) ∧
|         type_subst subst (tupleT ts) = tupleT (tsub ∘ ts) ∧
|         type_subst subst (schemaT sig) = schemaT (tsub ∘ sig)
```

Now the sub-signature relation (which we denote by $\sqsubseteq_\sigma$) can be specified as follows.

---

2 We use this notation for a substitution by analogy with renaming of schema components in Z [1].

$$\_ \sqsubseteq_{\sigma} \_ : Sig \leftrightarrow Sig$$

$$\forall \Sigma_1, \Sigma_2 : Sig \bullet \Sigma_1 \sqsubseteq_{\sigma} \Sigma_2 \Leftrightarrow (\exists\, subst : Subst \bullet \Sigma_1 \subseteq type\_subst\ subst \circ \Sigma_2)$$

3.1.4 The System State

To specify the system state, we need to introduce a given-set representing unique identifiers for definitions.

[Defid]

These identifiers can distinguish definitions which have the same "structure" (e.g. two schema definitions which have the same name and one is a re-definition of the other).

The environment for checking definitions is specified as follows. There is a store which maps each definitions' identifier to the corresponding definition. A specification (spec) in our model corresponds to a "root file" (i.e. a sequence of identifiers for definitions which are in store). The relation visible_ids records all the visible identifiers of each definition (i.e. the global identifiers introduced by a definition and the local identifiers introduced in the declaration part of schemas). The relation uses records the visible identifiers of other definitions which each definition uses. A definition d_1 uses a definition d_2 if any of the visible identifiers of d_2 is used in d_1. A dependency graph, represented by the relation depends_on, is used to keep track of dependencies between definitions. An auxiliary relation called before is true if a definition d_2 comes before a definition d_1. Hence, a definition d_1 depends_on a definition d_2 if d_1 uses d_2, and d_2 comes before d_1 in the sequence of definitions.

Env_Spec

$$
\begin{array}{l}
store : Defid \rightarrowtail\!\!\!\!\!\!+\ Def \\
spec : \mathrm{seq}\ Defid \\
visible\_ids : Defid \leftrightarrow Id \\
\_\ uses\ \_ : Defid \leftrightarrow (Defid \times Id) \\
\_\ before\ \_ : Defid \leftrightarrow Defid \\
\_\ depends\_on\ \_ : Defid \leftrightarrow Defid \\
\hline
\mathrm{ran}\ spec \subseteq \mathrm{dom}\ store \\
\mathrm{ran}\ (\_\ uses\ \_) \subseteq visible\_ids \\
\forall d1, d2 : \mathrm{ran}\ spec \bullet \\
\quad (d2\ before\ d1 \Leftrightarrow (\exists i, j : \mathbb{N}_1 \bullet spec\ i = d2 \wedge spec\ j = d1 \wedge i < j)) \wedge \\
\quad (d1\ depends\_on\ d2 \Leftrightarrow (\exists i : Id \bullet d1\ uses\ (d2, i) \wedge d2\ before\ d1))
\end{array}
$$

3.2 The Subtype Relationship

Due to the lack of space, we only specify the *subtype* relationship for given-sets and schema definitions. For both kinds of definition we "enrich" the structure of Def, introduced as a given-set declaration in subsection 3.1. Then we extend the schema[3] representing the environment according to the properties of the specified Z definition, and finally we specify the meaning of the *subtype* relationship related to the specified definition.[4] The specification of *subtype* for a general definition is given by combining the specifications of the *subtype* relationship for each kind of Z definition. The reader is referred to [9] for a full specification.

3.2.1 Given-Set Definitions

A given-set definition introduces a sequence of identifiers for given-sets (gsets). We specify gsets as a sequence of identifiers because we admit that these identifiers may be instantiated positionally if the definitions of a specification are imported into another specification.

3 Schema extension is a feature which was present in the early versions of Z [8]. It has been preserved in our model as it is often useful for introducing a concept incrementally.

4 We admit that definitions of different kinds cannot be in the same *subtype* relation.

```
GSet_Def
  gsets : seq₁ Id
  ─────────
  ran gsets ⊆ dom givenT
```

The structure of Def may be enriched as:

```
Def ::= gset «GSet_Def» | ...
```

Notice that we use ... to mean that the definition of Def is not complete yet.

The signature of a given-set definition is built by mapping each given-set identifier onto the powerset of its given-set type [6].

```
  mkgivdef_sig : GSet_Def → Sig
  ─────────
  ∀ g : GSet_Def • mkgivdef_sig g = {i : ran g.gsets • i ↦ powerT (givenT i)}
```

The visible identifiers of each given-set definition in a specification are all the identifiers which the definition introduces.

```
Env_Spec
  Env_Spec
  ─────────
  ∀ defid : ran spec; g : GSet_Def | gset g = store defid •
      visible_ids⦇{defid}⦈ = ran g.gsets
```

A given-set definition g_new is a *subtype* of g_old, if the signature of g_old is a sub-signature of the signature of g_new. The order in which declarations are introduced in both given-set definitions is not important for internal dependencies.

```
  _ sub_gset _ : GSet_Def ↔ GSet_Def
  ─────────
  ∀ g_new, g_old : GSet_Def •
      ∃ new_sig, old_sig : Sig |
          new_sig = mkgivdef_sig g_new ∧ old_sig = mkgivdef_sig g_old •
              g_new sub_gset g_old ⇔ old_sig ⊆ₛ new_sig
```

In fact it would be sufficient only to verify that the set of visible identifiers introduced by g_old is a subset of the set of visible identifiers introduced by g_new, because if two given-set declarations have the same identifier they will consequently have the same type (which is the powerset of the given-set type).

3.2.2 Schema Definitions

Before we define the *subtype* relationship for schemas, we need to introduce some other concepts. First we introduce the concept of declaration as being a list of identifiers and an expression to define their type.

```
Decl
  id_list : seq₁ Id
  exp : Exp
  ─────────
```

We assume the existence of the function typeof which is a ML-like type-checking algorithm. It is assumed that **typeof** has access to all the definitions up to (and including) the definition in which the expression to be type-checked is found. Examples of specifications of this function may be found in [6,3,7].

$$\mathsf{typeof} : (\mathrm{seq}\ \mathsf{Def} \times \mathsf{Exp}) \rightarrow \mathsf{Type}$$

The function **defs_before** will be used later to generate a sequence of definitions which will be used as an argument to the function **typeof**.

$$\begin{array}{l}
\mathsf{defs_before} : (\mathsf{Defid} \times \mathrm{seq}\ \mathsf{Defid}) \rightarrowtail \mathrm{seq}\ \mathsf{Defid} \\
\hline
\forall\, \mathsf{defid} : \mathsf{Defid};\ \mathsf{sd} : \mathrm{seq}\ \mathsf{Defid} \bullet \\
\quad \exists_1\, \mathsf{defs_after} : \mathrm{seq}\ \mathsf{Defid} \bullet \\
\qquad \mathsf{defs_before}\ (\mathsf{defid}, \mathsf{sd}) \frown \mathsf{defs_after} = \mathsf{sd} \land \\
\qquad \mathrm{last}\ (\mathsf{defs_before}\ (\mathsf{defid}, \mathsf{sd})) = \mathsf{defid}
\end{array}$$

The function **typeval** strips off the $\mathbb{P}$ from a **powerT**[5] and it is used to extract the type of an identifier introduced by a declaration. For instance, if an identifier i is declared as i: $\mathbb{P}\,\mathbb{Z}$, the expression $\mathbb{P}\,\mathbb{Z}$ has type $\mathbb{P}\,(\mathbb{P}\,\mathbb{Z})$ and the type of i is equivalent to $\mathbb{P}^{\sim}\,(\mathbb{P}\,(\mathbb{P}\,\mathbb{Z}))$.

$$\mathsf{typeval} == \mathsf{powerT}^{\sim} \circ \mathsf{typeof}$$

Each declaration generates a signature based on the type of the corresponding expression. The type of an expression depends on the types introduced in the current definition and on the types introduced in its previous definitions [See the specification of **typeval** and **typeof**].

$$\begin{array}{l}
\mathsf{mkdecl_sig} : (\mathrm{seq}\ \mathsf{Def} \times \mathsf{Decl}) \rightarrow \mathsf{Sig} \\
\hline
\forall\, \mathsf{sd} : \mathrm{seq}\ \mathsf{Def};\ \mathsf{dec} : \mathsf{Decl} \bullet \\
\quad \mathsf{mkdecl_sig}\ (\mathsf{sd}, \mathsf{dec}) = \{\, \mathsf{i} : \mathrm{ran}\ \mathsf{dec.id_list} \bullet \mathsf{i} \mapsto \mathsf{typeval}\ (\mathsf{sd}, \mathsf{dec.exp}) \,\}
\end{array}$$

Schemas and axiomatic definitions (not specified here) have a basic structure in common, which we call **Basic_Schema**. A basic schema has a set of identifiers for generic arguments (**gens**), and it introduces one or more declarations of variables (**dec_list**) as well as a predicate (**pred**). Non-generic schemas are special cases where **gens** = $\langle\rangle$. An extra attribute (**ids_in_sch**) is a derived variable representing all the visible identifiers introduced by the signature of a basic schema.

$$\begin{array}{l}
\textbf{Basic_Schema} \\
\hline
\mathsf{gens} : \mathrm{seq}\ \mathsf{Id} \\
\mathsf{dec_list} : \mathbb{P}_1\ \mathsf{Decl} \\
\mathsf{pred} : \mathsf{Pred} \\
\mathsf{ids_in_sch} : \mathbb{P}_1\ \mathsf{Id} \\
\hline
\mathsf{ids_in_sch} = \bigcup\, \{\, \mathsf{dec} : \mathsf{dec_list} \bullet \mathrm{ran}\ \mathsf{dec.id_list} \,\} \\
\mathsf{ids_in_sch} \cap \mathrm{ran}\ \mathsf{gens} = \varnothing
\end{array}$$

The function which creates the signature of a basic schema is based on its list of variable declarations.

---

5 **powerT** is a type constructor from a **Type** to $\mathbb{P}$ **Type**. Therefore, it is a total injection with a functional inverse.

```
mkbasic_sch_sig : (seq Def × ℙ₁ Decl) → Sig
─────────────────────────────────────
∀ sd : seq Def; declist : ℙ₁ Decl •
    mkbasic_sch_sig (sd, declist) = ⋃ { dec : declist • mkdecl_sig (sd, dec) }
```

A schema definition is a basic schema which also introduces an identifier for the schema (sch_id). The schema identifier is different from any of the identifiers introduced in the declaration part and the ones used as type parameters. An extra attribute (visible_ids_in_pred) records the visible identifiers of the schema which are used in its predicate part. In Section 3.3 we will explain the use of this attribute.

```
┌─ Schema_Def ─────────────
│ sch_id : Id
│ Basic_Schema
│ visible_ids_in_pred : ℙ Id
├──────────────────────────
│ sch_id ∉ ids_in_sch ∪ ran gens
│ visible_ids_in_pred ⊆ ids_in_sch
└──────────────────────────
```

The structure of Def may be enriched as:

```
Def ::= gset «GSet_Def» | schema «Schema_Def» | ...
```

We now extend Env_Spec with the relations uses_in_pred and depends_on_pred in order to record the dependencies of definitions on the predicate parts of schema definitions. The relation uses_in_pred records, for each definition, all the visible identifiers of other definitions which the definition uses in its predicate part. The relation depends_on_pred is defined similarly to the relation depends_on, specified in sub-section 3.1.4. A definition d_1 depends on the predicate part of a schema definition d_2, if d_1 uses d_2 in its predicate part, and d_2 comes before d_1 in the sequence of definitions. The use of the relations depends_on_pred in conjunction with uses_in_pred will be explained in sub-section 3.3. The visible identifiers of a schema definition correspond to the schema's identifier and all the other visible identifiers introduced by the corresponding basic schema.

```
┌─ Env_Spec ──────────────────────────────────────
│ Env_Spec
│ _ uses_in_pred _ : Defid ↔ (Defid × Id)
│ _ depends_on_pred _ : Defid ↔ Defid
├─────────────────────────────────────────────────
│ (_ uses_in_pred _) ⊆ (_ uses _)
│ (_ depends_on_pred _) ⊆ (_ depends_on _)
│ ∀ d1, d2 : ran spec | store d2 ∈ ran schema •
│     d1 depends_on_pred d2 ⇔
│         (∃ i : Id; s : Schema_Def | schema s = store d2 ∧ i = s.sch_id •
│             d1 uses_in_pred (d2, i) ∧ d2 before d1)
│ ∀ defid : ran spec; s : Schema_Def | schema s = store defid •
│     visible_ids⦇{defid}⦈ = {s.sch_id} ∪ s.ids_in_sch
└─────────────────────────────────────────────────
```

A schema s_new is a *subtype* of its old version s_old, if s_new has the same identifier as s_old and their generic schema types are the same. The variables new_pre and old_pre correspond to the sequence of definitions up to s_new and s_old respectively. These sequences are necessary in order to calculate the signatures of those schemas.

```
_ sub_schema _ : (Schema_Def × seq Def) ↔ (Schema_Def × seq Def)
─────────────────────────────
∀ s_new, s_old : Schema_Def; new_pre, old_pre : seq Def •
    (s_new, new_pre) sub_schema (s_old, old_pre) ⇔
        s_old.sch_id = s_new.sch_id ∧
        (∃ new_sig, old_sig : Sig; new_genT, old_genT : Type; subst : Subst |
         new_sig = mkbasic_sch_sig (new_pre, s_new.dec_list) ∧
         old_sig = mkbasic_sch_sig (old_pre, s_old.dec_list) ∧
         new_genT = genT (s_new.gens, schemaT new_sig) ∧
         old_genT = genT (s_old.gens, schemaT old_sig) •
             old_genT = type_subst subst new_genT)
```

The *subtype* relationship for schemas does not allow a replacement definition to extend its previous definition with other identifiers, because the introduction of new identifiers in the signature of a replacement schema may or may not introduce type errors. On the one hand, type-checking errors would be introduced in definitions which referred to the whole previous definition of a schema, because the signature of the replacement schema would be different from the signature of its previous definition (e.g. type errors would be introduced in definitions which use a schema as a type). On the other hand, type-checking errors would not be introduced in definitions which referred to qualified identifiers which have not changed their types. Hence, to guarantee the safety of the type system, the approach of requiring equality of both generic schema types was adopted.

Another problem of defining *subtype* based on the signatures of the schemas is that even if the new version of a schema has the same signature as its old version, it is possible that type-errors are introduced in the specification. For instance, the schema

```
Sch [X, Y]
─────────
a : ℙ (X × Y)
b : ℙ (Y × X)
```

with generic type genT (⟨ X, Y ⟩, schemaT (a ↦ powerT (X, Y), b ↦ powerT (Y, X))) and the schema

```
Sch [Y, X]
─────────
a : ℙ (X × Y)
b : ℙ (Y × X)
```

with generic type genT (⟨ Y, X ⟩, schemaT (a ↦ ℙ (X, Y), b ↦ ℙ (Y, X))) are not *subtypes* of each other, because the former cannot replace the latter without introducing instantiation errors in definitions which instantiate the first version of Sch since the order of the generic parameters has been changed. Instantiation errors would also occur if the number of generic parameters was not the same in both schemas. Strictly speaking the equivalence of the generic types of both schemas is sufficient, not necessary. However, any other approach would require very complex analysis of uses of definitions.

Finally, the general *subtype* relationship combines the specification of *subtype* for each kind of definition.

```
_ subtype _ : (Def × seq Def) ↔ (Def × seq Def)
─────────────────────────────
∀ g_new, g_old : GSet_Def; s_new, s_old : Schema_Def; new_pre, old_pre : seq Def •
    ((gset g_new, new_pre) subtype (gset g_old, old_pre) ⇔
        g_new sub_gset g_old) ∧
    ((schema s_new, new_pre) subtype (schema s_old, old_pre) ⇔
        (s_new, new_pre) sub_schema (s_old, old_pre)) ∧ ...
```

3.3 Theoretical Issues of Editing Operations

In this sub-section, we specify the issues that a type-checking algorithm must deal with when checking Z specifications incrementally. First we give some auxiliary definitions. Given a relation recording dependencies between definitions (i.e. a "subset" of the dependency graph of a specification) and a set of identifiers of definitions, the function dependents gives the identifiers of all the definitions which depend (directly or indirectly) on the definitions corresponding to the given identifiers of definitions. The definitions' identifiers returned by dependents are the ones which will be re-checked as a consequence of an editing operation.

```
dependents : (Defid ↔ Defid × ℙ Defid) → ℙ Defid
───────────────
∀ depends_on : Defid ↔ Defid; defids : ℙ Defid •
    dependents (depends_on, defids) = depends_on⁺∼ ⦇defids⦈
```

Given a set of identifiers of definitions to be checked (check?), the operation schema Check specifies an order (check!) for checking the corresponding definitions after an editing operation is executed.

```
┌─ Check ───────────
│ ΞEnv_Spec
│ check? : ℙ Defid
│ check! : seq Defid
├───────────
│ check! = spec ↾ check?
└───────────────────
```

For the sake of simplicity, when specifying the insertion and the modification of a definition in a specification, we assume that the edited definition was already checked before the system found out which dependent definitions must also be re-typechecked. We also assume that during the check of the edited definition neither syntactic errors nor type errors are found. In practice, definitions with errors should be flagged for later re-check, because the errors may be corrected as a consequence of editing other definitions. It is also assumed that dependent definitions directly, and indirectly affected by an editing operation are re-typechecked immediately after the editing operation is executed.

3.3.1 Insertion

- When a definition Def_2 is inserted into a specification after another definition Def_1, and Def_2 becomes a multiple definition (or a schema extension) of Def_1, or any visible identifier of Def_2 becomes a multiple declaration of an identifier introduced in Def_1, the definitions which depend on Def_1 and are subsequent to Def_2 must be re-typechecked in order to rebind the references to the identifiers (or definitions) which have become multiply declared (or defined/extended);
- When a definition Def_2 is inserted before another definition Def_1, and Def_1 becomes a multiple definition (or a schema extension) of Def_2, or any visible identifier of Def_1 becomes a multiple declaration of an identifier introduced in Def_2, Def_1 and the definitions which depend on Def_1 must be re-typechecked.

For instance, if we have the sequence of axiomatic definitions

```
│ a : ℤ      │ b : ℤ      │ c : ℤ      │ d : ℤ      │ e : ℤ
│            ├────        ├────        │            ├────
│            │ b = a      │ c = a      │            │ e = d
```

which can be represented as ⟨.....⟩ (where each capital letter from A to E corresponds to one of the above definitions in the same order of introduction), and we insert the definition

$$
\begin{array}{|l}
a, d : \mathbb{Z}
\end{array}
$$

after the definition A and before the definition B, the definitions B and C need to be re-checked because a became multiply declared in the inserted definition, and the definitions D and E need to be re-checked because d became multiply declared in D. We can specify the above issues in the following steps:

The schema Pure_AddDef specifies the insertion of a definition def? identified by defid? at the position pos? of a specification. The variable used? corresponds to the identifiers used by the definition. The variable used_in_pred? corresponds to the identifiers used by the definition in its predicate part (definitions without predicate part have used_in_pred? = $\varnothing$). The values of used? and used_in_pred? can be discovered by the type-checker when the definition is checked. The update of visible_ids is guaranteed by the invariant in Env_Spec that says what are the visible identifiers of each kind of Z definition and by the invariant ran (_ uses _) $\subseteq$ visible_ids. The update of depends_on is guaranteed by the invariant which relates uses to depends_on in Env_Spec. The update of depends_on_pred is guaranteed by the invariant in Env_Spec which relates uses_in_pred to depends_on_pred.

$$
\begin{array}{|l}
\textbf{Pure\_AddDef} \\
\hline
\Delta \text{Env\_Spec} \\
\text{defid?} : \text{Defid} \\
\text{def?} : \text{Def} \\
\text{pos?} : \mathbb{N}_1 \\
\text{used?} : \mathbb{P}\,(\text{Defid} \times \text{Id}) \\
\text{used\_in\_pred?} : \mathbb{P}\,(\text{Defid} \times \text{Id}) \\
\hline
\text{defid?} \notin \text{ran spec} \\
\text{pos?} \leq \#\text{spec} + 1 \\
\text{def?} \in \text{ran gset} \Rightarrow \text{used\_in\_pred?} = \varnothing \\
\text{store}' = \text{store} \cup \{\text{defid?} \mapsto \text{def?}\} \\
\text{spec}' = (1 \mathinner{..} \text{pos?} - 1) \lhd \text{spec} \frown \langle \text{defid?} \rangle \frown (\text{pos?} \mathinner{..} \#\text{spec}) \lhd \text{spec} \\
(\_\ \text{uses}'\ \_) = (\_\ \text{uses}\ \_) \cup \{\text{use} : \text{used?} \bullet (\text{defid?}, \text{use})\} \\
(\_\ \text{uses\_in\_pred}'\ \_) = \\
(\_\ \text{uses\_in\_pred}\ \_) \cup \{\text{usepred} : \text{used\_in\_pred?} \bullet (\text{defid?}, \text{usepred})\} \\
\hline
\end{array}
$$

Identifiers multiply defined/declared or schema extensions correspond to multiple occurrences of identifiers in scope. Given an identifier of a definition (defid), a set of definitions' identifiers (defids) and a relation recording the visible identifiers of all the definitions in a specification (visible_ids), the function defs_with_mult_ids returns a subset of defids corresponding to the identifiers of definitions (if any) which introduce in scope multiple occurrences of any of the visible identifiers (def_visible_ids) of defid.

$$
\begin{array}{|l}
\text{defs\_with\_mult\_ids} : (\text{Defid} \times \mathbb{P}\,\text{Defid} \times \text{Defid} \leftrightarrow \text{Id}) \rightarrow \mathbb{P}\,\text{Defid} \\
\hline
\forall\, \text{defid} : \text{Defid}; \text{defids} : \mathbb{P}\,\text{Defid}; \text{visible\_ids} : \text{Defid} \leftrightarrow \text{Id} \bullet \\
\quad \exists\, \text{def\_visible\_ids} : \mathbb{P}\,\text{Id} \mid \text{def\_visible\_ids} = \text{visible\_ids}(\!|\{\text{defid}\}|\!) \bullet \\
\qquad \text{defs\_with\_mult\_ids}\ (\text{defid}, \text{defids}, \text{visible\_ids}) = \\
\qquad (\text{visible\_ids}^{\sim}(\!|\text{def\_visible\_ids}|\!)) \cap \text{defids}
\end{array}
$$

The auxiliary schema Test_if_mult_ids returns in check_mult the identifiers of definitions (if any) which need to be re-checked due to the insertion of multiple occurrences of any visible identifier in scope. This schema does not modify the state of the system. However, the state before the execution of an editing operation (in particular, insertions and modifications as we will see later), as well as the modified state are used in Test_if_mult_ids.

```
┌─ Test_if_mult_ids ─────────────────────────────────────
│ ΔEnv_Spec
│ defid? : Defid
│ pos? : ℕ₁
│ check_mult : ℙ Defid
├──────────────────
│ spec' pos? = defid?
│ ∃ check_bf, check_af : ℙ Defid •
│     (defs_mult_bf = {} ⇒ check_bf = {}) ∧
│     (defs_mult_bf ≠ {} ⇒
│          check_bf =
│          dependents ((_ depends_on _), defs_mult_bf) ∩
│          ran ((pos? + 1 .. #spec') ◁ spec')) ∧
│     ((defs_mult_af = {} ⇒ check_af = {}) ∧
│      (defs_mult_af ≠ {} ⇒
│           check_af =
│           dependents ((_ depends_on _), defs_mult_af) ∪ defs_mult_af)) ∧
│     check_mult = check_bf ∪ check_af
│ where
│  ┌──────────
│  │ defs_mult_bf, defs_mult_af : ℙ Defid
│  ├──────────────────
│  │ defs_mult_bf = defs_with_mult_ids (defid?, ran ((1 .. pos? − 1) ◁ spec'), visible_ids')
│  │ defs_mult_af = defs_with_mult_ids (defid?, ran ((pos? + 1 .. #spec') ◁ spec'), visible_ids')
│  └──────────
└────────────────────────────────────────────────────────
```

This schema says that if there are definitions (defs_mult_bf) preceding the edited definition (defid?) which also introduce in scope any of the visible identifiers of defid? (i.e. defid? introduces multiple identifiers in scope), all the definitions which depended on the definitions in defs_mult_bf (before the editing of defid?) and became subsequent to defid? (after the editing of defid?) must be re-checked. If there are definitions subsequent (defs_mult_af) to defid? which also introduce any of its visible identifiers, the definitions in defs_mult_af and their dependent definitions must be re-checked.

In the schema Add_Def we specify the effect of inserting a definition which may introduce multiple identifiers or a schema extension in scope. In this schema and in subsequent schemas, the variable check records the identifiers of the definitions which need to be checked as a consequence of an editing operation.

Add_Def ≙ Pure_AddDef ∧ $\text{Test\_if\_mult\_ids}_{[check/check\_mult]}$

Finally, the schema AddCheck specifies the insertion operation in full.

AddCheck ≙ Add_Def ≫ Check

3.3.2 Deletion

- When a definition is removed from the scope of a specification, all the definitions which depend directly or indirectly on the removed definition must be re-typechecked. We specify this in two steps:

The schema Pure_DelDef corresponds to the deletion of a definition from the specification. A deleted definition is removed from the specification, and all the records of its use are also removed. The update of depends_on is guaranteed by the invariant which relates uses to depends_on, and the update of depends_on_pred is guaranteed by the invariant which relates uses_in_pred to depends_on_pred.

```
_ Pure_DelDef ____________________________________
  ΔEnv_Spec
  defid? : Defid
  dpos? : ℕ₁
____________
  defid? = spec dpos?
  store' = {defid?} ⩤ store
  spec' = (1 .. dpos? - 1) ◁ spec ⁀ (dpos? + 1 .. #spec) ◁ spec
  visible_ids' = {defid?} ⩤ visible_ids
  (_ uses' _) = {defid?} ⩤ (_ uses _)
  (_ uses_in_pred' _) = {defid?} ⩤ (_ uses_in_pred _)
__________________________________________________
```

Finally, we can specify the consequential effect of deleting a definition as follows.

```
_ Del_Def ________________________________________
  Pure_DelDef
  check! : ℙ Defid
____________
  check! = dependents ((_ depends_on _), {defid?})
__________________________________________________
```

Any definitions which depend on the deleted definition must be checked.

```
DelCheck ≙ Del_Def ≫ Check \ (defid?)
```

3.3.3 Modification

- When a definition is replaced by another definition *Def* which is a *subtype* of its previous version, the definitions which depended on the previous version of *Def* do not need to be re-typechecked, because type errors are not introduced in the specification. However, if *Def* is a schema definition, it is necessary to re-typecheck any definition that used the previous version of *Def* as a predicate, if the set of visible identifiers of *Def* that are used in the predicate part of *Def* are not the same as the visible identifiers used in the predicate part of its previous version. For instance, if S_1 is a schema used as a predicate in another schema S_2, then all the variables declared in the declaration part of S_1 and referred to in the predicate part of S_1 must also be in scope when S_1 is used as a predicate.

```
_ S₁ ______      _ S₂ ______
  x, y, z : ℤ      x, y : ℤ
_____            _____
  x > y            S₁
___________      ___________
```

If we modify the predicate part of S_1 to use z in its predicate part and re-check S_2, a scope error will be introduced in S_2 because z is not declared in S_2. Notice that this rule is not valid for the Z standard Version 1.0 [10] which insists that all variables declared in the schema S_1 have been declared in the current environment, even if some of them are not referenced by the current predicate;

- When a definition Def_1 is replaced by another definition Def_2 which is not a *subtype* of Def_1, all the definitions which depend on Def_1 must be re-typechecked;
- In both cases, it is possible that Def_2 becomes a multiple definition (or schema extension), or introduces multiple declarations in scope. In this case, it is necessary to re-typecheck all the definitions which are subsequent to Def_2 and depend on any definition that became multiply defined (or extended), or depend on any definition that introduced the identifiers which became multiply declared.

These issues can be specified in the following steps: The schema Pure_ChangeDef specifies the modification of a definition in the specification. This schema can be specified as a deletion of an old definition followed by an insertion of a new definition in its place.

$$\text{Pure\_ChangeDef} \mathrel{\widehat{=}} \text{Pure\_DelDef} \mathbin{⨟} \text{Pure\_AddDef}_{[dpos?/pos?]}$$

The schema Change_test_mult_ids specifies the modification of a definition followed by a test to discover if multiple identifiers were introduced in scope. As described above, this test is performed whether or not the modified definition is a *subtype* of its old version.

$$\text{Change\_test\_mult\_ids} \mathrel{\widehat{=}} \text{Pure\_ChangeDef} \wedge \text{Test\_if\_mult\_ids}_{[dpos?/pos?]}$$

The schema Change_is_not_SubType specifies the case when the new version of a modified definition is not a *subtype* of its old version. The definitions which depend on the old version of the modified definition and definitions affected by the insertion of multiple identifiers in scope need to be re-checked.

```
┌─ Change_is_not_SubType ──────────────────────────────
│ Change_test_mult_ids
│ check! : ℙ Defid
├──────────────────────
│ ¬ (new_def, new_pre) subtype (old_def, old_pre) ∧
│ check! = dependents ((_ depends_on _), {defid?}) ∪ check_mult
│ where
│   │ new_def, old_def : Def
│   │ new_pre, old_pre : seq Def
│   ├──────────────
│   │ new_def = store' defid?
│   │ new_pre = store' ∘ defs_before (defid?, spec')
│   │ old_def = store defid?
│   │ old_pre = store ∘ defs_before (defid?, spec)
└──────────────────────────────────────────────────────
```

Similarly, the schema Change_is_SubType specifies the type-checking issues when the new version of a modified definition is a *subtype* of its old version. As explained above, if the modified definition is a schema definition (s_new) and the visible identifiers used in the predicate part of the modified definition (s_new.visible_ids_in_pred) are not the same as the visible identifiers used in the predicate part of its old version (s_old.visible_ids_in_pred), the definitions which depend on the predicate part (dep_on_pred) of its old version (s_old) must be re-checked. It is also possible that some definitions (check_mult) need to be re-checked due to the insertion of multiple identifiers in scope.

```
─ Change_is_SubType ──────────────────────────
│ Change_test_mult_ids
│ check! : ℙ Defid
├──────────────────
│ (new_def, new_pre) subtype (old_def, old_pre) ∧
│ (∃dep_on_pred : ℙ Defid •
│     (new_def ∈ ran schema ∧ old_def ∈ ran schema ⇒
│         (∃s_new, s_old : Schema_Def |
│             schema s_new = new_def ∧ schema s_old = old_def •
│                 (s_new.visible_ids_in_pred ≠ s_old.visible_ids_in_pred ⇒
│                         dep_on_pred =
│                         dependents ((_ depends_on_pred _), {defid?})) ∧
│                 (s_new.visible_ids_in_pred = s_old.visible_ids_in_pred ⇒
│                         dep_on_pred = {})))) ∧
│     (new_def ∉ ran schema ∧ old_def ∉ ran schema ⇒ dep_on_pred = {}) ∧
│     check! = dep_on_pred ∪ check_mult)
│ where
│   │ new_def, old_def : Def
│   │ new_pre, old_pre : seq Def
│   ├──────────────
│   │ new_def = store' defid?
│   │ new_pre = store' ∘ defs_before (defid?, spec')
│   │ old_def = store defid?
│   │ old_pre = store ∘ defs_before (defid?, spec)
│   │
└──────────────────────────────────────────────
```

Finally, we can specify the effect of modifying a definition in the specification.

```
ChangeCheck ≙
    (Change_is_SubType ∨ Change_is_not_SubType) ≫ Check \ (defid?, check_mult)
```

3.3.4 Transference

• The transference of a definition from one position to another can be achieved by treating it as a deletion from the old position followed by an insertion into the new position. This is specified in the following two schemas.

$\text{Transfer\_Def} \mathrel{\hat=} \text{Del\_Def}_{[\text{delcheck/check!}]} \mathbin{⨾} \text{Add\_Def}_{[\text{addcheck/check!}]}$

```
TransferCheck ≙
    [Transfer_Def; check! : ℙ Defid | check! = delcheck ∪ addcheck] ≫ Check \
    (defid?, delcheck, addcheck)
```

4. CONCLUSION

The application of incremental type-checking mechanisms to specification languages, and particularly to the Z language, is almost an untouched research area. Some work has been done in the area of incremental checking (parsing and/or type-checking) applied to imperative and functional programming languages [11, 2, 12, 13, 14, 15]. However, the same techniques used when checking these languages cannot be directly applied to Z, due to differences in its type system and scope rules. Moreover, none of those existing incremental algorithms are appropriate for dealing with extensions of schemas. In those algorithms, an extension of a definition is either treated as a scope error or it overwrites the previous definition.

We believe that the formalisation of the possible dependencies between Z definitions, the extensive discussion of the incremental type-checking issues, and the description of our incremental type-checking algorithm represent a novel piece of research work towards clarifying theoretical problems related to the incremental processing of Z specifications.

5. REFERENCES

1. Spivey, J. M., *The Z Notation: A Reference Manual — 1st edition*, Prentice-Hall Intl. (1989).
2. Toyn, I., "Exploratory Environments for Functional Programming", D.Phil. Thesis, Department of Computer Science, University of York (Apr. 1987).
3. Spivey, J. M., *Understanding Z: A Specification Language and its Formal Semantics*, Cambridge University Press (Jan. 1988).
4. Atkins, M. C., "Implementation Techniques for Object-Oriented Systems", D.Phil. Thesis, Department of Computer Science, University of York (Jun. 1989).
5. Milner, R., "A Theory of Type Polymorphism in Programming", *Journal of Computer and System Sciences* 17(3), pp. 348-375 (Dec. 1978).
6. Sennett, C., "Review of Type Checking and Scope Rules of the Specification Language Z", Report No. 87017, Royal Signals and Radar Establishment, Malvern, UK (Nov. 1987).
7. Reed, J. N. and Sinclair, J. E., "An Algorithm for Type-Checking Z: A Z Specification", PRG-81, Programming Research Group, University of Oxford (Mar. 1990).
8. Hayes, I. (editor), *Specification Case Studies*, Prentice-Hall Intl. (1987).
9. de Vasconcelos, A. M. L., "Incremental Processing of Z Specifications", D.Phil. Thesis, Department of Computer Science, University of York (Oct. 1993).
10. Z Standards Review Committee, "Z Base Standard (Version 1.0)", BSI Panel IST/5-/52 (1993).
11. Nikhil, R. S., "Practical Polymorphism", *Lecture Notes in Computer Science*, Nancy, France 201, pp. 319-333, Springer-Verlag, Functional Programming Languages and Computer Architecture (Sep. 1985).
12. Toyn, I. and Runciman, C., "Performance Polymorphism", *Lecture Notes in Computer Science*, Proceedings of the Third International Conference on Functional Programming Languages and Computer Architecture (Sep. 1987).
13. R. Medina-Mora and P. H. Feiler, "An Incremental Programming Environment", *IEEE Transaction on Software Engineering* SE-7(5), pp. 471-481 (Sep. 1981).
14. Nicol, C., Crowe, M. K., Corr, M. E., Oram, J. W. and Jenkins, D. G., "IDEA — An Incremental Development Environment for ADA", *Software Engineering Journal* 2(6), pp. 194-198 (Nov. 1987).
15. Schwartz, M. D., Delisle, N. M. and Begwani, V. S., "Incremental Compilation in Magpie", *SIGPLAN Notices* 19(6), pp. 122-131, Proceedings of the SIGPLAN Symposium on Compiler Construction (Jun. 1984).