The DeCCo project papers VI

Z to Prolog DCTG translation guidelines

Susan Stepney

University of York Technical Report YCS-2003-363

June 2003

Contents

Preface

Historical background of the DeCCo project

In 1990 Logica's Formal Methods Team performed a study for RSRE (now QineticQ) into how to develop a compiler for high integrity applications that is itself of high integrity. In that study, the source language was Spark, a subset of Ada designed for safety critical applications, and the target was Viper, a high integrity processor. Logica's Formal Methods Team developed a mathematical technique for specifying a compiler and proving it correct, and developed a small proof of concept prototype. The study is described in [Stepney *et al* 1991], and the small case study is worked up in full, including all the proofs, in [Stepney 1993]. Experience of using the PVS tool to prove the small case study is reported in [Stringer-Calvert *et al*]. Futher developments to the method to allow separate compilation are described in [Stepney 1998].

Engineers at AWE read about the study and realised the technique could be used to implement a compiler for their own high integrity processor, called the ASP (Arming System Processor). They contacted Logica, and between 1992 and 2001 Logica used these techniques to deliver a high integrity compiler, integrated in a development and test environment, for progressively larger subsets of Pascal.

The full specifications of the final version of the DeCCo compiler are reproduced in these technical reports. These are written in the Z specification language. The variant of Z used is that supported by the *Z Specific Formaliser* tool [Formaliser], which was used to prepare and type-check all the DeCCo specifications. This variant is essentially the Z described in the *Z Reference Manual* [Spivey 1992] augmented with a few new constructs from *ISO Standard Z* [ISO-Z]. Additions to ZRM are noted as they occur in the text.

The DeCCo Reports

The DeCCo Project case study is detailed in the following technical reports (this preface is common to all the reports)

I. **Z Specification of Pasp**
The denotational semantics of the high level source language, Pasp. The definition is split into several static semantics (such as type checking) and a dynamic semantics (the meaningof executing a program). Later smeantics are not defined for those programs where the result of earlier semantics is *error*.

II. **Z Specification of Asp, AspAL and XAspAL**
The denotational semantics of the low level target assembly languages. XAspAL is the target of compilation of an individual Pasp module; it is AspAL extended with some cross-module instructions that are resolved at link time. The meaning of these extra instructions is given implicitly by the specification

of the linker and hexer. AspAL is the target of linking a set of XAspAL modules, and also the target of compilation of a complete Pasp program. Asp is the non-relocatable assembly language of the chip, with AspAL's labels replaced by absolute program addresses. The semantics of programs with errors is not defined, because these defintions will only ever be used to define the meaning of correct, compiled programs.

III. **Z Specification of Compiler Templates**
The operational semantics of the Pasp source language, in the form of a set of XAspAL target language templates.

IV. **Z Specification of Linker and Hexer**
The linker combines compiled XAspAL modules into a single compiled AspAL program. The hexer converts a relocatable AspAL program into an Asp program located at a fixed place in memory.

V. **Compiler Correctness Proofs**
The compiler's operational semantics are demonstrated to be equivalent to the source language's denotational semantics, by calculating the meaning of each Pasp construct, and the corresponding meaning of the AspAL template, and showing them to be equivalent. Thus the compiler transformation is *meaning preserving*, and hence the compiler is correct.

VI. **Z to Prolog DCTG translation guidelines**
The Z specifications of the Pasp semantics and compiler templates are translated into an executable Prolog DCTG implementation of a Pasp interpreter and Pasp-to-Asp compiler. The translation is done manually, following the stated guidelines.

Acknowledgements

We would like to thank the client team at AWE – Dave Thomas, Wilson Ifill, Alun Lewis, Tracy Bourne – for providing such an interesting development project to work on. We would like to thank the rest of the development team at Logica: Tim Wentford, John Taylor, Roger Eatwell, Kwasi Ametewee.

1. Introduction

This document describes the DeCCo process for mapping a Z specification of a denotational semantics into its demonstrably equivalent Prolog implementation.

It assumes familiarity with Z [Spivey 1992][Formaliser], Prolog [Clocksin 1984], and the way these are used to specify and implement denotational semantics [Stepney].

In this document, Z names are written in `typewriter font`, and Prolog names in **`bold typewriter font`**. Translations of small pieces of Z to corresponding Prolog are shown thus:

```
Z term  -->  Prolog translation
```

Caveat: the DeCCo specifications have been written in a "constructive" style, of the form `x = f(...)`. This makes them more suitable for a direct translation to Prolog. Most of the translation techniques described here rely on this style, and would not necessarily work for a general Z-to-Prolog translation system, with more general predicates describing the form of the functions.

2. Plain Z

This section describes the conventions we use for translating plain Z (rather than the meaning functions) into plain Prolog (not using DCTGs). This is the style used for implementing the Z toolkit functions [Spivey 1992, chapter 4], and also some of the intermediate expressions in the DeCCo semantic definitions.

2.1 Z and Prolog naming conventions

The DeCCo Z specifications are written using the following variable naming convention:

- Given sets, all upper case. For example, `VALUE`, `TYPE`, `STMT`
- Schema names, initial upper case, then mixed. For example, `IfStmt`, `Register`
- Other names, initial lower case, then mixed. For example, `ifStmt`, `bmul`. This rule is broken occasionally in variable names with very localised scope, to indicate the variable is a set or sequence. For example, `a : ATTR; A : ℙ ATTR`.
- Infix toolkit-like operators often have symbolic names. For example, ◁, ⊆
- Variables used in similar contexts have the same name, but with decorations – here usually dashes[1]. For example, `ρτt`, `ρτt′`

Prolog has a different naming requirement

- Constants must start with a lower case letter. For example, `ifStmt`, `bmul`
- Variables must start with an upper case letter, or an underscore. For example, `E1`, `IfStmt`, `_2b`

We have a further convention, for translating Z names to Prolog names. (This convention is occasionally ignored, to disambiguate Z names that end up with the same Prolog translation, or for readability.)

In the simplest case, the Prolog name is just the Z name, with the first character changed to be of the required capitalisation.

A symbolic Z name is given some appropriate Prolog name, usually its spoken form. For example, the Z union operator ∪ becomes the Prolog `union`. Greek letters are also symbolic characters, heavily used as variable names in DeCCo: they can be translated to their spoken equivalent (for example, the variable ξ might become `Xi`, `ρτ` might become `RhoTau`) or to their first character (δ might become `D`).

[1] Note that the dashes on names simply serve to provide distinct variables with related names. We are not using dashes to indicate "after state" variables, because we are not using a "Delta/Xi" state-and-operations style specification.

A dashed Z name has a digit in the Prolog form. For example, the Z variable `b′` becomes the Prolog variable `B1` (uppercased to become a variable, digit `1` added for the dash). Similarly, the Z variable `b′′′` becomes the Prolog variable `B3`.

There are a few occasions when DeCCo uses two Z names that differ only in their capitalisation:

- A lower case Z name indicates a single element, and the same name in upper case indicates a set or sequence of elements. For example, `a : ATTR; A : ℙ ATTR`. The conventional Prolog naming for lists is to add an `s` to the base name, so these two become the Prolog variables `A` and `As`, respectively.
- A lower case Z name indicates a free type branch, whose type is then a schema of a similar name. For example `ifStmt ⟪ IfStmt ⟫`. Fortunately in this case the branch name maps to a Prolog constant, and the schema name to a Prolog variable. So the translation causes no ambiguity.

2.2 Z types

Z is based on typed set theory: the world of values is partitioned into types, and every value has a particular type. Prolog is not explicitly typed, and its main data structure is the list. It is fairly straightforward to translate a Z set to a Prolog list – the main thing that requires care is ensuring that either the Prolog list has no duplicates, or that the operations on it make it *behave* as if it has no duplicates.

Every Z variable must be declared, and given a type. Z types provide redundancy, and also stop some potentially paradoxical statements (for example, Russell's paradox) from being formulated in Z. If a Z specification passes type-checking, the type of each of its variables is uniquely determined.

Prolog variables need not be declared. They may have any "type", fixed only once they have unified. So it is not necessary to translate the Z variable declarations and types into Prolog. However, a Z declaration may constrain a variable with an implicit predicate, as well as giving it a type:

- `x : ℤ` declares a variable `x` to have type `ℤ` (integer). In Prolog, the corresponding variable `X` may be used with no corresponding declaration.
- `x : ℕ` declares a variable `x` to have type `ℤ` (integer) and also be constrained to be in the set `ℕ` (non-negative). It is a shorthand for the equivalent normalised declaration `x : ℤ | x ∈ ℕ`. In Prolog, the corresponding variable `X` may again be used with no corresponding declaration, but there needs to be a translation of the predicate, `X >= 0`.

2.3 Z values

Z has given sets, which introduce a type whose values have no internal structure, and three type constructors: power set, Cartesian product, and schema type. It is necessary to decide how to model values whose types have structure, because the values have corresponding structure that can be manipulated (for example, a tuple, a value of type Cartesian product, can have its components accessed individually).

2.3.1 Given sets

There is one special given set in Z: the set of integers `ℤ`. Z integers are translated directly to Prolog integers.[2]

Given sets occur in two forms: simple given sets, and free type definitions.

2.3.1.1 Simple given sets

A given set declaration introduces a new type and a corresponding new set of elements to the specification, where these elements have no *internal* structure of their own. Certain elements from a given set may be named as global constants within the Z specification. These global names can be transliterated to Prolog objects.

The main given set in DeCCo is `ID`, modelling Pasp variable identifiers, with one distinguished name `maxunsigned`. This name is transliterated directly into the corresponding Prolog object, `maxunsigned`.

2.3.1.2 Free types

A free type is a given set with some *external* structure imposed: the elements in the given set are partitioned, each partition corresponding to a branch of the free type definition.

Simple branches, comprising just a Z name, contain a single element; constructor branches, comprising the name of an injective function and an argument, contain a set of elements.

The simple branch names are the equivalent of global constants, and so are transliterated into Prolog. DeCCo has various free types that have simple branches comprising a Z name, for example `skip` or `beq`. These are just transliterated directly into corresponding Prolog objects, in this case, `maxunsigned`, `skip` and `beq`.

The constructor branch injection names serve to convert one Z type into (part of) another (larger) type. We can tell, given an element of the larger type, which injection was used to construct it, that is, which one of the smaller types it came from. We model this in Prolog by modelling the branch constructors and their arguments as Prolog structures. So, if we have the following Z free type definition of `F`, along with a predicate that an element is constructed from one particular branch of `F`

```
F ::= a | b ⟪ N ⟫ | c ⟪ F × F ⟫
z = c(f1,f2)
```

we convert the predicate to Prolog thus:

---

[2] There are rather more Z integers than Prolog ones. However we never use them all in DeCCo. We use at most `- maxunsigned .. maxunsigned * maxunsigned` of them. Provided these values never crash the Prolog system, we can implement Z integers naively.

```
Z = c( F1 , F2 )
```

If we have a predicate that an element is in one particular branch of `F`

```
x = a
y ∈ ran b
z ∈ ran c
```

we convert the predicate to Prolog by asserting that the corresponding Prolog variable unifies with the relevant structure:

```
X = a
Y = b( _ )
Z = c( _ , _ )
```

2.3.2 Cartesian tuples

A Z Cartesian tuple is an ordered collection of elements. A Z tuple is translated into a Prolog list

```
(x, y, z)  -->  [X, Y, Z]
```

ISO Standard Z (but not ZRM) provides a notation for accessing tuple elements by their position in the collection. For example, `t.3`, or `(x,y,z).2 = y`

Tuple selection is translated as selecting the *n*th element out of the list. We define a general purpose operator for selecting the *n*th component of a list (Z typechecking ensures this is never applied to a list that does not contain at least this number of elements):

```
dot([X| Xs], 1, X).
dot([_| Xs], N, Y)  :-  N1 is N - 1, dot(Xs, N1, Y).
```

We use it to select say the third component thus

```
dot(T, 3, Z).
```

or we can just directly access the third component of `T` by writing `T` as an explicit list (here assuming `t` is a 6-tuple).

```
[ _, _, Z, _, _, _ ]
```

2.3.3 Schema bindings

A Z schema binding is a labelled collection of elements, where the labels are schema component names. ISO Standard Z (but not ZRM) provides a notation for the expressing the value of a particular binding, like

```
⟨| a == x, b == y, c == z |⟩
```

Each component of a Z binding is translated into a two element Prolog list, `[name, Value]`, and the whole binding to a list of these pairs:

```
⟨| a == x, b == y, c == z |⟩   -->  [ [a,X], [b,Y], [c,Z] ]
```

Binding elements can be accessed by their name. For example, `S.z`, or `⟨| a == x, b == y, c == z |⟩.b = y`

Component selection is translated as selecting the value corresponding to the pair with the relevant name. We define a general purpose operator for selecting a named component of a list (Z type-checking ensures this is never applied to a list that does not contain that name): [3]

```
dot([[Name, Y]| Xs], Name, Y).
dot([_| Xs], Name, Y) :- dot(Xs, Name, Y).
```

then use it to select say the `xi` component thus

```
dot(S, xi, X).
```

Or we can just directly access the `xi` component of `S` by writing `S` as an explicit list[4] (here assuming `S` is a 4-component binding, and that the `xi` component is the second component in the list – a potentially fragile assumption).

```
[ _, [xi,X], _, _ ]
```

2.3.4 Flattened schema bindings

Earlier versions of DeCCo did not specify the arguments to free types using schemas, but rather using cartesian products. As the number of components in each argument grows, the schema form is preferred, as it provides more mnemonic tags when accessing components. More recent versions of the DeCCo Z specifications use schemas.

However, in some cases the implementation has not yet been changed to use this more readable schema form of the specification, and still uses the old cartesian product form. Such an implementation corresponds to a translation from Z schema bindings to Prolog that drops the component names:

```
⟨| a == x, b == y, c == z |⟩  -->  [ X, Y, Z ]
```

---

[3] This definition makes it clear that tuples and schema bindings could be unified by explicitly translating tuples as pairs labelled with numbers, as

```
(x, y, z)  -->  [[1, X], [2, Y], [3, Z]]
```

Such unification might be have advantages in a general purpose translation system, but here it would merely be slower and require more storage.

[4] Indeed, if we are *sure* that the x component is always second, we could write

```
[ _, [_,X], _, _ ]
```

2.3.5 Other values

All other values in Z are constructed from these: given values, sets, tuples and bindings. We can have sets of tuples, for example (if the tuples are pairs, such a set is called a *relation*). So in general we can recursively construct in Prolog any Z value by translating its recursive construction in Z. For example, because a Z set is translated to a Prolog list, and a Z pair to a 2-component Prolog list, then a Z relation (set of pairs) is translated to a list of 2-component lists:

```
{ (a, x), (b, y), (c, z) }  -->  [ [A,X], [B,Y], [C,Z] ]
```

Occasionally we optimise the construction of particularly heavily used values, for example, by flattening some of the list nesting.

```
{ (a, x), (b, y), (c, z) }  -->  [ A, X, B, Y, C, Z ]
```

2.4 Schemas

2.4.1 Simple Schema declarations

Consider a schema defined in Z as

```
Schema ≙ [ x:X; y:Y; z:Z ]
```

The Z declaration `∃ Schema ...` automatically exposes its components. So it is translated to the Prolog `Schema = [ [x,X], [y,Y], [z,Z] ]` to enable the same style of access to the components. If a component is not used in the subsequent definition, it can be translated to underscore to highlight this fact. For example, `Schema = [ [x,X], _, [z,Z] ]`.

2.4.2 Predicate part

If the schema has a non-trivial predicate part (which includes the explicit predicates, together with any implicit predicates in the declaration), this needs to be translated too. (Most of the schemas in DeCCo simply capture abstract syntax, and so have no non-trivial predicate part.) For example, consider

```
Bound ≙ [ lb,ub: ℕ | lb < ub ]
```

Then the Z declaration `∃ Bound ...` is translated to the Prolog

```
Bound = [ [lb,Lb], [ub,Ub] ],
Lb >= 0, Ub >= 0,
Lb < Ub
```

where the first Prolog predicate captures the implicit Z predicate in the declaration, and the second captures the explicit Z predicate in the bar part of the schema.

2.4.3 Nested schemas

If a schema is defined in terms of component schemas, the resulting form is flat in Z, and so should remain flat in Prolog. Consider

```
AnotherSchema ≙ [ a:A ; Schema ]
```

The Z declaration `∃ AnotherSchema...` exposes its components with no nesting of the structure of `Schema` present. So it is translated to the Prolog

```
AnotherSchema = [ [a,A], [x,X], [y,Y], [z,Z] ]
```

to enable the same style of access to the components. A further clause of

```
Schema = [ [x,X], [y,Y], [z,Z] ]
```

may also be added if access is required to the Schema sub-component as a whole (usually by a theta expression). Unification ensures these two clauses refer to the same `X`, `Y` and `Z`, as required by the Z meaning.

2.4.4 Decorated Schema declarations

If a decorated version of the schema is declared in Z, as say, `∃ Schema' ...`, this needs to be translated into Prolog as `Schema1 = [ [x,X1], [y,Y1], [z,Z1] ]`. Note how the constant names remain the same; it is the variable names that are decorated.

2.4.5 Theta expressions

As well as accessing individual components, the whole binding can be accessed in Z using a theta expression, as `θ Schema`. In Prolog, this just becomes a reference to the variable `Schema`.

2.4.6 Schema as predicate

It is possible in Z to use a schema reference as a predicate. For example in the quantifier `∀ Module | ModuleDeclOkay • ...` the schema `ModuleDeclOkay` is being used as a predicate. Z type rules ensure that any variables used in this predicate are in scope. So this can be translated into Prolog just by translating its predicate part (which, as ever, includes the explicit predicates together with any implicit predicates in the declaration).

2.5 Modelling Z functions

2.5.1 Functions as sets of pairs (DeCCo environments)

Technically, a function in Z is merely a relation (a set of pairs of values) with a uniqueness property[5]. DeCCo uses functions to model environments (mappings from names to semantic values), and the semantic equations describe how these environments are built up and accessed. Hence translating these Z environment functions (explicit sets of pairs) to Prolog lists of 2-element lists is appropriate.

```
{a ↦ x, b ↦ y, c ↦ z}  -->  [[a, X], [b, Y], [c, Z]]
```

2.5.2 Functions as a computations (Z toolkit definitions)

Sometimes an explicit representation of a function as a set of pairs is not the most efficient way to model it in Prolog: we may wish to capture the *computation* represented by the function, along with the ability to *apply* the function to an argument, in which case we translate the computation, as discussed in the next section.

The Z mathematical toolkit [Spivey 1992, chapter 4] provides a collection of utility functions that are much used in Z specifications, including the DeCCo specifications. The ones that are used in DeCCo are translated into the relevant Prolog clauses that perform the computation they specify.

Consider domain restriction, defined in Z as

```
[X,Y]
_ ◁ _ : ℙ X × (X ⟷ Y) ⟶ (X ⟷ Y)

∀ a: ℙ X; r: X ⟷ Y •
  a ◁ r
   = { x: X; y: Y | x ∈ a ∧ (x,y) ∈ r • x ↦ y }
```

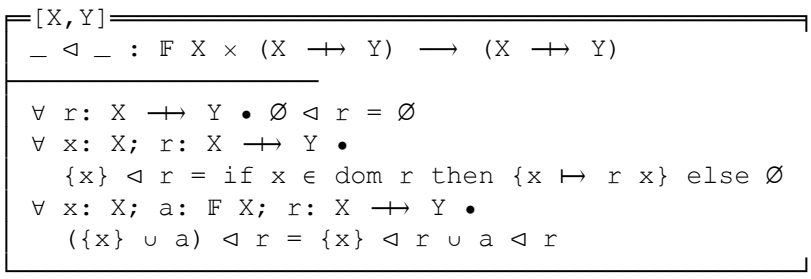
2.5.2.1 A recursive rewriting

As it stands, this is not in a useful form to translate directly into Prolog. An equivalent recursive form, explaining the effect of restricting to an empty set, a singleton set, and a union of sets, is more useful[6].

---

[5] This uniqueness property is a *semantic* property, and so violations are not checkable by Z typecheckers such as fuzz or Formaliser. In the constructive style of specification used for DeCCo, the functionality property is usually obvious, because the function is constructed using override. Occasionally functionality has to be further justified.

[6] There are various things about this definition that make it not equivalent to the previous one. It is equivalent only if `r` is functional, so that `y` is unique, and if `a` is finite, so that stripping elements out of `a` one at a time eventually terminates, and if the element `x` is not



This "readily translatable form" is suitable to be translated directly into Prolog. A Z function from several arguments to a result has the form in Prolog of

```
fun(Arg1, Arg2, ... Argn, Res) :- ...
```

In this case the general form is

```
domRes(A, R, Res) :- ...
```

For each case of the arguments, the left hand side matches the pattern of arguments, and the right hand side, defining `Res`, is a translation of the body of the quantifier. So

```
∀ r: X ⇸ Y • ∅ ◁ r = ∅
```

has an empty `a` and a general `r`, and so translates to

```
domRes(A, R, Res) :- A = [], Res = [].
```

which can be abbreviated to

```
domRes([], R, []).
```

Applying this to each of the three cases, we get the full translation (where `in` and `union` are translations of Z's membership and set union respectively) as

```
domRes([], R, []).

domRes([X], R, Res) :-
  (  in([X,Y], R)
  -> Res = [[X,Y]]
  ;  Res = []
  ).

domRes([X|A], R, Res) :-
  domRes([X], R, R1),
  domRes(A, R, R2)
  union(R1, R2, Res).
).
```

---

left in the remaining `a`. Let us assume in this example that this is the case, as it is the case in the DeCCo application.

In general, we do not explicitly give the recursive Z form of the toolkit operators, because the unwinding pattern is obvious. And we make sure that the DeCCo specification is written in such a "readily translatable form" initially.

2.5.2.2 Accumulator optimisation

Although the above is a correct translation of the Z function, it is rather inefficient, especially if the set `a` is large, because of the $O(\#a)$ unions of separate elements. There is a standard Prolog optimisation for this kind of operation, that uses an intermediate *accumulator* variable, storing the partial "answer so far" [Sterling, section 7.5]. When the recursion reaches the base case, the accumulated "answer so far" is the result of the complete call.

We also take the opportunity to merge the singleton and general case of the recursive definition, giving the source Z form as:

$$
\begin{array}{|l}
[X,Y] \\
\_ \lhd \_ : \mathbb{F}\, X \times (X \pfun Y) \fun (X \pfun Y) \\
\hline
\forall\, r : X \pfun Y \bullet \emptyset \lhd r = \emptyset \\
\forall\, x : X; a : \mathbb{F}\, X; r : X \pfun Y \bullet \\
\quad (\{x\} \cup a) \lhd r = \\
\qquad (\mathbf{if}\ x \in \mathrm{dom}\ r\ \mathbf{then}\ \{x \mapsto r\ x\}\ \mathbf{else}\ \emptyset) \cup a \lhd r
\end{array}
$$

A Z function from several arguments to a result in Prolog, with an accumulator, has a form of

```
fun(Arg1, Arg2, ... Argn, Res0, Res) :- ...
```

and is initially called with an empty accumulator:

```
fun(Arg1, Arg2, ... Argn, [], Res).
```

In this case the general form is

```
domRes(A, R, Res0, Res) :- ...
```

The full translation is

```
domRes([], R, [], Res).

domRes([X|Xs], R, Res0, Res) ::-
  (  in([X,Y], R)
  -> add(Res0, [X,Y], Res1)
  ;  Res1 = Res0
  ),
  domRes(Xs, R, Res1, Res).
```

And we could define the previous domain restriction as

```
domRes(A, R, Res) :- domRes(A, R, [], Res).
```

We do this on occasion without explicitly stating that we are using an accumulator.

3. DCTGs

To ease the mapping process, the Prolog implementation uses DCTGs (Definite Clause Translation Grammars), allowing it to have a structure closer to the Z. See [Abramson 1989, chapter 9], [Stepney, section 3.3].

3.1 Plain DCTGs

The general discussion and implementation in given in [Abramson 1989], with an example in the DeCCo context in [Stepney]. Essentially, a few new operators are defined in Prolog to allow syntax and semantic definitions to be structured thusly:

```
syntax
<:>
(semantics1),
(semanticsN).
```

3.1.1 Syntax

The syntactic portion defines how a production is constructed from non-terminal sub-productions and from terminal tokens. The various non-terminal constructs are labelled with derivation trees that are then referenced in the semantic portions. For example, an IF statement might have syntax portion like

```
cmd ::= tIF, expr^^E, tTHEN, cmd^^Ct, tELSE, cmd^^Ce
```

where `::=` and `^^` are DCTG operators, the Prolog variables `E`, `Ct` and `Ce` hold the derivation subtrees for the non-terminals, and the constants **`tIF`**, **`tTHEN`**, and **`tELSE`** represent tokens.

3.1.2 Semantics

The derivation trees introduced in the syntax part can be referenced in the semantics. For example, the dynamic semantics, that defines the state transitions, derived from a Z specification like

```
MC : CMD ⤔ State ⤔ State
─────────────────────────
∀ e : EXPR; ct, ce: CMD; s : State •
  ∃ v : VALUE ; s′ : State |
      v = ME e s
      ∧ s′ = if v = true then MC ct s else MC ce s •
    MC ifStmt(e, ct, ce) s = s′
```

[7]might look like

```
m(S, S1) ::-
  E ^^ m(S, V),
  ( V = true,
    Ct ^^ m(S, S1)
  ;
    V = false,
    Ce ^^ m(S, S1)
  )
```

where `::-` is a DCTG operator. This has the effect of evaluating the dynamic semantics of the expression `E` (as identified in the syntax) in the current state `S` to give a resulting value `V`. Depending on the value of `V`, the final state `S1` is determined by evaluating the dynamic semantics of command `Ct` or of `Ce` (similarly identified in the syntax).

Notice that there are several meaning functions in the typed Z, `MC` for commands, `ME` for expressions, and so on, but only one in the untyped Prolog, `m`. Prolog unification ensures the correct `m` is evaluated.

3.1.3 Prolog equivalent

The Prolog corresponding to this DCTG is

```
cmd( node( cmd, [ E, Ct, Ce ],
             [ m(S, S1) ::-
                 E ^^ m(S, V),
                ( V = true, Ct ^^ m(S, S1)
                ; V = false, Ce ^^ m(S, S1) ) ] ),
     D0, D ) :-
  expr( E, D0, D1 ),
  cmd( Ct, D1, D2 ),
  cmd( Ce, D2, D ).
```

This shows how the effect of the DCTG is to send calls to the subtrees, as expected, and to add some intermediate “accumulator” variables.

---

[7] This Z specification might more naturally be written as

```
MC : CMD ⇸ State ⇸ State
─────────────────────────
∀ e : EXPR; ct, ce: CMD; s : State •
  MC ifStmt(e, ct, ce) s
    = if ME e s = true then MC ct s else MC ce s
```

but here we have written it in the “unwound” style, to correspond more closely with the Prolog implementation.

3.1.4 Multiple semantics

Other semantics can be evaluated using the same trees. For example, the operational semantics, where the "meaning" is simply the list of assembly language instructions produced by the compiler, derived from a Z specification like

```
OC : CMD ⤔ seq INSTR
─────────────────────
∀ e : EXPR; ct, ce: CMD •
  ∃ eIs , ctIs, ceIs : seq INSTR |
      eIs = OE e
      ∧ ctIs = OC ct
      ∧ ceIs = OC ce •
    OC ifStmt(e, ct, ce)
      = eIs ⌢ ⟨ jmp l1 ⟩ ⌢ ctIs ⌢ ⟨ goto l2, lab l1 ⟩
        ⌢ ceIs ⌢ ⟨ lab l2 ⟩
```

[8]might look like

```
compile(Is) ::-
  E ^^ o(EIs),
  Ct ^^ o(CtIs),
  Ce ^^ o(CeIs),
  Is = [EIs, jmp(L1), CtIs, goto(L2), lab(L1), CeIs, lab(L2)]
```

The expression `E` and commands `Ct` or of `Ce` are again those identified in the syntax.

3.2 Extensions for free types

The Z specification makes heavy use of free types to define its abstract syntactic categories. The extended DCTG colon notation is used to capture this in Prolog. To capture a syntactic free type like

```
STMT ::= ... | ifStmt ⟪ EXPR × STMT × STMT ⟫ | ...
```

the DCTG colon operator is used thus:

```
sTMT ::= ...
sTMT ::= ifStmt(E:eXPR, S1,S2:sTMT)
sTMT ::= ...
```

---

[8] Again, this Z specification might be more naturally written as

```
OC : CMD ⤔ seq INSTR
─────────────────────
∀ e : EXPR; ct, ce: CMD •
  OC ifStmt(e, ct, ce) =
     OE e ⌢ ⟨ jmp l1 ⟩ ⌢ OC ct ⌢ ⟨ goto l2, lab l1 ⟩
     ⌢ OC ce ⌢ ⟨ lab l2 ⟩
```

but here we have written it in the "unwound" style.

Then, if the Prolog term `ifStmt(E, S1, S2)` is used somewhere in the body of a definition, the variable `E` unifies to the expression tree, and the variables `S1` and `S2` unify to the relevant statement trees.

For example, we have the Z specification for the dynamic semantics of the byte multiplication operator as

```
VALUE ::= ... | vbyte ⟪ BYTE ⟫ | vbool ⟪ BOOLEAN ⟫ | ...
BINOP ::= ... | bmul | ...

┌──────────────────────────────────────────────────
│ MBO : BIN_OP ⟶ VALUE × VALUE ⤔ VALUE
├──────────────────────────────────────────────────
│ ∀ b,c : BYTE •
│   MBO bmul (vbyte b, vbyte c) = vbyte (b * c)
```

with the Prolog implementation:

```
vALUE ::= vbyte(V:bYTE)
vALUE ::= vbool(V:bOOLEAN)
bINOP ::= bmul

bIN_OP ::= tBMUL
<:>
m(B, C, BRes) ::- !
  vbyte(B),
  vbyte(C),
  BRes is B * C,
  vbyte(BRes).
```

3.3 Concrete Syntax and parsing

Concrete syntax includes terminal tokens, and uses the DCTG `^^` notation to attach trees to non-terminals. Abstract syntax has no concrete "punctuation", and uses free type structures with the colon notation. The parser gives the link between the two, as a list of facts linking the two forms, as

```
absConc( abstract form, concrete form ).
```

For example

```
absConc ( ifStmt(E:eXPR, S1,S2:sTMT) ,
          ( tIF, eXPR^^E, tTHEN, sTMT^^S1, tELSE, sTMT^^S2) ).
```

The abstract syntax form, rather than the concrete syntax form, is used in the DeCCo implementation of the DCTG.

4. Semantic functions

4.1 Declaration

The DeCCo semantic functions are specified as Z mappings from an abstract syntactic construct to its semantic value. In general each function has a Z declaration like

```
M : S ⤔ X ⤔ Y ... ⤔ Z
```

where `M` is the name of the function, `S` is the abstract syntax category, and `X`, `Y`, … `Z` are the appropriate semantic values.

4.2 Specification

The function `Ms` is specified in a uniform manner, with a universal quantification over each of its curried arguments. If the arguments are not schema types, its specification is something like:

```
──────────────────────
∀ s1:S1 ; s2:S2 ; x:X ; y:Y •
  Ms f(s1, s2) x y = some expr
```

If the arguments are schema types, its specification is essentially:

```
──────────────────────
∀ S ; X ; Y •
  Ms θS θX θY = some expr
```

(Mixtures of the two kinds of arguments may be used.)

In practice, temporary variables are often introduced by existential quantification to "unwind" nested definitions, to show how the semantics of the whole is constructed from the semantics of the sub-expressions. For example, consider

```
──────────────────────
∀ s1:S1 ; s2:S2 ; x:X ; y:Y •
  ∃ y′,y′′:Y ; z:Z |
      y′ = Ma s1 x y
      ∧ y′′ = Ma s2 x y′
      ∧ z = g(y′, y′′) •
    Ms f(s1, s2) x y = z
```

Remember that the first argument of a Z meaning function, here `s1`, `s2` and `f(s1,s2)`, is the *syntactic* argument, so corresponds to a DCTG derivation tree, whereas the remaining arguments, here `x`, `y`, `y′`, `y′′` and `z`, are *semantic* arguments, so correspond to ordinary Prolog variables. So, in Prolog, this becomes

```
s ::= f(Tree1:s1, Tree2:s2)
<:>
m(X, Y, Z) ::-
  Tree1 ^^ m(X, Y, Y1),
  Tree2 ^^ m(X, Y1, Y2),
  g(Y1, Y2, Z).
```

There are several things to note:

- **declarations**: Prolog does not declare its variables, so the quantifiers have disappeared. The universally quantified arguments become the arguments (and result) to the Prolog meaning function. The existentially quantified temporaries become intermediate values in the Prolog computation. The Prolog may have yet further intermediate values to unwind some of the nested Z expressions.
- **ordinary functions**: are translated in the same way as toolkit functions.
- **meaning functions and syntax**: meaning functions use a special DCTG syntax to separate out the syntactic and semantic arguments.

5. Partiality

Some of the meaning function specifications are partial; that is, they do not define a result for every possible combination of their arguments. This is indicated by the use of a partial function arrow from syntactic to semantic values: `mf : SYN ⇸ Sem`.

There are several reasons for this partiality, with correspondingly different treatments in the implementation. Each stage of the processing (parsing, symbol checking, type checking,…) reduces the space of programs passed to the next stage. Each stage need be total only for programs passed to it (except for dynamic run-time checks). Rather than clutter the specification by stating the explicit domain on which each stage is total within each meaning function (which is simply those programs that have passed the previous stages), the various meaning functions are written as partial on the domain of all programs.

First we describe how to see where the partiality occurs in the specification, then describe the various cases, and how they are handled in the implementation. The only interesting form of partiality corresponds to a *dynamic (run-time) error*, such as division by zero, or array bounds error, with a corresponding *dynamic check* in the Pasp interpreter.

5.1 Spotting partiality

There are various ways to spot a partial definition in the specification.

5.1.1 Declared arguments do not match function signature

A definition may be partial if the declared arguments do not match exactly the function signature, and the expression defining the particular meaning function has a special case in its argument list.

This happens everywhere for the syntactic argument, where the definition is broken down into cases over the structure of the syntactic category. For example, consider the case of symbol declaration semantics of expressions. A total definition would look like:

```
TE : EXPR ⟶ EnvT ⇸ ExprType
─────────────────────────────
∀ e : EXPR; ρτ : EnvT •
  TE e ρτ = ...
```

Notice how the arguments to the quantifier are the same types as in the meaning function signature. However, the definition for constant expressions looks like

```
TE : EXPR ⤔ EnvT ⤔ ExprType
─────────────────────────
∀ κ : VALUE; ρτ : EnvT •
  TE (constant κ) ρτ = ...
```

The general expression argument is replaced by the particular branch being defined. There is a separate definition for each branch, which, taken together, cover all of `EXPR`. So each individual definition is partial, but their conjunction is total.

The Z could be rewritten equivalently as

```
TE : EXPR ⤔ EnvT ⤔ ExprType
─────────────────────────
∀ ε : EXPR; ρτ : EnvT | ε ∈ ran constant •
  ∃ κ : VALUE | ε = constant κ •
    TE ε ρτ = ...
```

So we see that the actual argument being more restricted than the formal argument is in fact a special case of our next case: a bar part.

5.1.2 Implication, or bar part in universal quantifier

A definition may be partial if there is an implication, or (equivalently) a bar part in a universal quantifier, putting constraints on the allowed values of expressions. For example, consider the case of type checking semantics of actual parameters.

```
TAP : EXPR ⤔ IDTYPE ⤔ EnvT ⤔ CHECK
─────────────────────────
∀ ε : EXPR; FormalType; ρτ : EnvT | c = val ∧ A = ∅ •
  ...
```

The bar part is what concerns us here. `c = val` constrains the actual parameter to be a call-by-value parameter. There is an accompanying definition to cover the `c = ref` case. `A = ∅` constrains the call-by-value parameter to have no attributes. This is another constraint that is true from context: symbol declaration type checking would have failed if there were attributes; this function is only ever called if symbol declaration checking succeeds.

5.1.3 Partial arrow in the signature

A definition may be partial if there is a partial arrow in the signature of the meaning function.

The symbol declaration semantics is (nearly) always total, because it must be applicable to any input program. (It is not always total in presentation, because it is total only on all *concrete* programs, which cover a space slightly smaller than the space of abstract programs.)

The later semantics are all partial, because they are defined only for programs that have passed the earlier semantics. The actual domain is not always made explicit, for simplicity. For example, that the argument to a function application is in the domain

of the function is not always stated explicitly: it is so because a previous semantics ensured it.

5.2 Implementing partiality

If a specification is partial, defined only for some inputs, what is to be done for inputs not specified? That depends on the reason for partiality.

5.2.1 Split into cases

Apparent partiality that is merely present to split the specification into cases is not an issue, because the cases taken together form a specification total on that category. The predicates partition the cases.

The partiality predicate indicating the case split is implemented in both the Pasp interpreter and in the compiler, as it serves to distinguish the disjoint cases. The disjunction of all these predicates gives *true*, and so there is there are no further case to consider.

That the predicates do indeed partition the cases is clear from inspection of the specification.

5.2.2 Syntactically disallowed behaviour

Some specifications are partial because the abstract syntax is wider than the concrete syntax. The partiality restricts the specification to the case of valid concrete syntax. For example, when importing a variable, the concrete syntax requires that variable to be read only, whereas the abstract syntax allows arbitrary attributes. The meaning function considers only the valid concrete cases:

```
DIM : IMPORT_DECL ⇸ STACK ⇸ EnvDTrace ⇸ EnvDTrace
─────────────────────────────────────────────────
∀ VarDecl; B : STACK; ρδt : EnvDTrace |
      A = ⟨readOnly⟩ ∧ V = ⟨⟩ •
  ...
```

The partiality predicate need not be implemented, because it is always *true*, and so there is no further case to consider. However, there is no harm in implementing it (apart from a trivial performance penalty), in order to make the demonstrability arguments clearer.

That the partiality predicate is indeed *true* can be determined by inspecting the relevant constraint in the syntax specification.

5.2.3 Statically undefined behaviour

All semantics later than the first are partial, because they are called only in the context of having successfully passed earlier tests. Most of these conditions are implicit (because it would merely be repeating the earlier semantics to make them explicit –

which would defeat the whole purpose of separating out the checks into separate passes).

Consider

```
TAP : EXPR ⤔ IDTYPE ⤔ EnvT ⤔ CHECK
─────────────────────────────────
∀ ε : EXPR; FormalType; ρτ : EnvT •
  TAP (formalType θ FormalType) ρτ = ...
```

Here the argument list is partial, because the `IDTYPE` must be a `FormalType`. There are no other definitions covering the other cases of `IDTYPE`. This is known to be the only possible case by *context*: this function is only ever called from a context where the expression being checked *is* an actual parameter expression, and hence will have the stated `IDTYPE`.

If the conditions are implicit, there is no partiality predicate to implement. Since the implicit condition is *true*, there are no further cases to consider.

That the implicit condition is indeed *true* can be determined by inspecting the relevant earlier static semantics specifications, and seeing that they fail to check in the cases where the condition would be *false*.

5.2.4 Dynamically undefined behaviour

An implication, or bar part in a universal quantifier, can introduce true partiality: cases where there is no definition of what happens in some cases, and where the function may indeed be called in those cases. These correspond to dynamic (run-time) conditions, because the values in the partiality predicate can be determined only at run-time.

For example, the specification of the dynamic semantics of the binary operator `bdiv` is

```
MBO : BIN_OP ⟶ VALUE × VALUE ⤔ VALUE
─────────────────────────────────
∀ b,c : BYTE | c ≠ 0 •
  MBO bdiv (vbyte b, vbyte c) = vbyte (b div c)
```

This is the entire specification for this operator, and so it does not say what happens in the case when `c` is zero. *Any* implementation in the case when `c` is zero is allowed by such a specification. The interpreter traps the case, and issues an error message. So the Prolog implementation is:

```
bIN_OP ::= tBDIV
<:>
dynBO(B, C, BRes) ::- !
  vbyte(B),
  vbyte(C),
  (  C > 0
  ;  dynamicError(bdiv, zero)
  ),
  BRes is B // C,
  vbyte(BRes).
```

The compiled code, on the other hand, has no check for zero. (The executing code will actually halt.)

The cases where there are such dynamic checks in the interpreter are:

- underflow and overflow in arithmetic operations
- divide by zero in arithmetic operations
- subrange checking (on function return, assignment)
- array bounds accessing

5.3 Summary

- **Explicit dynamic partiality predicate**. The predicate is implemented in the Pasp interpreter, guarding the implementation of the specified behaviour. The negation of the predicate, which indicates a run-time error, is implemented in the interpreter, guarding the execution of an error report. There is no implementation of the predicate or its negation in the compiler. All such cases are explicitly flagged in the specification commentary.
- **Explicit syntactic or static partiality predicate**. The guard indicates a condition trapped by a previous syntactic or semantic pass, which is *true* in this pass. The redundant predicate need not be implemented, but may be, to aid demonstrability arguments. Because it is *true*, there is no further case to consider.
- **Implicit partiality predicate**. There is nothing explicit to implement. Because it is *true*, there is no further case to consider.
- **Case partiality predicates**. The predicates are implemented in the Pasp interpreter and the compiler, to distinguish the cases they guard. They disjoin to *true*, so there is no further case to consider.

6. Examples

6.1 Unary Op pred, dynamic semantics (Pasp §6.2.4)

Demonstrates the translation of a dynamic check partial predicate in the dynamic but not in the operational semantics.

The Z specification of the dynamic semantics of the unary operator `unsgnToByte` is

```
MUO : UNY_OP ⟶ VALUE ⟶ VALUE
─────────────────────────────
∀ n : UNSIGNED | n ∈ BYTE •
  MUO unsgnToByte (vunsgn n) = vbyte n
```

The Prolog implementation is (remember, there is no need for the syntactic argument in the argument list of the meaning function: that is handled by the DCTG).

```
uNY_OP ::= unsgnToByte
<:>
m(N, ResB) ::- !
  vunsgn(N),
  (  N < 256
  ;  dynamicError(unsgnToByte, overflow)
  ),
ResB is N,
vbyte(ResB).
```

The Z constraint `n ∈ BYTE`, equivalent to `0 ≤ n < 256`, has been optimised to neglect the unnecessary lower bound check. The test is performed as soon as possible, because in some cases (for example, division by zero) the Prolog would be unhappy if the calculation were performed.

The operational semantics of the unary operator `unsgnToByte` is specified as

```
OUO : UNY_OP ⟶ LABEL ⟶ LABEL × seq X_INSTR
──────────────────────────────────────────
∀ l : LABEL •
  OUO unsgnToByte l = (l, ⟨ ⟩)
```

The operational semantics has been *totalised*, corresponding to a particular implementation decision about the partiality in the dynamic Pasp spec: overflow truncates.

The Prolog implementation is

```
o(Label, (Label, XIs)) :-
  XIs = [ [] ].
```

Notes:

1. The expected `LabelRes = Label` clause has been optimised away.
2. The `XIs` clause is not optimised away, because the full implementation has an extra comment "instruction".
3. The full implementation is split between two separate Prolog clauses, with the syntactic `unsgnToByte` occurring explicitly in the second, because of an implementation decision. The second of these operational semantics clauses, for the case of operators, is defined in a separate file, to modularise certain common instructions. The `unsgnToByte` tag acts as a link between these files.

6.2 Constant, type checking (Pasp §7.4.2)

The Z specification for type checking a constant is

```
TE : EXPR ⇸ EnvT ⇸ ExprType
─────────────────────────────
∀ κ : VALUE; ρτ : EnvT •
  TE (constant κ) ρτ = ⟨| τ == constType κ, A == ∅ |⟩
```

The corresponding Prolog translation is

```
eXPR ::= constant(K:vALUE)
<:>
typeE(_, ExprTypeRes) ::- !
  K ^^ constType(Tau),
  ExprTypeRes = [ [tau,Tau], [a,[]] ].
```

Notes:

1. The Z meaning function `TE` translates to the one argument, one result predicate `typeE` in Prolog, because the syntactic argument is handled by the DCTG.
2. Because the argument ρτ is not used, it is translated into a Prolog underscore.
3. The Z function `constType : VALUE ⟶ TYPE` is treated as a semantic function (it takes a syntactic argument). So in Prolog its single argument, κ, is passed by the DCTG, and it has a single result.

6.3 Unary expression, dynamic semantics (Pasp §7.6.4)

The Z specification is

```
ME : EXPR ⇸ STACK ⇸ EnvMTrace ⇸ State ⟶ ExprValue
──────────────────────────────────────────────────
∀ UnyExpr; B : STACK; ρt : EnvMTrace; σ : State •
  ∃ ExprValue′; ExprValue′′ |
      θ ExprValue′ = MeanE ε B ρt σ
    ∧ θ State′′ = θ State′
    ∧ v′′ = MeanUnyop Ψ v′ •
   MeanE (unyExpr θ UnyExpr) B ρt σ = θ ExprValue′′
```

So the corresponding Prolog translation is

```
eXPR ::= unyExpr(Op:uNY_OP, E:eXPR)
<:>
meanE(B, RhoT, Sigma, ExprValRes) ::- !
  ExprValue1 = [ [sigma,Sigma1], [v,V1] ],
  ExprValue2 = [ [sigma,Sigma2], [v,V2] ],
  E ^^ meanE(B, RhoT, Sigma, ExprValue1),
  Sigma2 = Sigma1,
  Op ^^ meanUnyOp(V1, V2),
  ExprValRes = ExprValue2.
```

Notes:

1. The Z schema declaration `ExprValue'` translates to the Prolog `ExprValue1 = [ [sigma,Sigma1], [v,V1] ]`, which exposes the schema components for later use.
2. The meaning function result `θ ExprValue'` translates to `ExprValue1`
3. Calculation of the meaning of the sub-expression gives the result `ExprValue1`, which in turn gives `Sigma1` and `V1`.

6.4 Actual parameter, type checking (Pasp §7.3.2)

The Z specification for the `c = val` case is

```
TAP : EXPR ⤔ IDTYPE ⤔ EnvT ⤔ CHECK
─────────────────────────────────
∀ ε : EXPR; FormalType; ρτ : EnvT | c = val ∧ A = ∅ •
  ∃ ExprType' | θ ExprType' = TypeE ε ρτ •
    TAP ε (formalParam θ FormalType) ρτ
     = if τ' = τ ∧ writeOnly ∉ A'
       then checkOK else checkTypeWrong
∀ ε : EXPR; FormalType; ρτ : EnvT | c = ref •
  ...
```

The corresponding Prolog translation is

```
aCTUAL ::= actualParam(E:eXPR)
<:>
typeAP(Idt, EnvT, CRes) ::- !
  Idt = formalParam(FormalType),
  FormalType = [ _, [c,C], [a,A], [tau,Tau], _, _],
  (  C = val, A = [],
  -> ExprType1 = [ [tau,Tau1[, [a,A1] ],
     E ^^ typeE(EnvT, ExprType1),
     (  Tau1 = Tau, notMember(writeOnly, A1)
     -> CRes = checkOk
     ;  CRes = checkTypeWrong
     ),
  ;
    "C = ref case ..."
  )
```

Notes:

1. The non-syntactic parameter `Idt` is partial here, so we need to construct it explicitly in the Prolog.

6.5 Variable declaration, type checking (Pasp §9.4.2.7)

Demonstrates using theta bindings of parts of declared schema.

The Z specification is

```
TypeV : VarDecl ⤔ STACK ⤔ EnvTTrace ⟶ EnvTTrace
─────────────────────────
∀ VarDecl; B : STACK; ρτt : EnvTTrace •
  ∃ VarType′; c,c′ : CHECK; idt : IDTYPE |
        (c′,θ SubrangeType′) = TT τ B ρτt
        ∧ θ ArrayType′ = TSRs SR B ρτt
        ∧ c = TI (θ VarDecl, θ SubrangeN′, numElts SR′)
        ∧ A′ = ran A
        ∧ idt = if TAs A ⋈ c ⋈ c′ = checkOK
                then variable θ VarType′
                else variable ⟨| A == ∅, SR == ⟨⟩, τa == τa′,
                         lb == 0, ub == 0, τ == typeWrong |⟩ •
    TV θ VarDecl B ρτt = update (ρτt,B,{ξ ↦ idt})
```

The corresponding Prolog translation is

```
vARDECL ::= varDecl(I:iD, A:seqATTR, SR:seqSUBR, T:tYPE,
                                              SV:seqVALUE)
<:>
typeV(B, RhoTauT, RhoTauTRes) ::- !

  VarType1 = [ [a,A1], [sr,SR1], [taua,TauA1],
                         [lb,LB1], [ub,UB1], [tau,Tau1]],
  SubrangeN1 = [ [lb,LB1], [ub,UB1] ],
  SubrangeType1 = [ [lb,LB1], [ub,UB1], [tau,Tau1] ],
  ArrayType1 = [ [sr,SR1], [taua,TauA1] ],

  T ^^ typeT(B, RhoTauT, [C1, SubrangeType1]),
  SR ^^ typeSRs(B, RhoTauT, ArrayType1),
  numElts(SR1, N), typeIFlat([I, A, SR, T, SV], SubrangeN1, N, C),
  A ^^ flat(ASeq), range(ASeq, A1),
  A ^^ TypeAs(C2),
  (  pess([C2, C, C1], checkOK)
  -> Idt = variable(VarType1)
  ;  Idt = variable([ [a,[]], [sr,[]], [taua,TauA1],
                           [lb,0], [ub,0], [tau,typeWrong] ]
  ),
  I ^^ flat(Id)
  update(RhoTauT, B, [Id,Idt], RhoTauTRes)
```

Notes:

1. The Z declaration `VarType′` translates to the Prolog `VarType1 = [[a,A1], [sr,SR1], [taua,TauA1], [lb,LB1], [ub,UB1], [tau,Tau1]]`, which flattened form exposes the all schema components for later use.
2. The component sub-schemas, `SubrangeN′`, `SubrangeType′`, and `ArrayType′`, used to define `VarType′` are also introduced in the Prolog, so that they can be used to translate various theta terms later.

3. The expression involving `numElts` is unwound to `n = numElts SR'`, in order to be translated.
4. The use of `TypeI` on the full syntactic argument is a rare usage. Usually meaning expressions are called on subcomponents of the syntax tree. The DCTG formulation does not directly support such usage. So `TypeI` is translated to a call to the ordinary Prolog predicate `typeIFlat`, which itself has the translation of the Z. Then this clause can make use of the same `typeIFlat`.
5. Before we can find the range of the sequence of the attributes `A` we need to extract the underlying sequence from within its DCTG superstructure, which is what `A^^flat(Aseq)` does. Similarly for `I` and `Id`.

7. References

[Abramson 1989] Harvey Abramson, Veronica Dahl. *Logic Grammars*. Springer. 1989.

[Clocksin 1984] W. F. Clocksin, C. S. Mellish. *Programming in Prolog*. 2nd edition. Springer. 1984

[Formaliser] *Z Specific Formaliser User Guide*. Logica. 1989-2000.

[ISO-Z] *Formal Specification -- Z Notation -- Syntax, Type and Semantics*. International Standard. ISO/IEC 13568. 2002.

[Spivey 1992] J. M. Spivey. *The Z Notation – a reference manual*. 2nd edition. Prentice Hall. 1992.

[Stepney *et al*] Susan Stepney, Dave Whitley, David Cooper, Colin Grant. A Demonstrably Correct Compiler. *Formal Aspects of Computing*, 3:58-101. BCS, 1991.

[Stepney 1993] Susan Stepney. *High Integrity Compilation*. Prentice Hall. 1993.

[Stepney 1998] Susan Stepney. Incremental Development of a High Integrity Compiler: experience from an industrial development. In *Third IEEE High-Assurance Systems Engineering Symposium (HASE'98), Washington DC* 1998.

[Sterling] Leon Sterling, Ehud Shapiro. *The Art of Prolog: advanced programming techniques*. MIT Press. 1986.

[Stringer-Calvert *et al*] David W. J. Stringer-Calvert, Susan Stepney, Ian Wand. Using PVS to prove a Z refinement: a case study. In *FME '97, Graz 1997*. LNCS vol 1313. Springer, 1997