Electronic Notes in Theoretical Computer Science 137 (2005) 111–129 

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

The Specification Logic *ν*Z

# Martin C. Henson[1](#_bookmark0) Besnik Kajtazi[2](#_bookmark0)

*Department of Computer Science University of Essex*

*Wivenhoe Park, Colchester, Essex, CO4 3SQ, UK*

Abstract

This paper introduces a wide-spectrum specification logic νZ. The minimal core logic is extended to a more expressive specification logic which includes a schema calculus similar (but not equiva- lent) to Z, some new additional schema operators and extensions to a programming and program development logic.

*Keywords:* Specification Language, Wide Spectrum Language, Specification Logic, Operation Refinement, Program Development.

# Introduction

In this paper we introduce a wide-spectrum logic *ν*Z. This is a very small spec- ification logic based on a total correctness relational semantics with refinement as its fundamental relation.

The language which underlies the logic is Z-like, that is to say, we have schemas and schema operators. A significant difference is that operation

schemas have two predicates, so resemble more the specification statements of the refinement calculus (*e.g.* [[1](#_bookmark2)] and [[6](#_bookmark7)]). This is, in fact, a fairly trivial differ- ence, and the language could easily be set up using single predicate schemas if preferred. On the other hand, there are several significant differences between *ν*Z and Z:

1 Email: [hensm@essex.ac.uk](mailto:hensm@essex.ac.uk)

2 Email: [bkajta@essex.ac.uk](mailto:bkajta@essex.ac.uk)

1571-0661 © 2005 Elsevier B.V. Open access under [CC BY-NC-ND license.](http://creativecommons.org/licenses/by-nc-nd/3.0/)

doi:10.1016/j.entcs.2005.04.027

* Z is based on a *partial* -correctness semantics; *ν*Z is based on a *total* -correctness semantics.
* Z permits refinement of over–specifications; *ν*Z does not.
* Z schema operators are not monotonic; *ν*Z schema operators are monotonic (anti-monotonic).
* Z is based on *equality* ; *ν*Z is based on *reﬁnement*.
* Z is a *speciﬁcation* language; *ν*Z is *wide-spectrum*.
* Z is relatively inflexible; *ν*Z is extensible.
* Z is a *language*; *ν*Z is a *logic*.

In this introductory paper we concentrate entirely on the system itself, its mathematical basis and methodologies for extending the core framework with additional features for specification, for programming and for program development. In future publications we will explore more pragmatic issues, providing techniques and examples to demonstrate how to effectively specify, refine and implement systems within *ν*Z.

# Core *ν*Z

*ν*Z is interpreted within the logic Z⊥, the extension of ZC introduced in [[3](#_bookmark3)] which includes ⊥ elements in all types. We assume familiarity with this theory (and notational conventions); all this is also covered in [[4](#_bookmark5)].

C

* 1. *Syntax of ν*Z

The syntax of the core *ν*Z framework is minimal. The type of an *operation schema*, *U* , is P *T* (written *U* P *T* ) where *T* is a schema type which has the form ∆*V* = *V* Y *V* '. Generally we will, as is usual in Z, write ∆*V* for *V* Y *V* '. We will write *U* (*v* ) to indicate that variable *v* may appear free in the schema

expression *U* . [3](#_bookmark1)

C

3 When the variable has the type P *T* and *T* is a schema type (that is: it is a variable over schemas) we shall write it in Z⊥, as we do in νZ, in upper-case.

Definition 2.1

*U* P *T* ::=

*X* P *T* – schema variable

[*T* | *P* | *Q* ] – atomic specifications

¬*U* P *T* – negation

*U* P *T*0 ∨ *U* P *T*1

(*T* = *T*

Y *T* ) – disjunction

0 1 0 1

∃ x*T*x • *U* P *T*0

0

(*T* = *T*0

— *T*x) – existential hiding

*µX* P *T* • *U* (*X* )P *T* – recursive schemas

* 1. *Semantics of ν*Z

We first need to define refinement. In this framework it is simply containment.

Definition 2.2

*U* P *T* ± *U* P *T* =

0

1

*df*

*U* ) ⊆

0

*T*

*U* )

We also need to specify the universe of specification models for a given type. Part (ii) is based on [[2](#_bookmark4)] section 8.1.

1

Definition 2.3

1. *magic*P *T* =

9

*df*

[*T* | *true* | *false*]

1. *WT*

=*df*

{ *U* P *T* | *magic*P *T* ± *U* ∧ *U* o *magic*P(∆*Tout* ) ± *U* }

Now we have the semantics of specifications.

Definition 2.4 In what follows, *T* ٨ =*df V*⊥ *\* V* ' . The types are omitted here, but are taken to be as specified in the syntax above.

⊥

*X* ) =*df X*

'

[*T* | *P* | *Q* ]) = {*z \* z* ∈ *T*

1

*df* 0 1

٨

٨ | *z*0*.P* ⇒ *z*0*.z* '*.Q* }

¬*U* ) =*df* {*z* ∈ *T*

| *z* =*V* ⊥∨ *z* /∈ *U* }

٨ . .

*U*0 ∨ *U*1) =*df* {*z* ∈ *T*

∃ x • *U*0) =*df* {*z* ∈ *T*

٨

| *z* ∈ *U*0) ∨ *z* ∈ *U*1)}

| ∃ *y* ∈ *T* ٨ • *y* ∈ *U* ) ∧ *z* = *y* T *T* }

0 0

*µX* • *U* (*X* )) =*df* .{*X* ∈ *WT* | *U* (*X* )) ± *X* }

In the case of recursion, the schema variable *X* must appear in a *positive* position in *U* . That is: this is monotone recursion. The notation *t.P* indicates the usual distribution of the binding *t* through the proposition *P* so that its component observations x are replaced by *t.*x. Note that ⊥ *.P* = *false* for all *P* (⊥ satisfies nothing, in particular it is outside every precondition).

* 1. *Logic of ν*Z

The semantics induces a logic for the constructs, as follows. In this introduc- tory paper we omit the proofs.

* + 1. *Reﬁnement*

The rules for operation refinement in *ν*Z are as follows:

Proposition 2.5 *Let z be a fresh variable.*

*z* ∈ *U*0 ▶ *z* ∈ *U*1 *(*±+*) U*0 ± *U*1 *t* ∈ *U*0 *(*±−*)*

*U*0 ± *U*1 *t* ∈ *U*1

* + 1. *Atomic operation schemas*

The rules for atomic operation schema in *ν*Z are as follows:

Proposition 2.6

*t*0*.P* ▶ *t*0*.t* ' *.Q*

1 +

*(U )*

*t*0 *\* t* ' ∈ [*T* | *P* | *Q* ]

1

*t*0 *\* t* ' ∈ [*T* | *P* | *Q* ] *t*0*.P*

*(U )*

1 −

*t* ' *.Q*

1

* + 1. *Negated schemas*

Note that negation in *ν*Z is not the relational inverse: it is well-known that the universe of total-correctness relations in this model is not closed under that operation (see *e.g.* [[2](#_bookmark4)]). An alternative characterisation of the semantics is available using a combination of relational inverse, disjunction and magic.

Definition 2.7

¬*U* = *U* −1 ∨ *magic*

In any event, the rules for negation are derivable:

Proposition 2.8

*t* /∈ *U (U* + *) t*0 =⊥

*(U* + *)*

*t* ∈ ¬*U* ¬*0*

1

*t*0 *\* t* ' ∈ ¬*U* ¬*1*

*t*0 *\* t* ' ∈ ¬*U t*0 *\* t* ' /∈ *U* ▶ *P t*0 =⊥ ▶ *P*

1 1 −

*(U )*

*P* ¬

The notion satisfies double negation and excluded middle.

Proposition 2.9

*t* ∈ *U t* ∈ ¬¬*U*

*t* ∈ ¬¬*U*

*t* ∈ *U t* ∈ ¬*U* ∨ *U*

* + 1. *Disjunction schemas*

The rules for disjunction schemas in *ν*Z are derivable, as follows:

Proposition 2.10 *Let i* ∈ 2*.*

. . .

*t* ∈ *Ui (U* + *) t* ∈ *U*0 ∨ *U*1 *t* ∈ *U*0 ▶ *P t* ∈ *U*1 ▶ *P*

*(U* −*)*

*t* ∈ *U*0 ∨ *U*1 ∨*i P* ∨

* + 1. *Existential hiding schemas*

The rules for existential hiding schemas in *ν*Z are derivable, as follows:

Proposition 2.11

*t* ∈ *U (U* +*) t* ∈ ∃ x*T*x • *U t \** ⟨| x$*y* |⟩ ∈ *U* ▶ *P* −

.

*t T*x ∃

*(U*∃ *)*

∈ ∃ x • *U P*

* + 1. *Recursive schemas*

The rules for recursive schemas in *ν*Z are derivable, as follows:

Proposition 2.12

*t* ∈ *U* (*µX* • *U* (*X* )) *(µ*+*) t* ∈ *µX* • *U* (*X* )

*(µ*−*)*

*t* ∈ *µX* • *U* (*X* ) *t* ∈ *U* (*µX* • *U* (*X* ))

# Specifying a specification language in *ν*Z

The principles on which *ν*Z is based include *economy* (the core system begin so small) and *extensibility* (the ease with which the core system can be made

more expressive). Since the core system is so inexpressive, a first ambition will be to provide additional infrastructure which provides for a considerably

more expressive specification language. We cover some aspects of this in this section, beginning with extensions providing other standard schema operators. Some of the operators which we consider here are familiar from Z (though, because the semantics is differnt, the logic of these operators departs from that in Z). In addition there will be variations on familiar operators, such as composition: in this section we provide a notion of composition which allows

*arbitrary* schemas to be composed, even when those schemas do not match for

type. Finally, we introduce a range of quite new operators, unfamiliar in Z, which we will see have some use when we turn to the topic of programming languages and program development logics in later sections.

* 1. *Schema conjunction and implication*

We can define these in terms of disjunction and negation, using the usual de Morgan definitions. We omit the proofs, which are a little more involved than usual, due to the more complex notion of negation we are obliged to use.

Definition 3.1

*U*0 ∧ *U*1 =*df* ¬(¬*U*0 ∨ ¬*U*1)

The usual rules are derivable.

Proposition 3.2 *Let i* ∈ 2*.*

*t* ∈ *U*0 *t* ∈ *U*1 *(U* +*) t* ∈ *U*0 ∧ *U*1 *(U* − *)*

Definition 3.3

*t* ∈ *U*0 ∧ *U*1 ∧

*t* ∈ *Ui* ∧*i*

*U*0 ⇒ *U*1 =*df* ¬*U*0 ∨ *U*1

With the obvious rules derivable:

Proposition 3.4

*z* ∈ *U*0 ▶ *z* ∈ *U*1 *(U* + *) t* ∈ *U*0 ⇒ *U*1 *t* ∈ *U*0 *(U* − *)*

*z* ∈ *U*0 ⇒ *U*1 ⇒ *t* ∈ *U*1 ⇒

* 1. *Universal Hiding*

Universal hiding is defined in terms of existential hiding and negation, us- ing the standard de Morgan definition. We provide the proofs in detail, for illustration.

Definition 3.5

∀ x*T*x • *U* =*df* ¬∃ x*T*x • ¬*U*

And then the usual introduction and elimination rules are derivable.

Proposition 3.6 *Let z be a fresh variable. We assume that t has the form t*0 *\* t* ' *.*

1

Proof.

*t \** ⟨| x$*z* |⟩ ∈ *U*

*t* ∈ ∀ x*T*x • *U*

*t* = (*0*)

0 ⊥

*δ*.0

.

*t*0 =⊥∨ *t*0 /=⊥

*t* ∈ ¬ ∃ x*T*x • ¬*U t* ∈ ¬ ∃ x*T*x • ¬*U*

*t* ∈ ¬ ∃ x*T*x • ¬*U* (*0*)

where *δ*0 is:

*t* ∈ ∃ x*T*x • ¬*U* (*1*)

*δ*.1

.

*false*

and where *δ*1 is:

*false* (*z*)

*t* /∈ ∃ x*T*x • ¬*U* (*1*)

*t* ∈ ¬ ∃ x*T*x • ¬*U*

*t* ⟨| x$*z* |⟩ ∈ ч*U* (2)

*t* ⟨| x$*z* |⟩ /∈ *U* (3) *t* ⟨| x$*z* |⟩ ∈ *U false*

*t*0 =⊥ (3) *t*0 /=⊥ (0)

*false false*

*false* (3)

Proposition 3.7 *Let t have the form t*0 *\* t* ' *.*

1

*t* ∈ ∀ x*T*x • *U v* ∈ *T*x *t \** ⟨| x$*v* |⟩ ∈ *U*

Proof.

*t*0 =⊥∨ *t*0 /=⊥

(*LEM* )

*δ*.0

.

*t \** ⟨| x$*v* |⟩ ∈ *U*

*δ*.1

.

*t \** ⟨| x$*v* |⟩ ∈ *U*

*t \** ⟨| x$*v* |⟩ ∈ *U* (*0*)

where *δ*0 is:

*t*0 ⟨| x$*v* |⟩ /∈ *U* (1)

*t*0 ⟨| x$*v* |⟩ ∈ ч*U t*0 ⟨| x$*v* |⟩ ∈ *Ts*

*t*0 =⊥

(0)

*t*0 ⟨| x$*v* |⟩ ∈ *T*0 (2)

*.*

*t*0 ∈ *T*0

*t*0 /=⊥

*t*0 ⟨| x$*v* |⟩ =⊥ (2)

⟨| x$*v* |⟩ =⊥

*v* =⊥

*v* ∈ *T*x

*v* /=⊥

*t*0 ⟨| x$*v* |⟩ ∈ *T*0⊥

*false*

*false*

*false* (2)

*t* ⟨| x$*v* |⟩ ∈ *U* (1)

and *δ*1 is:

*t* ∈ ч Е x*T*x • ч*U*

*t \** ⟨| x$*v* |⟩ /∈ *U* (*q*)

*t \** ⟨| x$*v* |⟩ ∈ ч*U t* ∈ Е x*T*x • ч*U*

*false t \** ⟨| x$*v* |⟩ ∈ *U* (*q*)

*t*0 *\** ⟨| x$ ⊥|⟩ =/

*t*0 /=⊥

(*0*)

⊥

* 1. Ξ*-schemas*

We have the usual idea of Ξ-schemas:

Definition 3.8

Ξ*T* =*df* [∆*T* | *true* | *θT* = *θ*'*T* ]

The rules are straightforward:

Proposition 3.9

1

*t \* t* ' ∈ Ξ*T*

*t*0 *\* t* ' ∈ Ξ*T t*0 = *t*1

* 1. *The* skip*-extension*

We use this to define the skip-extension of a schema:

Definition 3.10 When *T*0 and *T*1 are disjoint, we define:

*U* P *T*0 ⬦ *T* = *U* Λ Ξ*T*

1 *df* 1

Naturally this is well-defined even when the types are not disjoint, but the purpose of this is, as described, to extend a schema with skip and the definition has pathological effects in other circumstances.

The rules are straightforward:

Proposition 3.11

.

*t*0 *\* t* ' ∈ *U t*0 =*T t*1

1 +

*(U )*

*t*0 *\* t* ' ∈ *U* ⬦ *T* o

1

*t* ∈ *U* ⬦ *T*

o*0*

.

*(U* — *)*

*t \* t* ' ∈ *U* ⬦ *T*

1 *(U* — *)*

0

*t* ∈ *U*

*t*0 =*T t*1 o*1*

* 1. *Schema Composition*

In *ν*Z we wish to compose *arbitrary* specifications; even when the types of the operations do not match. In this regard *ν*Z differs from Z. For such composi- tions to make sense, it is necessary to match incompatible types and to ensure that operations do not arbitrarily adjust bindings in the process. The defini- tion of schema composition in *ν*Z is, therefore, a little more complex than in

Z. Nevertheless, it is possible to specify composition in the core theory, using the skip-extension operator.

Definition 3.12 Let *T*

= *T* − *T*

with the form ∆*T*

= *Tin* Y *T out* '

and

*L* 1 0

' *L L L*

let Let *TR* = *T*0 − *T*1 with the form ∆*TR* = *Tin* Y *T out* . Let t be a vector of

*R R* ' '

*L*

fresh observations with the size of the alphabet of *T out*

0

Y *T out*

(equivalently:

*Tin* Y *Tin* ).

1 *R*

P(*T in* Y*T out* ' )

P(*T in* Y*T out* ' )  ' '

0

*L*

0 0 o 1 1

*U*

*U*

0 9 1

=*df* E t • (*U*0 ⬦ *TL*)[*α*(*T out*

Y *T out* )*/* t] Λ

(*U*1 ⬦ *TR*)[*α*(*Tin* Y *Tin* )*/* t]

1 *R*

The following introduction and elimination rules are derivable for schema composition:

Proposition 3.13

.

*t*0 *\* t* ' ∈ *U*0 *t*0 =*T*

.

*t*2 *t*2 *\* t* ' ∈ *U*1 *t*2 =*T t*1 +

2 *L* 1 *R* *(U*o *)*

*t*0 *\* t* ' ∈ *U*0 o *U*1 9

1 9

Proposition 3.14

. .

*t*0 *\* t* ' ∈ *U*0 o *U*1 *t*0 *\* t* ' ∈ *U*0*, t*0 =*T t*2*, t*2 *\* t* ' ∈ *U*1*, t*2 =*T*

*t*1 ▶ *P*

1 9 2 *L* 1 *R* —

*(U )*

o

*P* 9

* 1. *Restricted chaos*

This definition introduces a restricted form of *chaos*: outside *P* this schema blocks.

Definition 3.15

*chaosP* =*df* [*T* | ч*P* | *false*]

This leads to the following logical rules.

Proposition 3.16

*t*0*.P*

*(chaos*+*)*

*t*0 *\* t* ' ∈ *chaosP* ч*t*0*.P*

*(chaos )*

1 —

*t*0 *\* t* ' ∈ *chaosP* P *false* P

1

* 1. *Schema specialisation*

We use restricted chaos to introduce the specialisation of a schema at a par- ticular observation (it blocks elsewhere).

Definition 3.17 Let *ET* be the schema type corresponding to the observa- tions contained in *E* . Let ∆[x*T*x ] ≤ *T* .

P *T E*

P ∆([xTx ]YET)

*U* [x$*E*

*T* ] =*df chaos*(x=E) Λ *U*

This induces the following rules:

Proposition 3.18

.

*t* ∈ *U t.*x = *z.E t* ∈ *U* [x$*E* ]

*t* ∈ *U* [x$*E* ]

~~.~~

*t* ∈ *U*

*t* ∈ *U* [x$*E* ]

*t.*x = *t.E*

* 1. *Strengthening preconditions*

This operator has the effect of (in general) strengthening the precondition of a schema *U* by stipulating an additional condition *P* .

Definition 3.19 Let *TP* ≤ *T* .

*U* P *T* ↑ *PTP* = *chaos* ⇒ *U*

*df P*

The operator is governed by induced logical rules.

Proposition 3.20

*t.P* ▶ *t* ∈ *U t* ∈ *U* ↑ *P*

*t* ∈ *U* ↑ *P t.P t* ∈ *U*

# Specifying a programming language in *ν*Z

It is central to the methodology of *ν*Z that it smoothly integrates specification and programming, and that it is possible to develop programs from specifica- tions. This is achieved by firstly *specifying* a programming language in *ν*Z and then inducing a corresponding program logic: refinement then automatically

permits development from specifications to programs. We will develop such a language incrementally in this section.

*df*

* 1. skip

Definition 4.1

Rules for skip:

Proposition 4.2

skipP(∆*T*) = Ξ*T*

*t \* t* '

∈ skip

*(*skip+*)*

*t*0 *\* t* ' ∈ skip

*(*skip *)*

1 —

*t*0 = *t*1

Inequation:

Proposition 4.3

*θT* = *θ*'*T* ▶ *Q*

skip ± [*T* | *P* | *Q* ]

* 1. *Assignment*

Let *V* = *TE* − [*x T* ]

Definition 4.4

x:= *E* =*df* [∆[*x T* ] | *true* | *x* ' = *E* ] Λ Ξ*V*

Rules for assignment:

Proposition 4.5

*t \* t* '[x'*/t.E* ] ∈ x:= *E*

*(*:=+*)*

*t*0 *\* t* ' ∈ x:= *E*

*(*:= *)*

1 —

*t*0[x*/t*0*.E* ] = *t*1

Rule for assignment:

*z .P* ▶ *z .z* '[x'*/z.E* ]*.Q*

x:=*E* ± [x*T* ; *V* | *P* | *Q* ]

* 1. *Conditional*

We define a new operator, a conditional schema, in terms of conjunction and strengthening of preconditions:

Definition 4.6 Let *TD* ≤ *T*0 Ω *T*1.

if *D* then *U* P *T*0 else *U* P *T*1 = *U* ↑ *D* Λ *U*

0 1 *df* 0 1

↑ ч*D*

Rules for the conditional:

Proposition 4.7

. .

*t.D* ▶ *z* ∈ *U*0 ч*t.D* ▶ *t* ∈ *U*1 *(*if+*)*

*t* ∈ if *D* then *U*0 else *U*1

*t* ∈ if *D* then *U*0 else *U*1 *t.D*

0

.

*t* ∈ *U*0

*(*if—*)*

*t* ∈ if *D* then *U*0 else *U*1 *t.*(ч*D* ) *(*if—*)*

. 1

*t* ∈ *U*1

Equations and inequations:

Proposition 4.8

.

if *true* then *U*0 else *U*1 = *U*0

Proof. Follow from specialisations of the introduction rule and the first elim- ination rule:

*false* (*1*)

. .

Proposition 4.9

*z* ∈ *U*0 *z* ∈ *U*1 (*1*)

*z* ∈ if *D* then *U*0 else *U*1

*z* ∈ if *D* then *U*0 else *U*1

.

*z* ∈ *U*0

Proposition 4.10

.

if *false* then *U*0 else *U*1 = *U*1

if *D* then [*T* | *P* | *D* Λ *Q* ] else [*T* | *P* | ч*D* Λ *Q* ] ± [*T* | *P* | *Q* ]

Proof. In what follows we write *φ* for

*z* ∈ if *D* then [*T* | *P* | *D* Λ *Q* ] else [*T* | *P* | ч*D* Λ *Q* ]

*D* ∨ ч*D*

*z .P* (*1*)

*φ z .D* (*z*)

*z* ∈ [*T* | *P* | *D* Λ *Q* ]

*z .*(*D* Λ *Q* )

*z .Q*

*z .P* (*1*)

*φ z .*(ч*D* ) (*z*) *z* ∈ [*T* | *P* | ч*D* Λ *Q* ] *z .*(ч*D* Λ *Q* )

*z .Q*

*z.Q* (*z*)

*z* ∈ [*T* | *P* | *Q* ] (*1*)

* 1. *Cases*

The previous section can easily be generalised to case commands:

We define a new operator, a case schema, in terms of conjunction and strengthening of preconditions:

Definition 4.11 Let *T* = {··· *ci* ·· ·}.

cases *ET* in *c*0

: *U* P *T*0

··· *cn*

: *U* P *Tn* endcases =

*def*

*U*0 ↑ *E* = *c*0 Λ ··· Λ *Un* ↑ *E* = *cn*

0

*n*

Rules and inequations are omitted here, but are obvious generalisations of those for the conditional.

* 1. *Scope*

Definition 4.12

begin var x : *T* ;*U* end = *T*x '*T*x • *U*

x *df* E x *,* x

Proposition 4.13

*t* ∈ *U*

.

*(*begin+*)*

*t* ∈ begin var x : *T*x;*U* end

*t* ∈ begin var x : *T*x;*U* end *t \** ⟨| x$*y*0*,* x'$*y*1 |⟩ ∈ *U* ▶ *P P*

*(*begin—*)*

* 1. *Procedure call*

This and the interpretation of procedures themselves are mutually dependent. Suppose that *f* is a procedure (we will see an example in the next section), then procedure call is trivially defined:

*f* (*E* ) =*df f* [x$*E* ]

This leads to inference rules:

.

*t* ∈ *f t.*x = *t.E*

*t* ∈ *f* (*E* )

*t* ∈ *f* (*E* )

~~.~~

*t* ∈ *f*

*t* ∈ *f* (*E* )

*t.*x = *t.E*

It is necessary to analyse this in advance of procedures themselves, as it is implicated in the definition, as we will now see.

* 1. *Primitive recursive procedures over numbers*

We define a new schema operator, primitive recursion over the natural num- bers, in terms of conjunction, strengthening of preconditions, existential hid- ing, schema application and recursive schemas.

Definition 4.14

proc *f* (x) cases x in 0 : *U*0; m +1 : *U*1(*f* (m)) endcases =*df*

*µX* • *U*0 ↑ x = 0 Λ E m • *U*1(*X* [x$m]) ↑ x = m +1

The idea is that *U*1 is a schema whose alphabet includes m and which con- tains a free schema variable *X* whose type is the type of the entire procedure.

And the rules.

Proposition 4.15 *Introduction:*

. .

*t.*x = 0 ▶ *t* ∈ *U*0 *t.*x = *t.*m +1 ▶ *t* ∈ *U*1(*f* (m))

~~.~~

*t* ∈ *f*

Consider the following derivation:

Proof.

(*1*)

*t.*x = *t*. *.*m +1 (*z*)

.

*t.*x .= 0 . .

. .

*t* ∈ *U*0

.

(*1*)

*t* ∈ *U*1[*f* (m)] (*z*)

*t* ∈ *U*1[*f* (m)] ↑ x = m +1

~~.~~

.

*t* ∈ *U*0 ↑ x = 0 *t* ∈ E m • *U*1[*f* (m)] ↑ x = m +1

~~.~~

*t* ∈ *U*0 ↑ x = 0 Λ E m • *U*1[*f* (m)] ↑ x = m +1 (*µ*+)

.

*t* ∈ *f*

Proposition 4.16 *Elimination:*

.

*t* ∈ *f t.*x = 0

~~.~~

*t* ∈ *U*0

.

*t* ∈ *f t.*x = m +1

~~.~~

*t* ∈ *U*1(*f* (m))

In what follows, we write *U* [*E* ] for *U* [x$*E* ], when x is understood.

Proposition 4.17 *The following rule is derivable:*

*n* ∈ N ▶ *f* (*n*) ± *U* [*n*]

*f* ± *U*

Proof. Consider the following derivation:

(*1*)

*z .*x .∈ N

*z* ∈ *f z .*x = *z .*x

*z* ∈ *f* (*z .*x)

.

*f* (*z .*x) ± *U* [*z .*x]

*z* ∈ *U* [*z .*x]

*z* ∈ *U f* ± *U*

(*1*)

And now, the key rule for program development for recursive programming: the rule for recursive synthesis:

Proposition 4.18 *The following rule is derivable:*

*U*0 ± *U* [0] *f* (m) ± *U* [m] ▶ *U*1(*f* (m)) ± *U* [m + 1]

*f* ± *U*

Proof. Consider the following derivation:

(*z*)

*z* ∈ *f* (0)

*z .*x = 0

*U*0 ± *U* [0] *z* ∈ *U*0

*z* ∈ *U* [0]

(*z*)

*f* (0) ± *U* [0]

(*z*)

*z* ∈ *f* (0)

*z* ∈ *f*

*f* (*n*) ± *U* [*n*]

*δ*.

.

*f* (m + 1) ± *U* [m + 1]

(*1*)

*f* ± *U* (*0*)

where *δ* is:

*z* ∈ *f* (m + 1) (*3*) *z* ∈ *f* (m + 1) (*3*)

(*0*)

*f* (m) ± *U* [m]

*z.*x = m + 1 *z* ∈ *f z* ∈ *U*1(*f* (m))

.

.

*U*1(*f* (m)) ± *U* [m + 1]

*z* ∈ *U* [m + 1]

*f* (m + 1) ± *U* [m + 1] (*3*)

* 1. *Primitive recursion over lists*

The technique is easy to generalise. For example:

Definition 4.19

proc *f* (x) cases x in Nil : *U*0; Cons m0 m1 : *U*1(*f* (m1)) endcases =*df*

*µX* • *U*0 ↑ x = Nil Λ E m0*,* m1 • *U*1(*X* [x$m1]) ↑ x = Cons m0 m1

The rule for recursive synthesis over lists:

Proposition 4.20 *The following rule is derivable:*

*U*0 ± *U* [Nil] *f* (m1) ± *U* [m1] ▶ *U*1(*f* (m1)) ± *U* [Cons m0 m1]

*f* ± *U*

* 1. *Primitive recursion over trees*

Similarly for trees:

Definition 4.21

proc *f* (x) cases x in Leaf m0 : *U*0; Node m1 m2 : *U*1(*f* (m)) endcases =*df*

*µX* • E m0 • *U*0 ↑ x = Leaf *m*0 Λ E m1*,* m2 • *U*1(*X* [x$m1]*, X* [x$m2]) ↑ x = Node m1 m2

The rule for recursive synthesis over trees:

Proposition 4.22 *The following rule is derivable:*

*U*0 ± *U* [Leaf m0] *f* (m1) ± *U* [m1]*, f* (m2) ± *U* [m2] ▶ *U*1(*f* (m1*,* m2)) ± *U* [Node m1 m2]

*f* ± *U*

* 1. *Primitive recursion over arbitrary free-types*

All these special cases can be generalised to syntax-directed *free types*.

Types of the form Υ are the names of the free types and are given by equations of the form:

Υ ::= ··· | *ci* ⟨⟨··· Υ*ij* ·· ·⟩⟩ | ···

The terms of free-type:

*t* Υ ::= *ci* ··· *t* Υ*ij* ···

The logic of free types permits the introduction of values in the type, equality reasoning and finally, elimination (generally by induction).

Proposition 4.23

··· *zij* ∈ Υ*ij* ··· *(*Υ+*)* ··· *zij* ∈ Υ*ij* ··· ··· *zkl* ∈ Υ*kl* ··· *(*Υ *)*

*ci* ··· *zij* ··· ∈ Υ

*ci* ··· *zij* ··· =/

*ck* ··· *zkl* ··· /=

*ci* ··· *zij* ··· = *ci* ··· *yij* ···

*zij* = *yij (*Υ=*)*

··· ··· *zij* ∈ Υ*ij* ··· *,* ··· *P* [*z/yk* ] ··· ▶ *P* [*z/ci* ··· *zij* ·· ·] ··· *(*Υ—*)*

*z* ∈ Υ ▶ *P*

*where the yk are all those variables occurring in the zij with type* Υ*.*

Given a general free type Υ, the corresponding recursive program scheme

is:

Definition 4.24

procΥ *f* (*x* ) cases x in ··· *Hi* ··· endcases

where the *Hi* are the component cases:

*Hi* =*df ci* ··· m*i* ··· : *Ui* (··· *f* (w*k* )*,* ·· ·) where the w*k* are those observations among the m*i* with type Υ.

The semantics in the general case is given by:

Definition 4.25

procΥ *f* (*x* ) cases x in ··· *Hi* ··· endcases =*df µX* • ··· Λ *Ki* (*X* ) Λ ···

where:

*Ki* (*X* ) =*df* E ··· m*i* ··· • *Ui* (··· *X* [x$w*k* ] ·· ·) ↑ x = *ci* ··· m*i* ···

# Conclusions and further work

As we mentioned in the introduction, this expository paper concentrates en- tirely on the theoretical basis of *ν*Z. We have showed how an extremely simple logic can be extended towards an expressive specification logic and a program (development) logic. One of the benefits of this approach is its flexibility: one is not constrained by any particular specification or programming language infrastructure. The ability to provide elegant rules for total correctness de- velopment of procedures is also a strength: these rules resemble those which proved so useful in program development within constructive theories (see, for example, [[5](#_bookmark6)]) but are here combined with the ability to synthesize imperative programs.

Much infrastructural and pragmatic work remains to be done, both at the level of specification and program development. At the pragmatic level in particular, much work is being undertaken by Kajtazi and this will be reported in his PhD thesis.

# Acknowledgements

The authors are grateful to Steve Reeves, Lindsay Groves and especially Moshe Deutsch for numerous discussions and critical appraisal of this work. We would like to acknowledge the support of the EPSRC RefineNet Network Grant (Ref: GR/S69979/02) in the development of this work, including comments from

participants at the January 2004 meeting in Sheffield where an earlier version of this material was presented.

# References

1. R. J. R. Back and J. von Wright. *Reﬁnement Calculus: A Systematic Introduction*. Springer, 1998.
2. W. P. de Roever and K. Engelhardt. *Data Reﬁnement: Model-Oriented Proof Methods and Their Comparison*. Prentice Hall International, 1998.
3. M. Deutsch, M. C. Henson, and S. Reeves. An analysis of total correctness refinement models for partial relation semantics I. *Logic Journal of the IGPL*, 11(3):287–317, 2003.
4. M. C. Henson and S. Reeves. Investigating Z. *Logic and Computation*, 10(1):43–73, 2000.
5. P. Martin-L¨of. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science VI*, pages 153–175. North Holland, 1982.
6. C. C. Morgan. *Programming from Speciﬁcations*. Prentice Hall International, 2nd edition, 1994.