Working Paper Series
ISSN 1170-487X

New Foundations for Z

by Martin C Henson and Steve Reeves

Working Paper 98/6
March 1998

Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, New Zealand

New Foundations for Z

Martin C. Henson and Steve Reeves

Department of Computer Science,
University of Essex, England
hensm@essex.ac.uk
fax: +44 1206 872788
Department of Computer Science,
University of Waikato, New Zealand
stever@cs.waikato.ac.nz
fax: +64 7 8384155

Abstract. We provide a constructive and intensional interpretation for the specification language Z in a theory of operations and kinds $\mathcal{T}$. The motivation is to facilitate the development of an integrated approach to program construction. We illustrate the new foundations for Z with examples.

1 Introduction

The standard account of the foundations of the specification language Z is given in *classical, extensional* set theory. In this paper we explore the consequences of an alternative, based on a *constructive, intensional* set theory. We are not suggesting that the standard model is in any way intrinsically inadequate; rather that there may be an alternative with a different set of practical consquences for specification and, in particular, for *program development.* There is, of course, also mathematical interest involved in exploring this different choice of foundation.

2 Foundations of Z: some suggestions

2.1 Extensional or intensional sets?

The standard interpretation of functions in Z is the extensional representation in classical set theory. An advantage is that this is an *abstract* approach, although what this amounts to requires a little exploration. It is certainly abstract with respect to *algorithm* (algorithmic detail is suppressed), though it is not abstract with respect to *representation* (a function is explicitly a set of ordered pairs). Let us consider these two issues in more detail.

Firstly, being abstract with respect to algorithm has the consequence that in many cases an operation schema will determine a *singleton* set of functions. For example:

Square

$x, x' \in \mathbb{R}^+$

$x' = x^2$

determines a unique function. In fact, the standard model of this schema is a set of *bindings* (because Z does not distinguish, formally, between *state* and *operation* schema) and it is this set which is in bijective correspondence with the unique function to which we are referring. Most crucially an element t of this schema is no more than a single *observation* of this function ($t.x' = (t.x)^2$) and, even if we could interpret an operation schema (*qua* set) in terms of the *function* determined by its bindings, there are no algorithmic distinctions to be made explicit: as a set in this sense, the schema *Square* is a singleton. Formally, of course, this presents no problems, but it does imply that programming (and reasoning at that level) has to be undertaken *outside* the standard interpretation of Z. The most mature example of this indirect strategy is certainly that of [7] (Chapter 17 in particular). Consider, for example their internal model of operation schema (*ibid.* p. 257).

On the other hand, the assertion $f \in Square$ might usefully represent a relationship between an implementation f and a specification *Square*. This would be possible if the representation (for what correspond, ultimately, to programs) were more intensional (*i.e.* distinguished different *ways* that that set-theoretic objects might be constructed) and if, furthermore, there were a formal distinction between state and operation schema. Under this regime the interpretation of *Square* would constitute a large class of *abstract operations*, and the framework would be able to incorporate, in a natural way, reasoning about varieties of implementations of a given specification. In particular, procedural refinement and program transformations would be directly expressible.

Secondly, the fact that the standard model is not abstract with respect to representation has the consequence that Z specifications often involve explicit calculations in the syntax of the underlying set theoretic framework. For example, in order to extend a partial function f at an element x one must write: $f \cup \{x \mapsto y\}$ which is an explicit (non-abstract) calculation in a particular *concrete representation* of functions in a particular target theory. Arguably, operations upon functions should be described (axiomatised) directly in the theory of the *object* language.

2.2 Classical or constructive logic?

With sufficient patience it would be possible to represent, in Z, the essentials of elementary recursion theory, including a specification H of the *halting function.* Since existence in Z is inherited from existence in classical, extensional set theory, the specification H determines a (single) function. If we wish to intepret the assertion $f \in H$ as a relationship between an implementation and a specification this obviously presents problems. Classical existence is very strong, and there are many more functions than algorithms. Weakening the set theory so that it

admitted only constructive existence would prevent a proof that H is inhabited, and it would have another well-known benefit: a proof that a specification is inhabited contains, implicitly, a *program which meets the specification.* There is, therefore, the prospect that reasoning in the logic for Z would be intimately linked with program development. In addition, because we are also arguing for an intensional theory, the precise details of a proof of inhabitation are linked with the *algorithmic* properties of the implicit program. We have, then, the prospect of a logic for program development which offers control over those crucial algorithmic aspects of program construction.

2.3 Implicit or explicit definedness?

Perhaps the most troublesome aspects of Z, in practice, arise from the potential undefinedness of terms. In fact, much has been written on this topic which is entirely misleading: terms of the form $f(x)$ arise in set theory syncategorematically and are always eliminable. Consequently, for example, an expression such as $f(x) = y$, when f is a partial function, is in no danger whatsoever of being *undefined* because the interpretation of this equality does not depend upon the interpretation of the two constituent terms. The proposition is a meta-language *convention* and is understood to represent the proposition $x \mapsto y \in f$ which is either true or false[1]. All occurrences of application meta-terms are then easily removed by the meta-linguistic convention that a proposition $P[z/f(x)]$ is understood to represent: $(\exists z)(f(x) = z \wedge P)$ in which the remaining offending occurrence of $f(x)$ can be removed as explained above.

Nevertheless there is a genuine *practical* problem associated with the use of partial functions *etc.* in Z. In the constructive and intensional version of Z that we introduce, there is an explicit form of proposition which establishes whether or not a term is defined. Note that, since our functions are intensional (operations), there is no prospect of treating potentially non-denoting terms syncategorematically as described above.

3 The specification logic Z_C

In this section we shall describe a simple specification logic which we call Z_C. This is a typed set-theory based upon the notion of *schema type.* Our formulation in this paper will be "Church-style", in which types are carried explicitly by the variables (and hereditarily by the terms). This constrasts with our earlier presentations (*e.g.* [3]) in which Z_C is presented "Curry-style", with the types assigned by type assignment rules).

[1] It would be interesting to consider how people came to be misled, especially considering the fact that in Z, much is made of the fact that functions are sets. Almost all the text books, for example, take pleasure in notational tricks which are at once obscure and unintuitive *e.g.* composing a (numeric) function with a *sequence.* Given that frame of mind, it is remarkable that confusions over definedness of function applications ever arose.

Once we have established Z_C it will be possible to construct the more complex apparatus (including the schema calculus) of Z within this system. Therefore, if we can provide Z_C with an intensional and constructive interpretation, this will be inherited by the, more ambitious, derived features.

Our commitment to abstract operations rather than to extensional functions implies that, in our reformulation, functions are *not* sets of ordered pairs, and so, function spaces are not definable in terms of power set and cartesian product. As a result, the (partial) functions enter Z_C (and therefore Z) as a basic *type* constructor.

3.1 The language of Z_C

Z_C is a *constructive, typed, partial set theory.* We begin with the types:

$$T ::= \mathbb{N} \mid \mathbb{P}\, T \mid T \times T \mid T \pfun T \mid [\cdots l_i : T_i \cdots]$$

Typing of terms must be *decidable.* As a consequence the function space type constructor must be *partial.*

We will often permit the meta-variable D^T to range over sequences of type assignments such as $\cdots l_i : T_i \cdots$, also writing $\alpha[D^T]$, when D^T is $\cdots l_i : T_i \cdots$, for the alphabet set (in the meta-language) of labels $\{\cdots l_i \cdots\}$. We shall need to indicate multiple subsitutions for all labels in a given alphabet, writing $[\alpha[D^T]/t(\alpha[D^T])]$ to indicate the family of subsitutions $\cdots [l_i/t(l_i)] \cdots$ when t is some term and $\alpha[D^T] = \{\cdots l_i \cdots\}$.

Types of the form $[D^T]$ are called *schema types.* Other operations on schema types that we will need: $[D_0^T] \sqsubseteq [D_1^T]$ holds when the set of type assignments in $[D_0^T]$ is a subset of those of $[D_1^T]$; $[D_0^T] \sqcup [D_1^T]$ is the schema type comprising all the type assignments appearing in the components. It is not defined if this union contains type assignments of the form $l : T_0$ and $l : T_1$ with $T_0 \neq T_1$. $[D_0^T] \sqcap [D_1^T]$ is the schema type comprising all type assignments occurring in both $[D_0^T]$ and $[D_1^T]$. Finally, $[D_0^T] - [D_1^T]$ is the schema type comprising all type assignments in $[D_0^T]$ which are not in $[D_1^T]$.

All categories of the language of Z_C must be well-formed with respect to these types.

First we have the category of sets:

$$\begin{array}{ll}
C^{\mathbb{P}\mathbb{N}} & ::= \mathbb{N}^* \\
C^{\mathbb{P}\, T} & ::= \mathbb{P}\, C^T \\
C^{\mathbb{P}(T_0 \times T_1)} & ::= C^{\mathbb{P}\, T_0} \times C^{\mathbb{P}\, T_1} \\
C^{\mathbb{P}(T_0 \pfun T_1)} & ::= C^{\mathbb{P}\, T_0} \pfun C^{\mathbb{P}\, T_1} \\
C^{\mathbb{P}([\cdots l_i : T_i \cdots])} & ::= [\cdots l_i \in C_i^{\mathbb{P}\, T_i} \cdots] \\
C^T & ::= \{z \in C^T \mid P\}
\end{array}$$

Sets, then, are formed from the natural numbers by powerset, cartesian product, partial function space, schema sets and separation (bounded comprehension). Among the sets are the *carriers* of the types. These are formed by closing

the carrier for the basic type $\mathbb{N}^*$ under the four set forming operations with corresponding operations in the type language. In the sequel we will often write T as a set (the carrier of the type T). In this regard we are following the notational abuse described in [5] (p. 24)).

The formulæ of Z_C delineate a constructive type-bounded partial predicate logic.

$$P ::= \bot \mid t{\Downarrow} \mid t_0^T = t_1^T \mid t^T \in C^{\mathbb{P}\,T} \mid \neg P \mid P_0 \vee P_1 \mid$$
$$P_0 \wedge P_1 \mid P_0 \Rightarrow P_1 \mid \forall z^T \in C^{\mathbb{P}\,T} \bullet P \mid \exists z^T \in C^{\mathbb{P}\,T} \bullet P$$

We shall need a weaker notion of membership in order to present the rules for membership in the carrier of partial functions. In addition we define a similarly weaker notion of equality.

Definition 1.

$$(i)\ \ t^T \in_w C^{\mathbb{P}\,T} =_{df} t^T{\downarrow} \Rightarrow t^T \in C^{\mathbb{P}\,T}$$
$$(ii)\ t_0^{T_0} =_w t_1^{T_1} =_{df} (t_0^{T_0} \vee t_1^{T_1}) \Rightarrow t_0^{T_0} = t_1^{T_1}$$

We assume the existence of a denumerable set of variables for each type T. The syntax of terms is then:

$$\begin{array}{lll}
t^T & ::= & x^T \mid C^T \mid t^{[\cdots l:T\cdots]}.l \mid t^{T'} \upharpoonright T \mid t^{T\times T_1}.1 \mid t^{T_0\times T}.2 \mid \\
 & & t^{T_0 \nrightarrow T}\, t^{T_0} \mid \mathbf{e}^T \\
t^{\mathbb{N}} & ::= & n \\
t^{[\cdots l_i:T_i\cdots]} & ::= & (\!| \cdots l_i \Rrightarrow t_i^{T_i} \cdots |\!) \\
t^{T_0\times T_1} & ::= & (t_0^{T_0}, t_1^{T_1}) \\
t^{T_0 \nrightarrow T_1} & ::= & \lambda\, x^{T_0}.t^{T_1} \\
t^{T_0 \sqcup T_1} & ::= & t_0^{T_0} \wedge t_1^{T_1}
\end{array}$$

where $T \sqsubseteq T'$. The constants $\mathbf{e}^T$ (there is one at every type) is an *exceptional* value. This, and the binding conjunction operator, are used in the interpretation of the operation schema calculus. The lambda terms in this grammar will denote genuine abstract operations in the model; this should be contrasted with the use of lambda expressions in standard Z that denote sets of ordered pairs. We pronounce the symbol $\upharpoonright$ "filter" and the purpose of filtered terms is to permit the restriction of bindings to a given schema type. These are crucial for establishing a logic for the schema calculus.

Numerals are as expected:

$$n ::= 0 \mid succ\ n$$

3.2 The logic of Z_C

The judgements of the logic have the form $\Gamma \vdash_C P$ where Γ is a set of formulæ.

We shall omit all data (entailment symbol, contexts, type *etc.*) which remains unchanged by a rule. The usual side-conditions apply.

$$\frac{}{x\!\downarrow}\quad\frac{}{C\!\downarrow}\quad\frac{}{n\!\downarrow}\qquad\frac{t.1\!\downarrow}{t\!\downarrow}\qquad\frac{t.2\!\downarrow}{t\!\downarrow}\qquad\frac{(t_0,t_1)\!\downarrow}{t_0\!\downarrow}\qquad\frac{(t_0,t_1)\!\downarrow}{t_1\!\downarrow}\qquad\frac{t_0\!\downarrow\quad t_1\!\downarrow}{(t_0,t_1)\!\downarrow}\qquad\frac{t.l\!\downarrow}{t\!\downarrow}$$

$$\frac{\cdots\ t_i\!\downarrow\ \cdots}{\langle\!|\cdots l_i \Rrightarrow t_i\cdots|\!\rangle\!\downarrow}\qquad\frac{\langle\!|\cdots l_i \Rrightarrow t_i\cdots|\!\rangle\!\downarrow}{t_i\!\downarrow}\qquad\frac{}{\lambda\, x.t\!\downarrow}\qquad\frac{t\!\downarrow}{(t\restriction T)\!\downarrow}\qquad\frac{(t\restriction T)\!\downarrow}{t\!\downarrow}$$

$$\frac{(t_0\ t_1)\!\downarrow}{t_0\!\downarrow}\qquad\frac{(t_0\ t_1)\!\downarrow}{t_1\!\downarrow}\qquad\frac{t\in C}{t\!\downarrow}\qquad\frac{}{\mathbf{e}\!\downarrow}\qquad\frac{(t_0\wedge t_1)\!\downarrow}{t_0\!\downarrow}\qquad\frac{(t_0\wedge t_1)\!\downarrow}{t_1\!\downarrow}\qquad\frac{t_0\!\downarrow\quad t_1\!\downarrow}{(t_0\wedge t_1)\!\downarrow}$$

$$\frac{P_0}{P_0\vee P_1}\ (\vee_0^+)\qquad\frac{P_1}{P_0\vee P_1}\ (\vee_1^+)\qquad\frac{P_0\vee P_1\quad P_0\vdash P_2\quad P_1\vdash P_2}{P_2}\ (\vee^-)$$

$$\frac{P_0\quad P_1}{P_0\wedge P_1}\ (\wedge^+)\qquad\frac{P_0\wedge P_1}{P_0}\ (\wedge_0^-)\qquad\frac{P_0\quad P_1}{P_1}\ (\wedge_1^-)$$

$$\frac{P_0\vdash P_1}{P_0\Rightarrow P_1}\ (\Rightarrow^+)\qquad\frac{P_0\Rightarrow P_1\quad P_0}{P_1}\ (\Rightarrow^-)$$

$$\frac{P\vdash\ \bot}{\neg P}\ (\neg^+)\qquad\frac{P\quad\neg P}{\bot}\ (\bot^+)/(\neg^-)\qquad\frac{\bot}{P}\ (\bot^-)$$

$$\frac{P[z/t]\quad t\in C}{\exists z\in C\bullet P}\ (\exists^+)\qquad\frac{\exists z\in C\bullet P_0\quad P_0[z/y]\vdash P_1}{P_1}\ (\exists^-)\qquad\frac{x\in C\vdash P}{\forall x\in C\bullet P}\ (\forall^+)$$

$$\frac{\forall x\in P\quad t\in C}{P[x/t]}\ (\forall^-)\qquad\frac{}{\Gamma,P\vdash P}\ (ass)\qquad\frac{t\!\downarrow}{t=t}\ (ref)$$

$$\frac{\cdots\quad t_i\!\downarrow\quad\cdots}{\langle\!|\cdots l_i \Rrightarrow t_i\cdots|\!\rangle.l_i=t_i}\ (\Rightarrow_0^=)\qquad\frac{t\!\downarrow}{\langle\!|\cdots l_i \Rrightarrow t.l_i\cdots|\!\rangle=t}\ (\Rightarrow_1^=)$$

$$\frac{t\!\downarrow\quad t'\!\downarrow}{(t,t').1=t}\ ((,)_0^=)\qquad\frac{t\!\downarrow\quad t'\!\downarrow}{(t,t').2=t'}\ ((,)_1^=)\qquad\frac{t\!\downarrow}{(t.1,t.2)=t}\ ((,)_2^=)$$

$$\frac{t_0^{T_0}.l=t\quad l\in\alpha T_0}{(t_0\wedge t_1).l=t}\ (\wedge_0^=)\qquad\frac{t_1^{T_1}.l=t\quad l\in\alpha T_1-\alpha T_0}{(t_1\wedge t_1).l=t}\ (\wedge_1^=)$$

$$\frac{t'=t}{t=t'}\ (sym)\qquad\frac{t=t'\quad P[z/t]}{P[z/t']}\ (sub)$$

$$\frac{P[z/t]\quad t\in C}{t\in\{z\in C\mid P\}}\ (\{\}^+)\qquad\frac{t\in\{z\in C\mid P\}}{t\in C}\ (\{\}_0^-)\qquad\frac{t\in\{z\in C\mid P\}}{P[z/t]}\ (\{\}_1^-)$$

$$\frac{z\in C_0\vdash z\in C_1}{C_0\in\mathbb{P}\,C_1}\ (\mathbb{P}^+)\qquad\frac{C_0\in\mathbb{P}\,C_1\quad t\in C_0}{t\in C_1}\ (\mathbb{P}^-)$$

$$\frac{t_0\in C_0\quad t_1\in C_1}{(t_0,t_1)\in C_0\times C_1}\ (\times^+)\qquad\frac{t\in C_0\times C_1}{t.1\in C_0}\ (\times_0^-)\qquad\frac{t\in C_0\times C_1}{t.2\in C_1}\ (\times_1^-)$$

$$\frac{}{0\in\mathbb{N}}\ (\mathbb{N}_0^+)\qquad\frac{z\in\mathbb{N}}{succ\ z\in\mathbb{N}}\ (\mathbb{N}_1^+)\qquad\frac{P[z/0]\quad z\in\mathbb{N};\ P\vdash P[z/succ\ z]}{z\in\mathbb{N}\vdash P}\ (\mathbb{N}^-)$$

$$\frac{\cdots\quad t_i\in C_i\quad\cdots}{\langle\!|\cdots l_i \Rrightarrow t_i\cdots|\!\rangle\in[\cdots l_i\in C_i\cdots]}\ ([]^+)\qquad\frac{t\in[\cdots l_i\in C_i\cdots]}{t.l_i\in C_i}\ ([]^-)$$

$$\frac{x\in C_0\vdash t\in_w C_1}{\lambda\, x.t\in C_0\twoheadrightarrow C_1}\ (\lambda^+)\qquad\frac{t\in C_0\twoheadrightarrow C_1\quad t'\in C_0}{t\ t'\in_w C_1}\ (\lambda^-)$$

$$\frac{t.l_i=t_i\quad[\cdots l_i:T_i\cdots]\sqsubseteq T'}{(t^{T'}\restriction[\cdots l_i:T_i\cdots]).l_i=t_i}\ (\restriction^=)$$

The transitivity of equality and numerous *equality congruence* rules for the various term forming operations are all derivable in view of rule (*sub*).

3.3 State schema calculus

The specification logic Z_C is essentially a typed set-theory in which we have, in particular, schema types. There are, however, no *schema* in Z_C and this may seem rather odd since these are archetypical of Z. In fact, given the schema types, schema are just special cases of the comprehensions. Specifically, we may introduce schema by metanotational convention using the following definition:

$$[\, D \mid P \,] =_{df} \{z \in [D] \mid P[\alpha[D]/z.\alpha[D]\}$$

In fact we shall make the further proviso that the predicate part of a schema be *decidable*. Z is most usually used to specify software or hardware and, consequently, this proviso *must* be operational at some point, even if it is left implicit. We highlight the point explicitly because there are logical implications which will surface later in the paper.

Note that this device requires us to allow the meta-variable P to range over the propositions *extended with labels as terms*. The definiendum is, of course, syntactically valid in Z_C. Given this definition we may provide the following versions of the comprehension rules using the schema notation:

$$\frac{t \in [D] \quad P[\alpha[D]/t.\alpha[D]]}{t \in [D \mid P]}\ (S^{+}) \qquad \frac{t \in [D \mid P]}{t \in [D]}\ (S^{-}_{\circ}) \qquad \frac{t \in [D \mid P]}{P[\alpha[D]/t.\alpha[D]]}\ (S^{-}_{1})$$

We allow the obvious generalisation of our alphabet operator to schema: $\alpha[D \mid P] = \alpha[D]$.

We can now introduce state schema expressions, including propositional connectives, hiding and renaming.

Definition 2.

$$\begin{array}{lll}
(i) & \neg S^{\mathbb{P}\, T} & =_{df} \{z \in T \mid z \notin S\} \\
(ii) & S_0^{\mathbb{P}\, T_0} \vee S_1^{\mathbb{P}\, T_1} & =_{df} \{z \in T_0 \sqcup T_1 \mid z \upharpoonright T_0 \in S_0 \vee z \upharpoonright T_1 \in S_1\} \\
(iii) & S_0^{\mathbb{P}\, T_0} \wedge S_1^{\mathbb{P}\, T_1} & =_{df} \{z \in T_0 \sqcup T_1 \mid z \upharpoonright T_0 \in S_0 \wedge z \upharpoonright T_1 \in S_1\} \\
(iv) & S_0^{\mathbb{P}\, T_0} \Rightarrow S_1^{\mathbb{P}\, T_1} & =_{df} \{z \in T_0 \nrightarrow T_1 \mid \forall x \in S_0 \bullet z\ x \in S_1\} \\
(v) & S^{\mathbb{P}\, T}/\{l : T'\} & =_{df} \{z \in T^{-l} \mid \exists x \in T \bullet x \in S \wedge z = x \upharpoonright T^{-l}\} \\
(vi) & S^{\mathbb{P}\, T}[l \leftarrow l'] & =_{df} \{z \in T[l \leftarrow l'] \mid \exists x \in S \bullet z = x[l \leftarrow l']\}
\end{array}$$

where $T^{-l} =_{df} T - [l : T']$.

We cannot present all the rules, which follow from these definitions, in this paper. We can illustrate the situation with the rules for conjunction.

$$\frac{t \upharpoonright T_0 \in S_0^{\mathbb{P}\, T_0} \quad t \upharpoonright T_1 \in S_1^{\mathbb{P}\, T_1}}{t \in S_0 \wedge S_1}\ (S^{+}_{\wedge}) \qquad \frac{t \in S_0^{\mathbb{P}\, T} \wedge S_1}{\Gamma \vdash t \upharpoonright T \in S}\ (S^{-}_{\wedge_\circ}) \qquad \frac{t \in S_0 \wedge S_1^{\mathbb{P}\, T}}{t \upharpoonright T \in S_1}\ (S^{-}_{\wedge_1})$$

$$\frac{S_0 = S_1}{S_0 \wedge S_2 = S_1 \wedge S_2} \qquad \frac{S_0 = S_1}{S_2 \wedge S_0 = S_2 \wedge S_1}$$

From these rules it is possible to prove the characteristic equation for conjunction, an equation which is often used, informally, to define schema conjunction in the literature:

$$\frac{}{[D_0 \mid P_0] \wedge [D_1 \mid P_1] = [[D_0] \sqcup [D_1] \mid P_0 \wedge P_1]} \ (\wedge^{=})$$

There is far more detail in [3]; at least in the setting of classical logic and extensional set-theory.

3.4 Operation schema

Our discussion in section 1 involved a suggestion that there could usefully be an explicit distinction between state and operation schema. This is established in Z_C by introducing operation schema separately, and translating them appropriately. Schema have the following form:

$$[\Delta S;\ \mathbf{in}\ D_i;\ \mathbf{out}\ D_o \mid \mathbf{pre}\ P;\ \mathbf{post}\ Q]$$

where the D_i introduce input labels, and D_o output labels. The precondition P may mention the labels of S and D_i only[2].

Let $S_i =_{df} S \wedge [D_i]$ and $S_o =_{df} S \wedge [D_o]$. Then operation schemas are translated by:

$$\{f \in S_i \nrightarrow S_o \mid \forall z \in S_i \bullet P[\alpha S_i / z.\alpha S_i] \Rightarrow Q[\alpha S_i / z.\alpha S_i, \alpha S' / f(z).\alpha S, \alpha[D_o] / f(z).\alpha[D_o]]\}$$

It is clear, then, that an operation schema specifies the set of partial functions which satisfy its predicate[3].

The derived rules for operation schema are, again, specialisations of those for comprehensions:

$$\frac{t \in [\Delta S;\ \mathbf{in}\ D_i;\ \mathbf{out}\ D_o \mid \mathbf{pre}\ P\ \mathbf{post}\ Q]}{t \in S_i \nrightarrow S_o}$$

$$\frac{t \in [\Delta S;\ \mathbf{in}\ D_i;\ \mathbf{out}\ D_o \mid \mathbf{pre}\ P\ \mathbf{post}\ Q] \quad t' \in S_i \quad P[\alpha S_i / t'.\alpha S_i]}{Q[\alpha S_i / t'.\alpha S_i, \alpha S' / t(t').\alpha S, \alpha[D_o] / t(t').\alpha[D_o]]}$$

[2] We depart from traditional Z in this paper by making the inputs, outputs, preconditions and postconditions explicit. These notational innovations are not strictly necessary, but in the context of this summary paper they *significantly* simplify the technical presentation.

[3] One may object that, in *standard* Z, operation schema generally determine *relations* (see *e.g.* [7] p. 257). There is, however, a distinction to be drawn between the meaning of a specification and the set of implementations of that specification. In formal methods more generally there as been a long-standing discussion regarding non-determinism: should this be a potential feature of specifications *alone*, or can it extend to the programming language in which implementations themselves are realised? The answer is, of course, moot. We have chosen one possible answer to this question here.

$$\frac{z \in S_i \vdash t \in_w S_o \quad z \in S_i, P[\alpha S_i/z.\alpha S_i] \vdash Q[\alpha S_i/z.\alpha S_i, \alpha S'/t.\alpha S, \alpha[D_o]/t.\alpha[D_o]]}{\lambda z.t \in [\Delta S;\ \mathbf{in}\ D_i;\ \mathbf{out}\ D_o \mid \mathbf{pre}\ P\ \mathbf{post}\ Q]}$$

We associate, with the definition of operation schema, the notion of a *program construction obligation* (or, simply, PCO) which is derived from the predicate part of the operation schema:

$$\forall z \in S_i \bullet \exists y \in S_o \bullet P[\alpha S_i/z.\alpha S_i] \Rightarrow Q[[\alpha S_i/z.\alpha S_i, \alpha S'/y.\alpha S, \alpha[D_o]/y.\alpha[D_o]]$$

The idea is simple and effective: we are obliged to prove the PCO of an operation schema in the logic. Because the theory is constructive, such a proof contains implicitly a program which is an element of that operation schema. Because the theory is intensional, each such distinct proof (in a sense which can be made precise) corresponds to a distinct program meeting the specification. Hence, a proof of the PCO for an operation schema S determines a proof of the judgement $f \in S$ for some f. Development of programs is fully integrated into the theory.

Note that a PCO has the form: $\forall z \in C_0 \bullet \exists y \in C_1 \bullet P$ where $P, z \in C_0$ and $y \in C_1$ are decidable predicates. This will become important at several points in the sequel. We shall continue to develop the schema calculus in section 7 below.

4 The theory of types and operations $\mathcal{T}$

We introduce a constructive theory of intensional types and operations called $\mathcal{T}$. This theory is a generalisation of Beeson's $\mathcal{EON}$ [1] with comprehensions and a hierarchy of predicative kinds. It is based on the logic of partial terms which is able, via an atomic judgement $t\!\downarrow$ (asserting that the term t is defined), to handle the problems of definedness which occur in Z. $\mathcal{T}$ will form the alternative foundation for Z_C. We have chosen this theory for a number of reasons. Firstly, it is very weak; in particular, the powerset operator is predicative: specification in Z does not require impredicativity and it makes good sense, mathematically, not to require inordinately strong foundations. Secondly, we have the opportunity to design the relationship between *proofs and programs* by constructing a suitable term-assignment version of $\mathcal{T}$. These issues will become clearer as we proceed with the technical development.

4.1 The language of $\mathcal{T}$

The terms are those of combinatory algebra together with constants for the formation of natural numbers and constants for the eventual interpretation of labels and the exceptional value **e**. In addition we have kinds and (some) decidable propositions as terms.

$$e \rightarrow x \mid 0 \mid \mathit{succ} \mid c \mid S \mid K \mid e\ e \mid \kappa^{(n)} \mid \varphi_D$$

Lambda abstraction is introduced *à la* Curry. With this in place we can easily represent pairing and selection $(\mathit{pair}, (\ \_\ )_0, (\ \_\ )_1)$, conditionals (*if then* (*else* or *elsif*)) and boolean constants *true* and *false*. $\mathcal{T}$ is, like Z_C, based on *strong*

membership and equality. We shall require the two weaker notions as we did in Z_C; these are defined in the same way.

The basic types (kinds of level zero) are type variables, natural numbers and comprehensions. Then at higher levels we include the kinds of the previous level:

$$\begin{array}{lll} \kappa^{(0)} & ::= & x^{(0)} \mid \mathbb{N} \mid \{x \mid \varphi^{(0)}(x)\} \\ \kappa^{(n+1)} & ::= & x^{(n+1)} \mid \kappa^{(n)} \mid \{x \mid \varphi^{(n+1)}(x)\} \end{array}$$

The prime and well-formed formulæ are:

$$\begin{array}{lll} \alpha & ::= & e \in \kappa \mid e = e \mid e{\downarrow} \mid \bot \\ \varphi & ::= & \alpha \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \Rightarrow \varphi \mid (\forall x)\varphi \mid (\forall x^{(n)})\varphi \mid \\ & & (\exists x)\varphi \mid (\exists x^{(n)})\varphi \end{array}$$

Negation is defined, as usual by: $\neg\varphi = \varphi \Rightarrow \bot$. We write $\varphi^{(n)}$ for formulæ containing no bound variable with index greater or equal to n (for this purpose we consider an unsuperscripted term variable as possessing a negative index). Occasionally we will write $\varphi^{(n)}$ to *assert* that the proposition φ has kind level n (*etc.*).

Note that the comprehensions are formed over term variables so that the hierarchy of kinds form a classification system over *terms* (and not kinds of the previous level). Consequently, we can introduce the following useful notation:

$$\{x^{(n)} \mid \varphi(x^{(n)})\} =_{df} \{x \mid (\exists x^{(n)})(x = x^{(n)} \wedge \varphi[x^{(n)}/x])\}$$

4.2 The logic of $\mathcal{T}$

The *judgements* of the theory are sequents of the form:

$$\Gamma \vdash_{\mathcal{T}} \varphi$$

with Γ a *set* of assumptions $\cdots \varphi_i \cdots$. We will drop the subscript on the entailment symbol whenever possible.

At the heart of $\mathcal{T}$ is the standard system $\mathcal{EON}$. We can, consequently, direct the reader to, for example, [1] (p. ???) for the logic of $\mathcal{EON}$, giving explicitly only those extra rules we require. This amounts to rules for set comprehension, quantification over kinds and (decidable) propositions as terms. As usual, we suppress references to the context, and any other data which is not changed by rule. Side conditions are as usual.

$$\frac{\varphi_D = \mathit{true}}{\varphi_D} \qquad \frac{\varphi_D}{\varphi_D = \mathit{true}} \qquad \frac{\varphi_D = \mathit{false}}{\neg\varphi_D} \qquad \frac{\neg\varphi_D}{\varphi_D = \mathit{false}}$$

$$\frac{\varphi(e)}{e \in \{x \mid \varphi(x)\}} \qquad \frac{e \in \{x \mid \varphi(x)\}}{\varphi(e)} \qquad \frac{\varphi(x^{(n)})}{(\forall x^{(n)})\varphi(x^{(n)})}$$

$$\frac{(\forall x^{(n)})\varphi(x^{(n)})}{\varphi(x^{(n)})} \qquad \frac{\varphi(\kappa^{(n)})}{(\exists x^{(n)})\varphi(x^{(n)})} \qquad \frac{\Gamma \vdash (\exists x^{(n)})\varphi(x^{(n)}) \quad \Gamma, \varphi(y^{(n)}) \vdash \eta}{\Gamma \vdash \eta}$$

The formulæ φ_D range over some set of decidable propositions (we assume this contains at least equality for constants and propositional logic).

The definition, in $\mathcal{T}$, of the powerset, for example, is:

$$\mathbb{P}^{(n)}\,\kappa^{(n)} =_{df} \{x \mid (\exists\, x^{(n)})(x^{(n)} = x \wedge (\forall\, y)(y \in x^{(n)} \Rightarrow y \in \kappa^{(n)}))\}$$

That is, using the convention: $\mathbb{P}^{(n)}\,\kappa^{(n)} = \{x^{(n)} \mid (\forall\, y)(y \in x^{(n)} \Rightarrow y \in \kappa^{(n)})\}$
We may then also omit the superscript, writing:
$\mathbb{P}\,\kappa = \{x \mid (\forall\, y)(y \in x \Rightarrow y \in \kappa)\}$ or, simply: $\mathbb{P}\,\kappa = \{x \mid x \subseteq \kappa)\}$ to represent the *family* of definitions.

5 Translating Z_C into $\mathcal{T}$

In this section we shall provide an interpretation of Z_C within our constructive and intensional theory of operations and types $\mathcal{T}$. The key is the interpretation of the type system of Z_C. First we need some infrastructure in $\mathcal{T}$. We have, where possible, used similar notation in the theory $\mathcal{T}$ for concepts in Z_C. This makes it easier to understand calculations in $\mathcal{T}$ and the potential ambiguity is always resolved by the context.

5.1 Sets and types

We begin by defining a number of useful families of operations in $\mathcal{T}$ (indexed by kinds) which can be used to give compositional interpretations of the notions of Z_C.

$$\begin{array}{lll}
(i) & \kappa_0 \twoheadrightarrow \kappa_1 & =_{df} \{f \mid (\forall\, x \in \kappa_0)((f\ x) \in_w \kappa_1)\} \\
(ii) & \Pi(x \in \kappa_0)\kappa_1(x) & =_{df} \{f \mid (\forall\, x \in \kappa_0)((f\ x) \in \kappa_1(x))\} \\
(iii) & \kappa_0 \times \kappa_1 & =_{df} \{z \mid (z)_0 \in \kappa_0 \wedge (z)_1 \in \kappa_1\} \\
(iv) & \mathbb{P}\,\kappa & =_{df} \{x \mid x \subseteq \kappa\} \\
(v) & \{\cdots x_i \cdots\} & =_{df} \{z \mid \cdots \vee\, z = x_i \vee \cdots\}
\end{array}$$

Now we proceed to the interpretation of the sets of Z_C.

$$\begin{array}{lll}
(i) & \mathcal{C}[\![\mathbb{N}]\!] & =_{df} \mathbb{N} \\
(ii) & \mathcal{C}[\![C_0 \times C_1]\!] & =_{df} \mathcal{C}[\![C_0]\!] \times \mathcal{C}[\![C_1]\!] \\
(iii) & \mathcal{C}[\![\mathbb{P}\,C]\!] & =_{df} \mathbb{P}\mathcal{C}[\![C]\!] \\
(iv) & \mathcal{C}[\![[\cdots l_i \in C_i \cdots]]\!] & =_{df} \Pi(x \in \kappa_0)\kappa_1(x) \\
(v) & \mathcal{C}[\![C_0 \twoheadrightarrow C_1]\!] & =_{df} \mathcal{C}[\![C_0]\!] \twoheadrightarrow \mathcal{C}[\![C_1]\!] \\
(vi) & \mathcal{C}[\![\{z \in C \mid P\}]\!] & =_{df} \{z \mid z \in \mathcal{C}[\![C]\!] \wedge \mathcal{P}[\![P]\!]\}
\end{array}$$

where $\kappa_0 =_{df} \{\cdots l_i \cdots\}$
and $\kappa_1(x) =_{df} \{z \mid x \in \kappa_0 \wedge \cdots z = x_i \Rightarrow z \in \mathcal{C}[\![C_i]\!] \cdots\}$.

5.2 Propositions

Propositions are translated as follows:

$$
\begin{array}{lll}
(i) & \mathcal{P}[\![\bot]\!] & =_{df} \bot \\
(ii) & \mathcal{P}[\![t_0 = t_1]\!] & =_{df} \mathcal{T}[\![t_0]\!] = \mathcal{T}[\![t_1]\!] \\
(iii) & \mathcal{P}[\![t \in C]\!] & =_{df} \mathcal{T}[\![t]\!] \in \mathcal{C}[\![C]\!] \\
(iv) & \mathcal{P}[\![\neg P]\!] & =_{df} \neg \mathcal{P}[\![P]\!] \\
(v) & \mathcal{P}[\![P_0 \vee P_1]\!] & =_{df} \mathcal{P}[\![P_0]\!] \vee \mathcal{P}[\![P_1]\!] \\
(vi) & \mathcal{P}[\![P_0 \wedge P_1]\!] & =_{df} \mathcal{P}[\![P_0]\!] \wedge \mathcal{P}[\![P_1]\!] \\
(vii) & \mathcal{P}[\![P_0 \Rightarrow P_1]\!] & =_{df} \mathcal{P}[\![P_0]\!] \Rightarrow \mathcal{P}[\![P_1]\!] \\
(viii) & \mathcal{P}[\![\forall z \in C \bullet P]\!] & =_{df} (\forall z)(z \in \mathcal{C}[\![C]\!] \Rightarrow \mathcal{P}[\![P]\!]) \\
(ix) & \mathcal{P}[\![\exists z \in C \bullet P]\!] & =_{df} (\exists z)(z \in \mathcal{C}[\![C]\!] \wedge \mathcal{P}[\![P]\!])
\end{array}
$$

Contexts, which are sets of propositions, are translated pointwise.

5.3 Terms

First we introduce an operator for restriction in $\mathcal{T}$.

$$e \restriction [\cdots l_i : T_i \cdots] =_{df} \lambda x.\mathit{if}\ \cdots \mathit{or}\ x = l_i\ \mathit{or} \cdots \mathit{then}\ e\ x$$

$$
\begin{array}{lll}
(i) & \mathcal{T}[\![x]\!] & =_{df} = x \\
(ii) & \mathcal{T}[\![0]\!] & =_{df} 0 \\
(iii) & \mathcal{T}[\![\mathit{succ}\ n]\!] & =_{df} \mathit{succ}\ \mathcal{T}[\![n]\!] \\
(iv) & \mathcal{T}[\![C]\!] & =_{df} \mathcal{C}[\![C]\!] \\
(v) & \mathcal{T}[\![t.l]\!] & =_{df} \mathcal{T}[\![t]\!](l) \\
(vi) & \mathcal{T}[\![(\!| \cdots l_i \Rrightarrow t_i \cdots |\!)]\!] & =_{df} \lambda x.e_0 \\
(vii) & \mathcal{T}[\![t.1]\!] & =_{df} (\mathcal{T}[\![t]\!])_0 \\
(viii) & \mathcal{T}[\![t.2]\!] & =_{df} (\mathcal{T}[\![t]\!])_1 \\
(ix) & \mathcal{T}[\![(t_0, t_1)]\!] & =_{df} (\mathcal{T}[\![t_0]\!], \mathcal{T}[\![t_1]\!]) \\
(x) & \mathcal{T}[\![\lambda x.t]\!] & =_{df} \lambda x.\mathcal{T}[\![t]\!] \\
(xi) & \mathcal{T}[\![t_0\ t_1]\!] & =_{df} \mathcal{T}[\![t_0]\!](\mathcal{T}[\![t_1]\!]) \\
(xii) & \mathcal{T}[\![t \restriction [D]]\!] & =_{df} \mathcal{T}[\![t]\!] \restriction [D] \\
(xiii) & \mathcal{T}[\![\lambda x.t]\!] & =_{df} \lambda x.\mathcal{T}[\![t]\!] \\
(xix) & \mathcal{T}[\![t_0\ t_1]\!] & =_{df} \mathcal{T}[\![t_0]\!]\ \mathcal{T}[\![t_1]\!] \\
(xx) & \mathcal{T}[\![\mathbf{e}]\!] & =_{df} \mathbf{e} \\
(xxi) & \mathcal{T}[\![t_0 \wedge t_1]\!] & =_{df} \lambda x.\mathit{if}\ \mathcal{T}[\![t_0]\!] = \mathbf{e}\ \mathit{then}\ \mathcal{T}[\![t_1]\!]\ \mathit{else}\ \mathcal{T}[\![t_0]\!]
\end{array}
$$

where:

$$e_0 =_{df} \mathit{if}\ \cdots \mathit{elsif}\ (x = l_i)\ \mathit{then}\ \mathcal{T}[\![t_i]\!]\ \mathit{elsif} \cdots\ \mathit{else}\ \mathbf{e}$$

We must, of course, show that the interpretation is *correct.*

Proposition 3 Soundness.

$$
\begin{array}{ll}
(i)\ \mathit{If}\ t^T & \mathit{then}\ \mathcal{T}[\![t]\!] \in \mathcal{C}[\![T]\!] \\
(ii)\ \mathit{If}\ \Gamma \vdash_C P & \mathit{then}\ \mathcal{P}[\![\Gamma]\!] \vdash_{\mathcal{T}} \mathcal{P}[\![P]\!]
\end{array}
$$

Proof. (i) By induction on the structure of t. (ii) By induction on the structure of the antecedent derivation. □

6 The term assignment system $\mathcal{T}^{\mathcal{TA}}$

We now turn to a *term assignment* formulation of the theory: $\mathcal{T}^{\mathcal{TA}}$. This is, in essence, an explicit presentation of the soundness proof for a realizability interpretation of $\mathcal{T}$ which incorporates a theory of information loss for Harrop formulæ ([2]). We begin by isolating that class of formulæ.

$$\begin{array}{lll}
\varphi_H & ::= & \alpha_H \mid \varphi_H \wedge \varphi_H \mid \varphi \Rightarrow \varphi_H \mid (\forall x)\varphi_H \mid (\forall x^{(n)})\varphi_H \\
\alpha_H & ::= & e \in \kappa_H \mid e = e \mid e{\downarrow} \mid \bot \\
\kappa_H^{(0)} & ::= & \mathbb{N} \mid \{x \mid \varphi_H^{(0)}(x)\} \\
\kappa_H^{(n+1)} & ::= & \kappa_H^{(n)} \mid \{x \mid \varphi_H^{(n+1)}(x)\}
\end{array}$$

These formulæ contain *no computational content.* The rules of $\mathcal{T}^{\mathcal{TA}}$ are arranged according to the incidence of these Harrop formulæ in order to suppress computationally irrelevant information. We cannot possibly provide the system in its entirety within the limits of this paper, but we can illustrate it with a number of examples which are of particular importance and which we will need explicitly in our examples in section 8. To simplify presentation we shall assume, from now on, that formulæ which are written φ (*etc.*), *i.e.* with no subscript, are *not* Harrop. Sequents, then, of the theory have the form: $\Gamma \vdash_{\mathcal{TA}} t : \varphi$ or $\Gamma \vdash_{\mathcal{TA}} \varphi_H$ where contexts are sets of judgements of the form φ_H or $x : \varphi$ and each such variable x may occur at most once.

As usual, as much as possible is suppressed in the presentation of the rules. The usual side conditions apply. Note that the existential elimination rule is analogous to the *strong* Σ-type of Martin-Löf type theory (see *e.g.* [1] p. ???).

$$\frac{\varphi_H(e) \quad e{\downarrow}}{e : (\exists x)\varphi_H(x)} \qquad \frac{e_0 : (\exists x)\varphi_H(x) \quad \varphi_H(y) \vdash e_1(y) : \eta(y)}{e_1(e_0) : \eta(e_0)} \qquad \frac{e : \varphi(x)}{\lambda x.e : (\forall x)\varphi(x)}$$

$$\frac{e_0 : (\forall x)\varphi(x) \quad e_1{\downarrow}}{e_0\ e_1 : \varphi(e_1)} \qquad \frac{\textit{if } \varphi_D \textit{ then } e : \varphi_D \Rightarrow \varphi \quad \varphi_D}{e : \varphi} \qquad \frac{\varphi_D \vdash e : \varphi}{\vdash \textit{if } \varphi_D \textit{ then } e : \varphi_H \Rightarrow \varphi}$$

$$\frac{\varphi_H \vdash e : \psi}{\lambda x.e : (\forall x)(\varphi_H \Rightarrow \psi)} \qquad \frac{e : (\forall x)(\varphi_H(x) \Rightarrow \psi(x)) \quad \varphi_H(e')}{e\ e' : \psi(e')}$$

There are many more rules in $\mathcal{T}^{\mathcal{TA}}$ than in $\mathcal{T}$. For example, there are *four* formulations of the introduction rule for conjunction which correspond to the four possible forms (Harrop or non-Harrop) in which the two conjuncts might occur. Nevertheless, providing one respects the incidence of Harrop formulæ, it is easy to see how each proof in $\mathcal{T}$ can be injected into a corresponding proof in $\mathcal{T}^{\mathcal{TA}}$. Additionally, we have already established a sound translation of Z_C into $\mathcal{T}$. Consequently, we have a mechanism for making the computational content of a proof in Z_C explicit.

A little reflection will show that a PCO in Z_C will be translated into a proposition of $\mathcal{T}$ of the form:

$$(\forall z)(\varphi_0(z) \Rightarrow (\exists y)(\varphi_1(z, y)))$$

in which the constituent propositions φ_0, φ_1 are decidable (more usually known as Π^0_2 statements). In view of the fact that De Morgan's laws apply in full generality to *decidable* propositions (in a constructive framework) these constituent propositions are, moreover, equivalently presented as Harrop formulæ. We shall assume in the sequel that all decidable propositions are represented in $\mathcal{T}$ in Harrop form. For example, a proof of a PCO in the Z_C logic will induce a proof in $\mathcal{T}^{\mathcal{TA}}$ of the corresponding Π^0_2 statement in $\mathcal{T}$. We have, roughly, the following picture:

$$\frac{\vdots}{e : (\forall\, z)(\varphi_H(z) \Rightarrow (\exists\, y)(\psi_H(z, y)))}$$

such that $\psi(z, e\ z)$, whenever $\varphi_H(z)$ for some z. The term e is an operation (program) which meets the representation of the PCO in $\mathcal{T}$. Moreover, e, provably, represents an element of the operation schema which gave rise to the PCO.

Proposition 4.

If $\vdash_{\mathcal{T}^{\mathcal{TA}}} e : [\![PCO(S)]\!]$ *then there exists* t *such that* $\vdash_C t \in S \wedge \mathcal{T}[\![t]\!] = e$

□

This sketch omits many mathematical details, though they are, in fact, quite easy to check.

7 The Operation schema calculus

At the core of Z is the schema calculus. In standard Z, and in our earlier treatment of the schema calculus too, there is no distinction between state and operation schema. Consequently, the sketch of a schema calculus logic that we gave earlier for *state* schema is sufficient. In our current enterprise we have a formal distinction and we therefore require a separate treatment for expressions formed from operation schema.

We begin with conjunction of operation schemas.
Let $U_j =_{df} [\Delta S_j;\ \mathbf{in}\ D_{j_i};\ \mathbf{out}\ D_{j_o} \mid \mathbf{pre}\ P_j;\ \mathbf{post}\ Q_j]$ for $j \in 2$.

$$\begin{aligned} U_0 \wedge U_1 =_{df} \{f \in S_{i_0}^{\mathbb{P}\, T_{i_0}} \wedge S_{i_1}^{\mathbb{P}\, T_{i_1}} \nrightarrow S_{o_0} \wedge S_{o_1} \mid\ & \exists f_0 \in U_0 \bullet \exists f_1 \in U_1 \bullet \\ & \forall x \in S_{i_0} \wedge S_{i_1} \bullet \\ & f_0\ (x \upharpoonright T_{i_0}) = (f\ x) \upharpoonright T_{o_0} \wedge \\ & f_1\ (x \upharpoonright T_{i_1}) = (f\ x) \upharpoonright T_{o_1}\} \end{aligned}$$

The most important derived rule we can then obtain is is the introduction rule which enables us to decompose a specification into components:

$$\frac{f_0 \in U_0 \quad f_1 \in U_1}{f_0 \wedge f_1 \in U_0 \wedge U_1}$$

where, for partial functions $f_0^{T_0 \nrightarrow T_1}$ and $f_1^{T_2 \nrightarrow T_3}$ we have:

$$f_0 \wedge f_1 =_{df} \lambda z.(f_0\ (z \upharpoonright T_0)) \wedge (f_1\ (z \upharpoonright T_2))$$

Note that this form of operation conjunction is not associative: the is the penalty one pays for a rule which applies to *any* implementations of the premise operations. There is, however, a useful special case which applies when the underlying types of the components are the same.

$$\frac{f \in U_0 \quad f \in U_1}{f \in U_0 \wedge U_1}$$

This permits the proof derivation obligations to be split.

Disjunctions are handled in a similar fashion:

$$\begin{aligned}\{f \in S_{i_0}^{\mathbb{P}\, T_{i_0}} \wedge S_{i_1}^{\mathbb{P}\, T_{i_1}} \nrightarrow S_{o_0} \wedge S_{o_1} \mid \exists f_0 \in U_0 \bullet \exists f_1 \in U_1 \bullet \\ \forall x \in S_{i_0} \wedge S_{i_1} \bullet (f_0\ (x \upharpoonright T_{i_0}) = \mathbf{e} \Rightarrow f_1\ (x \upharpoonright T_{i_1}) = (f\ x) \upharpoonright T_{o_1}) \wedge \\ (f_1\ (x \upharpoonright T_{i_1}) = \mathbf{e} \Rightarrow f_0\ (x \upharpoonright T_{i_0}) = (f\ x) \upharpoonright T_{o_0})\}\end{aligned}$$

The introduction rule which enables us to decompose a specification into components is then:

$$\frac{f_0 \in U_0 \quad f_1 \in U_1}{f_0 \vee f_1 \in U_0 \vee U_1}$$

where, for partial functions $f_0^{T_0 \nrightarrow T_1}$ and $f_1^{T_2 \nrightarrow T_3}$ we have:

$$f_0 \vee f_1 =_{df} \lambda z.(f_0\ (z \upharpoonright T_0)) \vee (f_1\ (z \upharpoonright T_2))$$

and

$$t_0 \vee t_1 =_{df} \mathit{if}\ t_0 = \mathbf{e} \wedge t_1 \neq \mathbf{e}\ \mathit{then}\ t_1\ \mathit{elsif}\ t_0 \neq \mathbf{e} \wedge t_1 = \mathbf{e}\ \mathit{then}\ t_0\ \mathit{else}\ \mathbf{e}$$

Finally, we turn to perhaps one of the most important operations used for structuring complex operations: composition. In standard Z this factors into two operators called *composition* and *piping*. The former connects *states* whereas the latter connects *inputs* with *outputs*. In our setting, in this paper at least, we have a *single* operation which connects one program with another; connecting both states and inputs with outputs.

Our definition of composition, for operation schema U_0 and U_1 is:

$$U_0^{\mathbb{P}(T_0 \nrightarrow T_1)};\ U_1^{\mathbb{P}(T_1 \nrightarrow T_2)} = \{f \in T_0 \nrightarrow T_2 \mid \exists f_0 \in U_0 \bullet \exists f_1 \in U_1 \bullet f = f_0;\ f_1\}$$

where, as usual, (reverse) composition at the term level is given by $f;\ g = \lambda z.g\ (f\ x)$.

Note that our composition operator is when the inputs of the second component agree with the outputs of the first: as we indicated earlier, our notational innovations for operation schema significantly simplify the technical presentation.

We then have the following derived rule for program construction:

$$\frac{f_0 \in U_0 \quad f_1 \in U_1}{f_0;\ f_1 \in U_0;\ U_1}$$

There is much more that could be said on this topic, and this will be covered in detail in a later, fuller version of this paper. The approach we have taken to treat propositional operators, for example, is only one of a number of possibilities which we will develop there.

8 Some illustrative examples

We provide only illustrative examples in this section which, we hope, outline the trajectory of our approach and demonstrate how the rather crude lambda notation of the underlying model can be enriched in a useful way.

Example 1. Consider the state schema $S =_{df} [z \in \mathbb{N}]$ and the operation schema:

$$Inc =_{df} [\Delta S \mid \mathbf{post}\ z' = z + 1]$$

The program construction obligation for *Inc* is then simply:

$$\forall x \in S \bullet \exists y \in S \bullet y.z = x.z + 1$$

We may prove this in the logic as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x \in S \vdash x \in S}\,(ass)}{x \in S \vdash x.z \in \mathbb{N}}\,([]^-) \quad \dfrac{}{x \in S \vdash 0 \in \mathbb{N}}\,(\mathbb{N}_0^+) \quad \dfrac{}{x \in S \vdash 1 \in \mathbb{N}}\,(\mathbb{N}_1^+)}{x \in S \vdash x.z + 1 \in \mathbb{N}}\,([]^+)}{x \in S \vdash \langle\!|\ z \Rrightarrow x.z + 1\ |\!\rangle \in [z \in \mathbb{N}]}}{x \in S \vdash \langle\!|\ z \Rrightarrow x.z + 1\ |\!\rangle \in S}\,(S^+) \quad \dfrac{\dfrac{\dfrac{\dfrac{}{x \in S \vdash x \in S}\,(ass)}{x \in S \vdash x\!\downarrow}}{x \in S \vdash (x.z)\!\downarrow}\,(\downarrow) \quad \dfrac{}{x \in S \vdash 1\!\downarrow}}{x \in S \vdash (x.z + 1)\!\downarrow}}{x \in S \vdash \langle\!|\ z \Rrightarrow x.z + 1\ |\!\rangle.z = x.z + 1}}{x \in S \vdash \exists y \in S \bullet y.z = x.z + 1}$$
$$\forall x \in S \bullet \exists y \in S \bullet y.z = x.z + 1\,(\forall^+)$$

If we inject this derivation into the term assignment version of the underlying theory of operations and types $\mathcal{T}^{\mathcal{TA}}$, we obtain the following term assignment version of the derivation[4]:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash (\lambda v.(\sigma\ v) + 1) \in \Pi(v \in \{z\})\kappa(v)}}{\Gamma \vdash (\lambda v.(\sigma\ v) + 1) \in \mathcal{C}[\![S]\!]} \quad \dfrac{}{\Gamma \vdash (\lambda v.(\sigma\ v) + 1)\ z = (\sigma\ z) + 1)}}{\sigma \in \mathcal{C}[\![S]\!] \vdash (\lambda v.(\sigma\ v) + 1) \in \mathcal{C}[\![S]\!] \wedge (\lambda v.(\sigma\ v) + 1)\ z = (\sigma\ z) + 1)}}{\sigma \in \mathcal{C}[\![S]\!] \vdash \lambda v.(\sigma\ v) + 1 : (\exists y)(y \in \mathcal{C}[\![S]\!] \wedge y\ z = (\sigma\ z) + 1)}$$
$$\lambda\sigma.\lambda v.(\sigma\ v) + 1 : (\forall x)(x \in \mathcal{C}[\![S]\!] \Rightarrow (\exists y)(y \in \mathcal{C}[\![S]\!] \wedge y\ z = (x\ z) + 1)$$

Where $\kappa(v) = \{u \mid\ v \in \{z\} \Rightarrow u \in \mathbb{N}\}$ and $\Gamma =_{df} \sigma \in \mathcal{C}[\![S]\!]$.

$\lambda\sigma.\lambda v.(\sigma\ v) + 1$ is a very low-level program and, that fact notwithstanding, a *functional* program. Are we restricted in our framework to deriving only functional programs?

To see that we are not first requires us to note that we can treat the underlying term language of our systems as a *representation language* for higher level constructs. In this regard we are no more restricted to deriving functional programs than are the techniques of denotational semantics restricted to the

[4] We shall use the variable name σ in what follows to underline the *state* variable over which a derived program operates.

account of functional programmming languages (recall that the underlying representation language of denotational semantics is not fundamentally different to what we have here). What we must do is to develop appropriate programming idioms on the term language at our disposal. We can illustrate the idea easily enough with the current example. Consider a simple language of assignment commands:

$$\begin{array}{lll} cmd & ::= & x := exp \\ exp & ::= & x \mid n \mid exp + exp \end{array}$$

Then a translation into our underlying term language:

$$\begin{array}{ll} [\![x := exp]\!] & =_{df} \lambda\sigma.\lambda v.\mathit{if}\ v = x\ \mathit{then}\ [\![exp]\!]\sigma\ \mathit{else}\ \sigma\ x \\ [\![x]\!] & =_{df} \lambda\sigma.\sigma\ x \\ [\![n]\!] & =_{df} \lambda\sigma.n \\ [\![exp_0\ + exp_1]\!] & =_{df} \lambda\sigma.[\![exp_0]\!]\sigma + [\![exp_1]\!]\sigma \end{array}$$

This begs the question of the precise *type* of states (the variables σ). A full account would require this to be a parameter of the definition and the use of dependent types. We must avoid the details here.

Under the auspices of this simple regime, our program, obtained from the derivation above, collapses to the command[5]:

$$z := z + 1$$

Note that we have the corollary:

$$z := z + 1 \in [\Delta S \mid \mathbf{post}\ z' = z + 1]$$

as a consequence of the way the system has been set up.

Example 2. We can now build upon our simple example. Suppose that we make the following specification:

$$Dinc =_{df} Inc;\ Inc$$

It is then rather easy to establish the following:

$$\frac{\begin{array}{cc} \vdots & \vdots \\ x := x + 1 \in Inc & x := x + 1 \in Inc \end{array}}{x := x + 2 \in Dinc}$$

where the ellipsis stands for the derivation we developed in the previous example.

Alternatively, we might extend our imperative programming fragment as follows:

$$cmd\ ::= \cdots \mid cmd;\ cmd$$

[5] There is no conditional in our lambda term because the state comprises only a *single* label: under the typing constraints imposed by the proof the terms $\lambda v.(\sigma\ v) + 1$ and $\lambda v.\mathit{if}\ v = z\ \mathit{then}\ (\sigma\ v) + 1\ \mathit{else}\ \sigma\ v$ are *equal*.

and the interpretation with new clause:

$$[\![cmd_0;\ cmd_1]\!] =_{df} \lambda\sigma.\, [\![cmd_0]\!]\ ([\![cmd_1]\!]\,\sigma)$$

and then we also have:

$$\frac{\begin{array}{cc}\vdots & \vdots \\ x := x+1 \in Inc & x := x+1 \in Inc\end{array}}{x := x+1;\ x := x+1 \in Dinc}$$

Example 3. Our next example concerns the use of disjunction in the operation schema calculus. Consider the following two state schemas:

$$S_0 =_{df} [z, x_0 \in \mathbb{N} \mid x_0 = 0]$$
$$S_1 =_{df} [z, x_1 \in \mathbb{N} \mid x_1 = 1]$$

We define two operation schema, each over one of the states above:

$$Op_0 =_{df} [\Delta S_0 \mid \mathbf{pre}\ even(z)\ \mathbf{post}\ z' = x_0 \wedge x_0' = x_0]$$
$$Op_1 =_{df} [\Delta S_1 \mid \mathbf{pre}\ odd(z)\ \mathbf{post}\ z' = x_1 \wedge x_1' = x_1]$$

Our specification, finally, is:

$$S =_{df} Op_0 \vee Op_1$$

We can now present a program derivation for this simple situation. First we can derive a program for Op_0:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{u \in S_0, even(u.z) \vdash u \in S_0}}{u \in S_0, even(u.z) \vdash u\!\downarrow}}{u \in S_0, even(u.z) \vdash u.x_0\!\downarrow}}{u \in S_0, even(u.z) \vdash t.z = u.x_0} \qquad \dfrac{\begin{array}{c}similarly \\ \vdots \\ u.x_0\!\downarrow\end{array}}{u \in S_0, even(u.z) \vdash t.x_0 = u.x_0}}{u \in S_0, even(u.z) \vdash t.z = u.x_0 \wedge t.x_0 = u.x_0}}{u \in S_0 \vdash \exists\, v \in S_0 \bullet even(u.z) \Rightarrow v.z = u.x_0 \wedge v.x_0 = u.x_0}}{\forall\, u \in S_0 \bullet \exists\, v \in S_0 \bullet even(u.z) \Rightarrow v.z = u.x_0 \wedge v.x_0 = u.x_0}$$

where $t =_{df} (\!|\ z \Rrightarrow u.x_0, x_0 \Rrightarrow u.x_0\ |\!)$. The program which is obtained from this derivation is:

$$\lambda\sigma.\mathit{if}\ even(\sigma\ z)\ \mathit{then}\ (\!|\ z \Rrightarrow \sigma\ x_0, x_0 \Rrightarrow \sigma\ x_0\ |\!)\ \mathit{else}\ \mathbf{e}$$

Let us extend[6] our small imperative fragment as follows:

$$\begin{array}{lll} cmd & ::= & \cdots \mid \mathit{if}\ exp\ \mathit{then}\ cmd_0\ \mathit{else}\ cmd_1 \\ exp & ::= & \cdots \mid P(\cdots exp_i \cdots) \mid \mathbf{e} \end{array}$$

[6] The notational ambiguity we introduce here is clear enough, give its very limited role in this example.

when P is an n-ary decidable predicate. The interpretation is also extended with new clauses:

$$\begin{array}{ll} [\![\mathit{if}\ \mathit{exp}\ \mathit{then}\ \mathit{cmd}_0\ \mathit{else}\ \mathit{cmd}_1]\!] & =_{df} \lambda\sigma.\mathit{if}\ [\![\mathit{exp}]\!]\sigma\ \mathit{then}\ [\![\mathit{cmd}_0]\!]\sigma\ \mathit{else}\ [\![\mathit{cmd}_1]\!]\sigma \\ [\![P(\cdots \mathit{exp}_i \cdots)]\!] & =_{df} \lambda\sigma.P(\cdots [\![\mathit{exp}_i]\!]\sigma \cdots) \\ [\![\mathbf{e}]\!] & =_{df} \lambda\sigma.\mathbf{e} \end{array}$$

Then we can express our program (and our conclusion) by means of:

$$\mathit{if}\ \mathit{even}(z)\ \mathit{then}\ z := x_0\ \mathit{else}\ \mathbf{e} \in \mathit{Op}_0$$

Naturally, there is a similar derivation which demonstates that

$$\mathit{if}\ \mathit{odd}(z)\ \mathit{then}\ z := x_1\ \mathit{else}\ \mathbf{e} \in \mathit{Op}_1$$

Putting these together we can show, using the $\vee$-introduction rule for schema operation, the following:

$$\frac{\begin{array}{cc} \vdots & \vdots \\ \mathit{if}\ \mathit{even}(z)\ \mathit{then}\ z := x_0\ \mathit{else}\ \mathbf{e} \in \mathit{Op}_0 & \mathit{if}\ \mathit{odd}(z)\ \mathit{then}\ z := x_1\ \mathit{else}\ \mathbf{e} \in \mathit{Op}_1 \end{array}}{\mathit{if}\ \mathit{even}(z)\ \mathit{then}\ z := x_0\ \mathit{else}\ z := x_1 \in S}$$

This relies on the observation that:

$$\mathit{if}\ \varphi_D\ \mathit{then}\ e_0\ \mathit{else}\ \mathbf{e} \vee \mathit{if}\ \neg\varphi_D\ \mathit{then}\ e_1\ \mathit{else}\ \mathbf{e} = \mathit{if}\ \varphi_D\ \mathit{then}\ e_0\ \mathit{else}\ e_1$$

Example 4. Our final, short example, illustrates the use of the special rule for operation conjunction. Consider the state S of example 1 and the operation schemas $\mathit{Op}_0 =_{df} [\Delta S \mid z' > 0]$ and $\mathit{Op}_1 =_{df} [\Delta S \mid z' < 10]$. It will be evident, by this point, how we may show that $z := 5 \in \mathit{Op}_0$ and $z := 5 \in \mathit{Op}_1$. Then, by conjunction introduction, we have:

$$\frac{x := 5 \in \mathit{Op}_0 \quad x := 5 \in \mathit{Op}_1}{x := 5 \in \mathit{Op}_0 \wedge \mathit{Op}_1}$$

Here, conjunction is used to decompose a loose specification into yet looser components.

9 Conclusions and future work

In this paper we have provided a specification logic Z_C which forms the basis for a logic of Z (see [3]). We further described a new interpretation or semantics for Z_C (and, therefore, for Z) within a constructive and intensional theory of operations and kinds. Finally, we have demonstrated, if only in outline, how this novel interpretation can be used to integrate program development in a highly structured fashion: an approach which reflects the high level structure of the specification.

We have argued that the functional programming notation which forms the basis of the theory $\mathcal{T}$ should, by no means, be taken to indicate either a restriction in, or a preference for a particular, programming paradigm and we made the analogy with the meta-language of denotational semantics, which is also functional, to support this position. Indeed, we have shown that similar techniques can be employed to integrate the *imperative* programming paradigm within our framework. These examples, however, are extremely rudimentary and serve only, at this stage of our research, to indicate *strategy*, rather than to prove the adequacy or success of our approach. The project of developing a fully integrated, logically based, methodology for program development in the context of the Z specification language is an enormous one, and is one that requires care and precise development. We believe we have made an interesting start, and one which may be developed, in the ways we have indicated here, in the future. An implementation of the system described here, in Isabelle [4], is currently under construction [6] and this, we hope, will prove to be of great help in driving the research forward.

Acknowledgements

We would like to thank the Department of Computer Science at the University of Waikato, New Zealand, the Centre for Discrete Mathematics and Theoretical Computer Science, New Zealand, the Royal Society of Great Britain, and the EPSRC (grant number GR/L57913) for financial assistance which has supported the development of this research. We are most grateful to Rob Arthan, Jonathan Bowen, Ricardo Calderon-Cruz, Doug Goldson, Lindsay Groves, Andrew Martin, Ian Toyn, Ray Turner, Mark Utting, Sam Valentine, Norbert Völker and Jim Woodcock for many useful discussions.

References

1. M. Beeson. *Foundations of Constructive Mathematics.* Springer Verlag, 1985.
2. R. Harrop. On disjunctions and existential statements in intuitionistic logic. *Math. Ann.*, 132:347–361, 1956.
3. M. C. Henson and S. V. Reeves. Revising Z: semantics and logic. *Formal Aspects of Computing (submitted)*, 0:000–000, 1998.
4. L. Paulson. *Isabelle: A generic theorem prover.* Springer Verlag, LNCS Vol. 828, 1994.
5. J. M. Spivey. *The Z notation: A reference manual.* Prentice Hall, 1992.
6. N. Völker. Private communication, 1998.
7. J. Woodcock and J. Davies. *Using Z: Specification, Refinement and Proof.* Prentice Hall, 1996.