Under consideration for publication in Formal Aspects of Computing

A logic for schema-based program development

Martin C. Henson[1] and Steve Reeves[2]

[1]Department of Computer Science, University of Essex, U.K;
[2]Department of Computer Science, University of Waikato, New Zealand

Abstract. We show how a theory of specification refinement and program development can be constructed as a conservative extension of our existing logic for Z. The resulting system can be set up as a development method for Z, or as a generalisation of a refinement calculus (with a novel semantics). In addition to the technical development we illustrate how the theory can be used in practice.

1. Introduction

In this paper we will present a theory of specification refinement and implementation (and hence program) development based on the schema calculus of Z. We will show how it can be constructed as a conservative extension of our existing logic for Z (see [6], [7], [8]) and illustrate the use of the theory in practice with a number of examples.
The basis of our approach is to model a specification as a set of *legitimate implementations*. Thus, when p is a program (an implementation) and U is a specification, the proposition:

$$p \in U$$

which is understood so that:

$$[\![p \in U]\!] =_{df} [\![p]\!] \in [\![U]\!]$$

expresses the claim that p correctly implements U.
Refinement is then simply containment:

$$[\![U_0 \sqsupseteq U_1]\!] =_{df} [\![U_0]\!] \subseteq [\![U_1]\!]$$

In the approach we develop here, then, we take the unusual step of distinguishing between implementations and specifications, eschewing the more common view that implementations are special cases of specifications (*e.g.* [11]). This distinction, however, lives quite happily with the standard development *methodology* in which only refinement appears explicitly. This will be as simple as observing that:

$$p \in U \Leftrightarrow \{p\} \sqsupseteq U$$

An immediate consequence of this distinction is that the relation of *implementation* becomes for us the basic

Correspondence and offprint requests to: Martin Henson, Department of Computer Science, University of Essex, Wivenhoe Park, Colchester, Essex, C04 3SQ, U.K.

notion, and the relation of *refinement* is then derived from that. This offers us the opportunity to investigate a semantics for specifications which would not otherwise be available; moreover, as we will see, it possesses very satisfactory mathematical and pragmatic properties.

1.1. The specification language and its logic

The specifications we wish to consider are, in general, schema *expressions* constructed in a similar manner to the *algebra of schemas* of the specification language Z. This algebra is one of the major features of Z and enables specifications to be presented and developed in a highly structured manner (see *e.g.* [13]). The algebra becomes a schema *calculus* when the Z language is extended to a Z *logic*, as it does in the context of our earlier work, cited earlier.
The main thrust of this paper is to develop an approach to program development by refinement in which one may use an expressive language for specifications. The schema algebra of Z is an excellent example of such a language, and we shall explain in detail how a logical theory of refinement can be combined smoothly with the logic of the schema calculus.
Although our notation resembles Z, our technical contribution can be viewed in two significantly different ways. Firstly, the work can be viewed as providing Z with an integrated logic of operation refinement and program development. Secondly, we can view the *atomic* schema (the simplest specifications that do not involve schema operators) as *specification statements*, as they appear in refinement calculus (*e.g.* [11]), albeit with a modified syntax, but much more significantly, with a different (non weakest-precondition) semantics. We will explore the technical issues at the point where the Z and refinement calculus threads part company below in sections 3.2 and 3.3.1; in the latter section showing exactly the different semantics required for atomic Z schemas and for specification statements. Our main focus, in fact, will be on the latter approach (syntactic rather than logical preconditions), and our examples in section 6 will be based on this model.

1.2. The implementation language and its logic

An implementation of a specification, *i.e.* one of its members, will be a function with appropriate properties: roughly it will be a function that takes any state satisfying the preconditions of the specification to a state that satisfies its postconditions. The language we use to denote functions will be the λ-notation. Thus, the language and logic for implementations will be a very general one, since the underlying language and the criteria for being an implementation (and not just any function) are both very general, *i.e.* not tied to any one view of implementation.
This generality can be exploited when we have a particular target programming language in mind as the computational vehicle for our implementations. We can view the language of implementations rather as we view the λ-notation in the context of denotational semantics, and so we can, on top of the (single) logic of implementations, have a particular programming language with its semantics given in the usual (denotational) way via a semantic function. Then, of course, we can derive rules for program development in our target language via the implementation logic and the semantic function.
Thus, the logic of specifications and implementations we give here can be specialised to a variety target programming languages. We give an example of this for a simple imperative language in section 5 below, and our worked examples in section 6 will be based on that.

1.3. Related Work

The point of departure for much of the relevant related work is the observation that Z is a specification language which lacks a program development method, whereas the refinement calculus is a programming development method lacking an expressive specification language. It is perhaps unsurprising that such related work aims to connect these two frameworks in order to provide that which each alone lacks.
Perhaps the earliest attempt at a synthesis along these lines was that of King [9] and the most recent, comprehensive and formal approach, ZRC, due to Cavalcanti and Woodcock [1] [2]. In this work, Z is provided with a weakest precondition semantics equivalent to its standard relational semantics, and as a result can be integrated with the specification statement and refinement calculus more generally. The

passage from Z to specification statements is mediated by the conversion law bC which can be proved sound in view of the uniform semantics. ZRC has refinement laws involving schema operators, such as conjunction and disjunction, but with stringent and unwelcome sideconditions. These are a consequence of the weakest precondition semantics: recall that the semantics of the specification statement ensures that observations outside the frame *must remain unchanged*. This does not fit in well with schema expressions in Z in which the atomic schema components may have disjoint or overlapping (rather than equal) alphabets. Indeed, a major motivation for us in adopting the semantics we have employed is that it is a much more appropriate basis for the refinement of Z specifications. Groves [4] considers the conjunction of specification statements and introduces special technical devices in order to address the problem of non-equal frames. Our semantic basis does not force observations outside the alphabet of a schema to remain unchanged and, consequently, no special measures need be taken in order to handle schema expressions involving non-equal alphabets.
We also take work investigating program development within constructive theories as a natural precursor to our own. Although $\mathcal{Z}_\mathcal{C}$ is a classical system, the work reported here grew out of earlier investigations based on constructive logic (see [5] for example). Indeed, the idea of basing a program development framework upon a relation of membership, while unfamiliar in classical approaches such as the refinement calculus, is fundamental to approaches based on constructive systems. For example, Martin-Löf made the observation [10] that the judgement $a : A$, in his theory of types, could be read either as "a is a proof of the proposition A" or alternatively as "a is a program that meets the specification A"; this gave initial impetus to the entire research area. The advantages of constructive program development include very natural and powerful methods for developing recursive programs inductively. On the other hand, the specification language, while powerful, is not expressive: specifications are essentially Π^0_2 statements. Additionally, the natural programming notation for this approach to program development is functional rather than imperative.
What we hope to illustrate in this paper is that our approach builds on the advantages of earlier approaches and avoids some of their limitations. We retain, for example, the powerful links between induction and recursion from constructive approaches, while at the same time permitting imperative program development. We also retain the essential core of the refinement calculus (albeit based on a novel, alternative semantics) while permitting a much more expressive specification language. Moreover, as we explained earlier, we offer our alternative semantics as a more satisfactory basis for this generalised specification language.

1.4. Organisation and summary of the paper

The paper is organised as follows. In section 2 we provide an overview of the logic of the state schema calculus. This logic is a conservative extension of the basis logic $\mathcal{Z}_\mathcal{C}$. Full details may be found in [8] though, for convenience, we have included a summary in appendix A. The appendix also contains useful information regarding the notation we have employed in the paper.
In section 3 we describe our semantics for operation schemas, since in this framework the semantics is distinct from that of the state schema calculus. In particular, we will explore the issues which separate Z operation schemas from the specification statements of the refinement calculus and provide semantics for both approaches. Alongside the semantics, we provide full technical details of our operation schema logic (or calculus). In addition to the usual methods for combining schemas, we introduce novel operations of schema abstraction and schema application which are critical for the development of procedures.
Section 4 is devoted to the logic of refinement. Here we will establish and explore an inequational logic of refinement covering the entire operation schema calculus.
In section 5 we develop one possible, and very simple, application of the framework. We describe a simple programming language, establish its semantics in the underlying logic $\mathcal{Z}_\mathcal{C}$ and then link programs to specifications through the general notions of refinement and implementation. This leads to a battery of implementation rules tailored to the programming language in question. It should be noted that this language is but one simple application of the framework we established in the previous sections.
Section 6 is devoted to simple examples. We do not provide large and complex examples, merely illustrating the theory and its application to the simple programming language $\mathcal{P}_\mathbb{N}$. We do, however, demonstrate some aspects of specification, refinement and program development using the schema logic, including simple examples of promotion.
The paper finishes with some concluding remarks, indications for future work, acknowledgements, references and an appendix on the core logic $\mathcal{Z}_\mathcal{C}$.

2. The state schema calculus

In this section and section 3 below, we extend the core specification logic $\mathcal{Z}_\mathcal{C}$ to cover the language of Z schemas. In this paper, and in contrast to our earlier work, we distinguish *state* schemas and *operation* schemas. The methods we adopt here for the state schema calculus are those we have previously introduced and, since the details have been thoroughly investigated before, we now present an overview. The reader is invited to consult [8] for a comprehensive treatment. The explanation of unfamiliar notation, and further details of the underlying logic $\mathcal{Z}_\mathcal{C}$, are give in appendix A

2.1. State schema sets and atomic state schemas

Let $T = [\cdots \mathbf{z}_i : T_i \cdots]$; then the syntax of basic state schemas is:

$$S^{\mathbb{P}\,T} ::= [\cdots \mathbf{z}_i : C_i^{T_i} \cdots] \mid [S^{\mathbb{P}\,T} \mid P]$$

These are the *state schema sets* and *atomic state schemas* respectively. As usual, we will write schemas of the form: $[[\cdots \mathbf{z}_i : C_i \cdots] \mid P]$ as $[\cdots \mathbf{z}_i : C_i \cdots \mid P]$. We allow the obvious generalisation of our alphabet operator to atomic state schemas and state schema sets: $\alpha[S \mid P] =_{df} \alpha S$ and $\alpha[\cdots \mathbf{z}_i : C_i^{T_i} \cdots] =_{df} \alpha[\cdots \mathbf{z}_i : T_i \cdots]$. Note that, as in our earlier work, observations may occur as constants of the appropriate type in propositions occurring in schemas. In the rules below it is clear that such constants become *bona fide* terms (leading to *bona fide* propositions) in the core logic $\mathcal{Z}_\mathcal{C}$ when eliminated from schemas: a schema proposition P becomes $z.P$ and so forth. This is, as we will see in detail later in the paper, a very similar phenomenon to the appearance of program variables in a program when, in the underlying logic, these variables naturally refer to values in a certain (implicit) state.
The rules for *schema sets* are:

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

and, for *atomic schemas*:

$$\frac{t \in S \quad t.P}{t \in [S \mid P]}\ (S^+) \qquad \frac{t \in [S \mid P]}{t \in S}\ (S^-_{\mathrm{o}}) \qquad \frac{t \in [S \mid P]}{t.P}\ (S^-_{\mathrm{1}})$$

2.2. State schema disjunction

When the schemas S_0 and S_1 have the types $\mathbb{P}\,T_0$ and $\mathbb{P}\,T_1$, the schema expression $S_0 \vee S_1$ has the type $\mathbb{P}(T_0 \curlyvee T_1)$.

$$\frac{t \mathrel{\dot\in} S_0}{t \in S_0 \vee S_1}\ (S^+_{\vee_{\mathrm{o}}}) \qquad \frac{t \mathrel{\dot\in} S_1}{t \in S_0 \vee S_1}\ (S^+_{\vee_{\mathrm{1}}})$$

$$\frac{t \in S_0 \vee S_1 \quad t \mathrel{\dot\in} S_0 \vdash P \quad t \mathrel{\dot\in} S_1 \vdash P}{P}\ (S^-_{\vee})$$

2.3. State schema conjunction

When the schemas S_0 and S_1 have the types $\mathbb{P}\,T_0$ and $\mathbb{P}\,T_1$, the schema expression $S_0 \wedge S_1$ has the type $\mathbb{P}(T_0 \curlyvee T_1)$.

$$\frac{t \mathrel{\dot\in} S_0 \quad t \mathrel{\dot\in} S_1}{t \in S_0 \wedge S_1}\ (S^+_{\wedge}) \qquad \frac{t \in S_0 \wedge S_1}{t \mathrel{\dot\in} S_0}\ (S^-_{\wedge_{\mathrm{o}}}) \qquad \frac{t \in S_0 \wedge S_1}{t \mathrel{\dot\in} S_1}\ (S^-_{\wedge_{\mathrm{1}}})$$

2.4. State schema inclusion

In addition our notion of atomic schemas combines with schema conjunction to provide an immediate treatment of *schema inclusion* by interpreting the separation of declarations in a schema as schema conjunction. For example, the schema $[\mathtt{z} : T;\ S \mid P]$ is just $[[\mathtt{z} : T] \wedge S \mid P]$ and so on.

2.5. State schema existential hiding

If the schemas S_0 and S_1 have the types $\mathbb{P}\, T_0$ and $\mathbb{P}\, T_1$ with $T_0 \preceq T_1$, then the type of the schema expression $\exists\, S_0 \bullet S_1$ is $\mathbb{P}(T_1 - T_0)$.

$$\frac{t \in S_1^{\mathbb{P}\, T_1} \quad t \ \dot{\in}\ S_0^{\mathbb{P}\, T_0} \quad T_0 \preceq T_1}{t \ \dot{\in}\ \exists\, S_0 \bullet S_1}\ (\exists\, S^+)$$

$$\frac{t \in \exists\, S_0 \bullet S_1 \quad x \in S_1, x \ \dot{\in}\ S_0, x \doteq t \vdash P}{P}\ (\exists\, S^-)$$

2.6. State schema preconditions

In our logic, we give a simple and comprehensive definition for the precondition of arbitrary state schema expressions rather than the usual somewhat complex syntactic account. The introduction and elimination rules are as follows:

$$\frac{z_0 \star z_1' \in S}{Pre\ S\ z_0}\ (Pre^+)$$

$$\frac{Pre\ S\ z \quad z \star y \vdash P}{P}\ (Pre^-)$$

3. The operation schema calculus

3.1. Priming and Querying

In introducing operation schemas we will, of course, need to refer to special distinguished observations corresponding to values in an after state, and to inputs.[1] Our notational conventions concerning observations are necessarily very precise and they not quite usual, so we will explain them carefully here.
Suppose that $Ob = \{\mathtt{z}_0, \mathtt{z}_1, \cdots\}$ is the set of all *before observations*. We set up the set $Ob' = \{\mathtt{z}_0', \mathtt{z}_1', \cdots\}$ of all *after observations* with the prime also denoting an obvious bijective mapping between these two sets: so that $\mathtt{z}'' = \mathtt{z}$ and so on. We will refer to $\mathtt{z}$ and $\mathtt{z}'$ as *co-observations*. The priming bijection can be extended to bindings in an obvious way: $\langle\!|\, \cdots \mathtt{z}_i \Rrightarrow v_i \cdots \,|\!\rangle' = \langle\!|\, \cdots \mathtt{z}_i' \Rrightarrow v_i \cdots \,|\!\rangle$.
We take labels such as $\mathtt{z}?$, which traditionally denote *input observations*, as certain elements of Ob, since they are before observations. Note, though, that there *are* corresponding labels $\mathtt{z}?'$, and so on, in Ob'. But we will only ever treat labels of this form as input observations: the primed versions are superfluous and will *never* be used.
This somewhat tedious attention to detail does have major benefits in the sequel, allowing us to present a much simpler programming logic than would be the case if we treated, as is more common, *priming* and *querying* as distinct forms of syntactic decoration for labels.

[1] Outputs observations (*e.g.* $\mathtt{z}!$) can also be handled by a simple extension of the methods described here; but we will not need outputs in this paper.

3.2. Preconditions and Postconditions

There are two possible approaches to preconditions: the *syntactic* approach offered by B, Morgan's refinement calculus and very many others; and the *logical* approach adopted by Z. The latter is essentially a *postcondition only* approach, in which one induces the weakest condition consistent with meeting the postcondition. If one is content to use Z for specification and design, the logical approach has much to recommend it. On the other hand, if one wishes to derive programs, there are many circumstances in which logical preconditions impose a serious burden: the task of discharging them is, in the worst case, *equivalent* to deriving a program. Whilst there may be trivial methods in some cases, often there are not. For example, in deriving a sorting algorithm, discharging the logical precondition of the specification requires a demonstration that sorted permutations always exist: but this may be *equivalent to the task of deriving a sorting algorithm.*
As we indicated in the introduction to this paper, our framework may be set up using either approach, so the partisan may choose quite freely between a system based on either logical or syntactic preconditions. The main focus in this paper will be to explore, in depth, the case in which atomic operation schemas have syntactic preconditions: the case more reminiscent of the refinement calculus. We will also indicate how the framework may be set up with the usual logical precondition approach of Z. This turns out to be as simple as changing just one clause in the semantics for operation schemas.
In this paper, then, operation schemas will have the following general structure:

Op
T

Pre

Post

We insist that *Pre*, the precondition, may only refer to observations on the input state and input observations listed amongst the schema declarations. There need be no restriction on the observations permitted in *Post*, the postcondition. A notational convention simplifies matters: if the precondition is true then that section of the schema can be omitted. One must take some care, however, because specifications written to look like standard Z will *not necessarily have equivalent syntactic and logical preconditions.* We tend to retain the new dividing line in elaborating the theory, however, for clarity of presentation.
T, in this context, must always be a schema type and always has the form $T^{in} \curlyvee T^{out'}$ where T^{in} contains declarations of all before observations and $T^{out'}$ contains declarations of all after observations. We will also need to refer to T^{out}, the co-observations of $T^{out'}$.
Of course, it would be possible to display atomic schemas of this form in a syntactic guise more reminiscent of the refinement calculus:

$$Op = w : [Pre, Post]$$

with the frame w listing those observations introduced in T above (and other fairly tedious syntactic adjustments which distinguish the Z and refinement calculus approaches to indicating before and after state observations, see *e.g.* [9]). We have decided not to do this for one particularly important reason concerning the solution the refinement calculus adopts towards the *frame problem*: our semantics deliberately approaches this in a novel way in order to better integrate refinement with the schema algebra (see especially proposition 4.2 below). We fear that the standard syntax for specification statements is so bound up with its standard semantics (with the corresponding *expand frame law*, see *e.g.* [11]) as to be potentially confusing. But of course this is, ultimately, simply a matter of syntax and notation, and the paper, or future work, could just as easily be presented in terms of specification statements.

3.3. Logic and semantics

We can now relate functions and specifications. As we promised, the interpretation of operation schemas is to be the set of functions which implement it; consequently the implementation relation is simply membership:[2]

[2] At this point, then, the implementation relation ($\in$) associates a *function* with a specification (text). Later, the implementation relation will become purely linguistic, associating a program (text) with a specification (text).

Definition 3.1.

$$f \Subset U =_{df} f \in [\![U]\!]$$

Here, U is an operation schema expression and the semantic function denoted by the brackets is defined by induction over the structure of schema expressions in the following sections. Implementations, such as f, are transformations of a *global state* $\mathbb{W}$. This schema type may vary according to the programming logic one wishes to establish and essentially determines a sufficiently large universe of observations. That is, all other (before state) schema types are taken to be subtypes under the relation $\preceq$. The precise characterisation of $\mathbb{W}$ will play no role in program development and is introduced much the same way that some sufficiently large set $\mathtt{z}$ of *locations* would be chosen to provide the denotational semantics of an imperative programming language; indeed we will see, in section 5 below, that $\mathbb{W}$ will also play this role.
Refinement of specifications, as we have previously indicated, is the subset relation on sets of implementations:

Definition 3.2.

$$U_0 \sqsupseteq U_1 =_{df} [\![U_0]\!] \subseteq_{\mathbb{W} \to \mathbb{W}} [\![U_1]\!]$$

Proposition 3.3. The following rules are derivable:

$$\frac{f \Subset U_0 \vdash f \Subset U_1}{U_0 \sqsupseteq U_1} \ (\sqsupseteq^+)$$

and:

$$\frac{U_0 \sqsupseteq U_1 \quad f \Subset U_0}{f \Subset U_1} \ (\sqsupseteq^-)$$

That is: *refinement preserves implementation.*
We shall need a conditioned version of the implements relation for situations in which we consider choice.

Definition 3.4.

$$f \Subset_C U =_{df} \exists g \Subset U \bullet g =_C f$$

We also permit the following idiom:

$$f \Subset_P U =_{df} f \Subset_{\{z | z.P\}} U$$

The rules are immediate:

Proposition 3.5. The following introduction and elimination rules for conditioned implementation are derivable.

$$\frac{g \Subset U \quad g =_C f}{f \Subset_C U} \qquad \frac{f \Subset_C U \quad y \Subset U, y =_C f \vdash P}{P}$$

3.3.1. Atomic operation schemas

We begin with the atomic operation schemas.

Definition 3.6.

$$[\![[T \mid P \mid Q]]\!] =_{df} \{ f \in \mathbb{W} \to \mathbb{W} \mid \forall z^{\mathbb{W}} \bullet z.P \Rightarrow z.(f\ z)'.Q \}$$

Note that the domain and range of the functions extend the observations made explicit in the schema. In particular, given our description of the global state $\mathbb{W}$ above, there is an assumption that $T \preceq \mathbb{W} \curlyvee \mathbb{W}'$.
Furthermore, under this definition, the implementations may effect arbitrary transformations of those values which are outside the precondition of the schema, *or even outside its alphabet* αT^{in}. This second point is crucial, and, as we indicated earlier, distinguishes our approach from those, like Morgan's refinement calculus [11], for which a stronger *frame axiom* holds. See proposition 4.2 in section 4.1 below.
Finally, note that implementations are, naturally, transformations from $\mathbb{W}$ to $\mathbb{W}$ whereas the specification, as is usual in Z notation, establishes a connection between observations in $\mathbb{W}$ and observations in $\mathbb{W}'$ (note the prime). The reader should recall that priming establishes a bijection between before and after observations, and as a consequence the term $(f\ z)'$ is a binding in $\mathbb{W}'$ as required.

Proposition 3.7. The following rules are derivable for implementation of atomic operation schemas:

$$\frac{z.P \vdash z.t'.Q}{\lambda z \bullet t \in [T \mid P \mid Q]} \ (\in^+)$$

and:

$$\frac{f \in [T \mid P \mid Q] \quad t.P}{t.(f\ t)'.Q} \ (\in^-)$$

As we indicated in the introduction, and then discussed in section 3.2, our framework can be set up using standard, single predicate, Z operation schemas. The semantics of operation schemas would modified as follows:

Definition 3.8. Let U be an atomic schema of the form $[T \mid P]$.

$$[\![U]\!] =_{df} \{f \in \mathbb{W} \to \mathbb{W} \mid \forall z^{\mathbb{W}} \bullet Pre\ U\ z \Rightarrow z \star (f\ z)' \in U\}$$

In this definition $Pre\ U\ z$ holds if z is a binding in the precondition of the schema U (see [8]). The same relaxed frame axiom applies in this interpretation. The basic rules are, however, slightly different:

Proposition 3.9. The following rules are derivable for implementation of atomic operation schemas of standard Z form:

$$\frac{Pre\ [T \mid P]\ z \vdash z \star t' \in [T \mid P]}{\lambda z \bullet t \in [T \mid P]} \ (\in^+)$$

and:

$$\frac{f \in [T \mid P] \quad Pre\ [T \mid P]\ t}{t \star (f\ t)' \in [T \mid P]} \ (\in^-)$$

As usual, we permit declarations which involve sets rather than simply types. We find it clearer, and more systematic, to use the colon exclusively for type judgements and the membership relation exclusively for set membership judgements, so our notation for these more general declarations is not standard.

Definition 3.10.

$$[x, x' \in C^{\mathbb{P}\,T} \mid P \mid Q] =_{df} [x, x' : T \mid x \in C \wedge P \mid x' \in C \wedge Q]$$

This definition, in fact, describes only a simple case of the notational device. The general version is easy to construct, though much less easy to read. The reader should have no difficulty with more general circumstances.

3.3.2. Freezing frames

In this section we introduce an operation for freezing the values of observations across an operation schema. This will be an important operation in establishing reasonable programming logics and provides a natural reasoning technique in our more permissive regime in which the strong frame expansion axiom does not hold. The schema:

$$\Psi \mathtt{x} : T \bullet U$$

is that set of implementations in U for which the value of the observation $\mathtt{x}$ does not change.

Definition 3.11.

$$[\![\Psi \mathtt{x} : T \bullet U]\!] =_{df} \{f \in [\![U]\!] \mid \forall z^{\mathbb{W}} \bullet (f\ z).\mathtt{x} = z.\mathtt{x}\}$$

Naturally, this can be generalised to arbitrary lists of observations by means of:

$$\Psi \mathtt{x}_0 : T_0, \mathtt{x}_1 : T_1 \cdots \mathtt{x}_n : T_n \bullet U =_{df} \Psi \mathtt{x}_0 : T_0 \bullet (\Psi \mathtt{x}_1 : T_1 \cdots \mathtt{x}_n : T_n \bullet U)$$

Note that we obviously have:

$$\Psi \mathtt{x} : T \bullet U \sqsupseteq U$$

But we will be especially interested in establishing conditions under which this can be strengthened to an equivalence.

3.3.3. Operation schema conjunction and disjunctive choice

The definition of conjunction is both obvious and simple.

Definition 3.12.

$$[\![U_0 \wedge U_1]\!] =_{df} [\![U_0]\!] \cap_{\mathbb{W}} [\![U_1]\!]$$

Proposition 3.13. The following rules are derivable for operation schema conjunction:

$$\frac{f \Subset U_0 \quad f \Subset U_1}{f \Subset U_0 \wedge U_1} \; (\Subset^{+}_{\wedge})$$

$$\frac{f \Subset U_0 \wedge U_1}{f \Subset U_0} \; (\Subset^{-}_{\wedge_0}) \qquad \frac{f \Subset U_0 \wedge U_1}{f \Subset U_1} \; (\Subset^{-}_{\wedge_1})$$

This definition, despite its simplicity, has rather complex properties. Naturally, it covers the case where we wish to ensure that an implementation *meets two conditions*. However that is ambiguous: there is a conjunctive interpretation when those conditions have to be met over a *common domain*, that is, when the preconditions of the component schemas are consistent (they overlap as sets of states); there is also a disjunctive interpretation when the preconditions are contradictory (distinct sets of states): such implementations are choices, or conditionals.
We will see this more clearly below in section 4.2 when we see the range of refinement inequations which follow from this definition. For notational clarity we will write the connective as a disjunction in those circumstances in which the preconditions of the component schemas are contradictory.

3.3.4. Operation schema existential quantification

We give the semantics for hiding two labels related by priming.

Definition 3.14.

$$[\![\exists \mathtt{z}, \mathtt{z}' : C \bullet U]\!] =_{df} \{ f \mid \exists\, g \in [\![U]\!] \bullet f = \lambda\, \sigma \bullet (g\sigma)[\mathtt{z}/\sigma.\mathtt{z}] \}$$

The definition ensures that the meaning is the set of implementations of U which ensure that the value of the observation $\mathtt{z}$ is left unchanged. This means that its unprimed and primed values remain the same, which is enforced by the substitution $[\mathtt{z}/\sigma.\mathtt{z}]$.

Proposition 3.15. The following rules are derivable:

$$\frac{g \Subset U \quad f = \lambda\sigma \bullet (g\ \sigma)[\mathtt{z}/\sigma.\mathtt{z}]}{f \Subset \exists \mathtt{z}, \mathtt{z}' : C \bullet U} \; (\Subset^{+}_{\exists})$$

$$\frac{f \Subset \exists \mathtt{z}, \mathtt{z}' : C \bullet U \quad y \Subset U, f = \lambda\sigma \bullet (y\sigma)[\mathtt{z}/\sigma.\mathtt{z}] \vdash P}{P} \; (\Subset^{-}_{\exists})$$

By analogy with ordinary quantification, which introduces the terms *bound* and *free* with respect to variables, we introduce the terms *hidden* and *visible* for this form of quantification with respect to observations. These terms can be extended to derivations, so that we will say that occurrences of the observations $\mathtt{x}$ and $\mathtt{x}'$ are hidden in a derivation whose conclusion is $\exists \mathtt{x}, \mathtt{x}' \bullet U$ (otherwise they are visible).

3.3.5. Operation schema composition

Definition 3.16.

$$[\![U_0 \fatsemi U_1]\!] =_{df} \{ f \mid \exists f_0 \in [\![U_0]\!] \bullet \exists f_1 \in [\![U_1]\!] \bullet f = f_0 \circ f_1 \}$$

Proposition 3.17. The following rules are derivable:

$$\frac{f_0 \Subset U_0 \quad f_1 \Subset U_1}{f_0 \circ f_1 \Subset U_0 \fatsemi U_1} \ (\Subset \fatsemi^{+})$$

$$\frac{f_0 \circ f_1 \Subset U_0 \fatsemi U_1 \quad y_0 \Subset U_0, y_1 \Subset U_1, f = y_0 \circ y_1 \vdash P}{P} \ (\Subset \fatsemi^{-})$$

3.3.6. Operation schema abstraction and application

We need to introduce notions of *schema abstraction* and *application* in order to deal with program development using procedure definitions and procedure calls. In order to provide a semantics for these new objects we first introduce a mechanism for *currying* and *uncurrying* functions over the global state $\mathbb{W}$. This requires a generalisation of the standard concepts from cartesian products to schema types.
First of all we fix $[\mathtt{z} : T]$ to be some subtype of $\mathbb{W}$. This may be expressed by the following equation:

$$\mathbb{W} = \mathbb{W}^{-} \upharpoonright [\mathtt{z} : T]$$

For notational convenience we use σ to range over $\mathbb{W}$.

Definition 3.18. Let $f \in \mathbb{W} \to \mathbb{W}$. We define

$$curry_{[\mathtt{z:T}]}\ f \in T \to \mathbb{W} \to \mathbb{W}$$

by means of

$$curry_{[\mathtt{z:T}]}\ f\ t^{T}\ \sigma\ =_{df} f\ \sigma[\mathtt{z}/t]$$

This enables us to curry operation schemas.[3]

Definition 3.19. Let $C \in \mathbb{P}(\mathbb{W} \to \mathbb{W})$.

$$curry_{[\mathtt{z:T}]}\ C =_{df} \{curry_{[\mathtt{z:T}]}\ f \mid f \in C\}$$

We are now in a position to provide the semantics for schema abstractions:

Definition 3.20.

$$[\![\lambda \mathtt{z} : T \bullet U]\!] =_{df} curry_{[\mathtt{z:T}]}\ [\![U]\!]$$

Note that objects of the form $\lambda \mathtt{z} : T \bullet U$ are not operation schemas (they have the wrong type). However, we may define a new form of operation schema by applying schema abstractions to appropriate arguments.

Definition 3.21. For any schema abstraction η and term t of appropriate types:

$$[\![\eta[t]]\!] =_{df} \{\lambda\sigma \bullet f\ (\sigma.t)\ \sigma \mid f \in [\![\eta]\!]\}$$

Finally we require the inverse operation of uncurrying.

Definition 3.22. Let $f \in T \to \mathbb{W} \to \mathbb{W}$. We define:

$$uncurry_{[\mathtt{z:T}]} f \in \mathbb{W} \to \mathbb{W}$$

by means of

$$uncurry_{[\mathtt{z:T}]} f\ \sigma\ = f\ \sigma.\mathtt{z}\ \sigma$$

We are able to prove a beta-like equation for our lambda abstraction and application over operation schemas. First we need to introduce a syntactic notion of substitution for atomic operation schemas.

[3] We use standard informal set definitions here (complex expressions where the bound variable should be used) as they are easier to read than the strictly formal notation which requires use of the existential quantifier. In the proofs based on these definitions, the reader will see the strictly formal versions.

Definition 3.23. Let t be a term such that αt and αD are disjoint, and let D_0 be the declaration corresponding to the alphabet of observations αt.

$$[D \mid P \mid Q][\mathbf{z}/t] = [D;\ D_0 \mid P[\mathbf{z}/t] \mid Q[\mathbf{z}/t]]$$

With this in place we can move on to beta-reduction.

Proposition 3.24. Let $U =_{df} [D \mid P \mid Q]$.

$$(\lambda \mathbf{z} \bullet U)[t] = U[\mathbf{z}/t]$$

Proof.

$$\begin{array}{ll}
[\![(\lambda \mathbf{z} \bullet U)[t]]\!] & = \\
\{g \mid \exists f \bullet g = \lambda\sigma \bullet f\ (\sigma.t)\ \sigma \wedge f \in [\![\lambda \mathbf{z} \bullet U]\!]\} & = \\
\{g \mid \exists f \bullet g = \lambda\sigma \bullet f\ (\sigma.t)\ \sigma \wedge \exists\, h \bullet f = curry_{\mathbf{z}} h \wedge h \in [\![U]\!]\} & = \\
\{g \mid \exists f, h \bullet g = \lambda\sigma \bullet f\ (\sigma.t)\ \sigma \wedge f = curry_{\mathbf{z}} h \wedge h \in [\![U]\!]\} & = \\
\{g \mid \exists\, h \bullet g = \lambda\sigma \bullet curry_{\mathbf{z}} h\ (\sigma.t)\ \sigma \wedge h \in [\![U]\!]\} & = \\
\{g \mid \exists\, h \bullet g = \lambda\sigma \bullet h\ \sigma[\mathbf{z}/\sigma.t] \wedge h \in \{f \mid \forall \sigma \bullet \sigma.P \Rightarrow \sigma.(f\ \sigma)'.Q\}\} & = \\
\{g \mid \exists\, h \bullet g = \lambda\sigma \bullet h\ \sigma[\mathbf{z}/\sigma.t] \wedge \forall \sigma \bullet \sigma.P \Rightarrow \sigma.(h\ \sigma)'.Q\} & = \\
\{g \mid \forall \sigma \bullet \sigma.P[\mathbf{z}/t] \Rightarrow \sigma.(g\ \sigma)'.Q[\mathbf{z}/t]\} & =
\end{array}$$

The last of these steps makes use, in one direction, of the following:

$$\begin{array}{lll}
\forall \sigma \bullet \sigma.P \Rightarrow \sigma.(h\ \sigma)'.Q & \Rightarrow & \\
\sigma[\mathbf{z}/\sigma.t].P \Rightarrow \sigma[\mathbf{z}/\sigma.t].(h\ \sigma[\mathbf{z}/\sigma.t])'.Q & \Leftrightarrow & \text{(lemma A.1)} \\
\sigma.P[\mathbf{z}/t] \Rightarrow \sigma.(h\ \sigma[\mathbf{z}/\sigma.t])'.Q[\mathbf{z}/t] & \Leftrightarrow & \\
\forall \sigma \bullet \sigma.P[\mathbf{z}/t] \Rightarrow \sigma.(h\ \sigma[\mathbf{z}/\sigma.t])'.Q[\mathbf{z}/t] & &
\end{array}$$

Definition 3.23 can easily generalised to arbitrary schema expressions by recursion over their structure. Then we have the following.

Proposition 3.25.

$$(\lambda \mathbf{z} \bullet U)[t] = U[\mathbf{z}/t]$$

Proof. By induction over the structure of U with proposition 3.24 supplying the base case. We will refer to this equation, when used left to right, as *β-reduction*.

We will drop the target observation in substitutions, and the types in schema abstractions, when these are clear from the context.

4. Refinement Logic

4.1. Basic inequations

The following inequations for refinement are derivable:

Proposition 4.1. Weakening preconditions:
(i)

$$\frac{z^{T^{in}}.P_1 \vdash z.P_0}{[T \mid P_0 \mid P] \sqsupseteq [T \mid P_1 \mid P]}\ (\sqsupseteq^{+}_{pre})$$

Strengthening postconditions:
(ii)

$$\frac{z^{T}.P_0 \vdash z.P_1}{[T \mid P \mid P_0] \sqsupseteq [T \mid P \mid P_1]}\ (\sqsupseteq^{+}_{post})$$

Additionally, we have the following *expand frame* rule:

Proposition 4.2.

$$\frac{}{[T_0;\ T_1 \mid P \mid Q] = [T_0 \mid P \mid Q]}\ (exp)$$

Note that these schemas are *equal*, that is they are the same set of implementations. This is very different from the situation in, for example, [11] in which implementations may not change the values of observations outside the frame. Our law indicates that a program which implements a specification is unconstrained both outside the specification's precondition *and* outside its frame. This is much more convenient when specifications may be constructed using schema operations such as conjunction in which the frames do not necessarily coincide, as we explained above in the introduction and in sections 3.2 and 3.3.1 above.
Naturally there are consequences of our more relaxed regime, since stability outside the alphabet (or frame) cannot be assumed. This is where the frame freezing operation becomes important, so important in fact that we introduce a constraint on programming logics built on our framework:

Principle 4.3. All programming logics should satisfy the *frame freezing principle*: Whenever δ is a derivation with conclusion $p \Subset U$ and $\mathtt{x}'$ an observation such that $\mathtt{x}'$ is not visible in δ, then $p \Subset \Psi\mathtt{x} : T \bullet U$.

4.2. Conjunction and choice logic

Our first inequation expresses the conjunctive behaviour of the connective.

Proposition 4.4.

$$[T_0 \mid P_0 \mid Q_0] \wedge [T_1 \mid P_1 \mid Q_1] \sqsupseteq [T_0 \upharpoonright T_1 \mid P_0 \wedge P_1 \mid Q_0 \wedge Q_1]$$

Proof. We treat the equation as:

$$U_0 \wedge U_1 \sqsupseteq U$$

Then:

$$\dfrac{\dfrac{\dfrac{f \Subset U_0 \wedge U_1}{f \Subset U_0} \quad \dfrac{\overline{}\ 1}{z.P_0 \wedge z.P_1}}{z.P_0}}{z.(f\ z)'.Q_0} \quad \dfrac{\dfrac{\dfrac{f \Subset U_0 \wedge U_1}{f \Subset U_1} \quad \dfrac{\overline{}\ 1}{z.P_0 \wedge z.P_1}}{z.P_1}}{z.(f\ z)'.Q_1}}{\dfrac{z.(f\ z)'.(Q_0 \wedge Q_1)}{f \Subset U}\ 1}$$

In this derivation, and those that now follow, we have omitted steps that involve merely rewriting by substitution, for example, steps using the identity $z.(P_0 \wedge P_1) = z.P_0 \wedge z.P_1$.

Proposition 4.5.

$$[T_0 \upharpoonright T_1 \mid P_0 \vee P_1 \mid Q_0 \wedge Q_1] \sqsupseteq [T_0 \mid P_0 \mid Q_0] \wedge [T_1 \mid P_1 \mid Q_1]$$

Proof. We treat the equation as:

$$U \sqsupseteq U_0 \wedge U_1$$

Then:

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{}}{f \Subset U} \quad \dfrac{\overline{}\ 1}{z.(P_0 \vee P_1)}}{z.(f\ z)'.Q_0 \wedge z.(f\ z)'.Q_1}}{z.(f\ z)'.Q_0}}{f \Subset U_0}\ 1 \quad \dfrac{\dfrac{\dfrac{\dfrac{\overline{}}{f \Subset U} \quad \dfrac{\overline{}\ 2}{z.(P_0 \vee P_1)}}{z.(f\ z)'.Q_0 \wedge z.(f\ z)'.Q_1}}{z.(f\ z)'.Q_1}}{f \Subset U_1}\ 2}{f \Subset U_0 \wedge U_1}$$

The final inequation expresses the disjunctive choice inherent in the connective.

Proposition 4.6.

$$[T_0 \mid P_0 \mid Q_0] \wedge [T_1 \mid P_1 \mid Q_1] \sqsupseteq [T_0 \curlyvee T_1 \mid P_0 \vee P_1 \mid Q_0 \vee Q_1]$$

Proof. We treat the equation as:

$$U_0 \wedge U_1 \sqsupseteq U$$

Then:

$$\dfrac{\dfrac{\dfrac{f \notin U_0 \wedge U_1}{f \notin U_0}\quad\dfrac{}{z.P_0}\ (2)}{z.(f\ z)'.Q_0}\quad\dfrac{\dfrac{f \notin U_0 \wedge U_1}{f \notin U_1}\quad\dfrac{}{z.P_1}\ (2)}{z.(f\ z)'.Q_1}}{\dfrac{\dfrac{}{z.P_0 \vee z.P_1}\ (1)\qquad z.(f\ z)'.(Q_0 \vee Q_1)\qquad z.(f\ z)'.(Q_0 \vee Q_1)}{z.(f\ z)'.(Q_0 \vee Q_1)}\ (2)}{f \notin U}\ (2)$$

Conjunction is monotonic with respect to refinement.

Proposition 4.7.

$$\frac{U_0 \sqsupseteq U_1}{U_0 \wedge U \sqsupseteq U_1 \wedge U}$$

4.3. Composition logic

For simplicity of presentation we consider a special case. This is easily generalised.

Proposition 4.8.

$$[\mathtt{x}, \mathtt{x}' : T \mid P_0 \mid Q_0] \fatsemi [\mathtt{x}, \mathtt{x}' : T \mid P_1 \mid Q_1] \sqsupseteq$$
$$[\mathtt{x}, \mathtt{x}' : T \mid P_0 \wedge \forall\, u : T \bullet Q_0[\mathtt{x}'/u] \Rightarrow P_1[\mathtt{x}/u] \mid \exists\, v : T \bullet Q_0[\mathtt{x}'/v] \wedge Q_1[\mathtt{x}/v]]$$

Proof. Treat the equation as:

$$U_0 \fatsemi U_1 \sqsupseteq U$$

Then:

$$\dfrac{f \notin U_0 \fatsemi U_1 \qquad \dfrac{\dfrac{f = y_0 \fatsemi y_1}{f \notin U}\ 1 \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{y_1 \notin U_1}\ 1 \quad \dfrac{\vdots\ \delta_1}{(y_0\ z).P_1}}{(y_0\ z).(y_1\ (y_0\ z))'.Q_1}\ \delta_0}{z.(y_0\ z)'.Q_0 \wedge (y_0\ z).(y_1\ (y_0\ z))'.Q_1}}{z.(y_1\ (y_0\ z))'.\exists\, u : T \bullet Q_0[\mathtt{x}'/u] \wedge Q_1[\mathtt{x}/u]}}{y_0 \fatsemi y_1 \notin U}\ 2}{f \notin U}\ 1}{f \notin U}\ 1$$

where δ_0 is:

$$\dfrac{\dfrac{}{y_0 \notin U_0}\ 1 \qquad \dfrac{\dfrac{}{z.P_0 \wedge z.\forall\, u : T \bullet Q_0[\mathtt{x}'/u] \Rightarrow P_1[\mathtt{x}/u]}\ 2}{z.P_0}}{z.(y_0\ z)'.Q_0}$$

and δ_1 is:

$$\dfrac{\dfrac{}{z.P_0 \wedge z.\forall\, u : T \bullet Q_0[\mathtt{x}'/u] \Rightarrow P_1[\mathtt{x}/u]}\ 2}{z.(y_0\ z)'.Q_0 \Rightarrow (y_0\ z).P_1}$$

Composition is monotonic with respect to refinement:

Proposition 4.9.

$$\frac{U_0 \sqsupseteq U_2 \quad U_1 \sqsupseteq U_3}{U_0 \fatsemi U_1 \sqsupseteq U_2 \fatsemi U_3}$$

4.4. Existential hiding logic

Here we state and prove two refinement rules involving hiding.

Proposition 4.10.

$$\exists\, \mathtt{x}, \mathtt{x}' : T \bullet [\mathtt{x}, \mathtt{x}' : T;\ D \mid P \mid Q] \sqsupseteq$$
$$[D \mid \exists\, u : T \bullet P[\mathtt{x}/u] \mid \exists\, u, v : T \bullet Q[\mathtt{x}, \mathtt{x}'/u, v]]$$

Proof.

$$\frac{\dfrac{}{\exists\, g \in U_0 \bullet f = \lambda z \bullet (g\ z)[\mathtt{x}/z.\mathtt{x}]}\ def \quad \dfrac{\begin{array}{c}\delta \\ \vdots \\ z.(f\ z)'.\ \exists\, u, v : T \bullet Q[\mathtt{x}, \mathtt{x}'/u, v]\end{array}}{f \in U_1}\ 2}{f \in U_1}\ 1$$

where δ, putting $t = \lambda z \bullet (y\ z)[\mathtt{x}/z.\mathtt{x}]$, is:

$$\frac{\dfrac{}{f = t}\ 1 \quad \dfrac{\dfrac{}{z.\exists\, u : T \bullet P[\mathtt{x}/u]}\ 2 \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{y \in U_0}\ 1 \quad \dfrac{}{z.P[\mathtt{x}/w]}\ 3}{z.(y\ z)'.Q[\mathtt{x}/w]}}{z.(y\ z)'.\exists\, u, v : T \bullet Q[\mathtt{x}/w][w, \mathtt{x}'/u, v]}}{z.(y\ z)'.\exists\, u, v : T \bullet Q[\mathtt{x}, \mathtt{x}'/u, v]}}{z.(y\ z)'[\mathtt{x}'/z.\mathtt{x}].\exists\, u, v : T \bullet Q[\mathtt{x}, \mathtt{x}'/u, v]}}{z.(t\ z)'.\exists\, u, v : T \bullet Q[\mathtt{x}, \mathtt{x}'/u, v]}\ 3}{z.(f\ z)'.\exists\, u, v : T \bullet Q[\mathtt{x}, \mathtt{x}'/u, v]}$$

Proposition 4.11.

$$\exists\, \mathtt{x}, \mathtt{x}' : T \bullet [\mathtt{x}, \mathtt{x}' : T;\ D \mid P \mid Q] \sqsupseteq$$
$$[D \mid \forall\, u : T \bullet P[\mathtt{x}/u] \mid \exists\, u, v : T \bullet Q[\mathtt{x}, \mathtt{x}'/u, v]]$$

Proof.

$$\frac{\dfrac{}{\exists\, g \in U_0 \bullet f = \lambda z \bullet (g\ z)[\mathtt{x}/z.\mathtt{x}]}\ def \quad \dfrac{\begin{array}{c}\delta \\ \vdots \\ z.(f\ z)'.\ \exists\, u, v : T \bullet Q[\mathtt{x}, \mathtt{x}'/u, v]\end{array}}{f \in U_1}\ 2}{f \in U_1}\ 1$$

where δ is:

$$\dfrac{\dfrac{\dfrac{}{y \in U_0}\,1 \quad \dfrac{\dfrac{}{z.\forall\, u : T \bullet P[\mathtt{x}/u]}\,2}{z.P}}{z.(y\ z)'.Q} \quad \dfrac{}{f = \lambda z \bullet (y\ z)[\mathtt{x}/z.\mathtt{x}]} \quad \dfrac{}{z.((y\ z)[\mathtt{x}/z.\mathtt{x}])'.\exists\, u, v : T \bullet Q[\mathtt{x}, \mathtt{x}'/u, v]}}{z.(f\ z)'.\exists\, u, v : T \bullet Q[\mathtt{x}, \mathtt{x}'/u, v]}\,1$$

Existential quantification is monotonic with respect to refinement:

Proposition 4.12.

$$\dfrac{U_0 \sqsupseteq U_1}{\exists\, \mathtt{z} : T \bullet U_0 \sqsupseteq \exists\, \mathtt{z} : T \bullet U_1}$$

Our final rule allows the introduction of hidden state. It is analogous to a rule for introducing a local variable in refinement calculus (*e.g.* [11]).

Proposition 4.13. The following rule is derivable. Let y and y' be fresh observations:

$$\dfrac{\exists\, u, v : T \bullet Q_1}{\exists\, \mathtt{y}, \mathtt{y}' : T \bullet [D;\ \mathtt{y}, \mathtt{y}' : T \mid P \mid Q_0 \wedge Q_1[u, v/\mathtt{y}, \mathtt{y}']] \sqsupseteq [D \mid P \mid Q_0]}$$

Proof. Since, by the premise, we know that $\exists\, u, v : T \bullet Q_1$ holds, we may infer that:

$$[D \mid P \mid Q_0] = [D \mid P \mid Q_0 \wedge \exists\, u, v : T \bullet Q_1]$$

We may move the quantifier without loss of generality giving us:

$$[D \mid P \mid Q_0] = [D \mid P \mid \exists\, u, v : T \bullet Q_0 \wedge Q_1]$$

Then we use 4.11 to obtain:

$$\exists\, \mathtt{y}, \mathtt{y}' : T \bullet [D;\ \mathtt{y}, \mathtt{y}' : T \mid P \mid Q_0 \wedge Q_1[u, v/\mathtt{y}, \mathtt{y}']]$$

as required. Note that y is fresh, so does not occur in P in particular, and so:

$$P \Leftrightarrow \forall\, u : T \bullet P[\mathtt{y}/u]$$

in this case.

5. An Example Programming logic

Given our logic for specification implementation above, we can, as promised, now specialize this for some target programming language. We choose a typical imperative language in what follows.

5.1. The programming language $\mathcal{P}_{\mathbb{N}}$

$\mathcal{P}_{\mathbb{N}}$ is a very simple language for computing over the natural numbers. As a consequence of our mathematical treatment of schema types, and our insistence that observations in Z are constants, we can model the variables of our programming language directly as (unprimed) observations.
A command will correspond directly to an implementation of a specification, and so its meaning will be a transformation of the global state $\mathbb{W}$. We will reserve the symbol σ (with, if necessary, diacritical additions) for bindings that range over the global state. The semantics of programs, then, amounts to a translation into the term language of $\mathcal{Z}_{\mathcal{C}}$. For technical reasons, which will be evident when we consider the semantics of recursive procedures, we will permit $\mathcal{Z}_{\mathcal{C}}$ variables of appropriate type (for example the type $\mathbb{W} \to \mathbb{W}$ that represents commands) to appear in the programming language. As might be expected, these variables translate to themselves under the semantic function, for example: $[\![v^{\mathbb{W} \to \mathbb{W}}]\!] = v$.

Our simplest command is `skip`, which as usual leaves the state unchanged:

$$[\![\texttt{skip}]\!]\,\sigma =_{df} \sigma$$

Next we have simultaneous assignment:

$$[\![\cdots \mathtt{x}_i \cdots \texttt{:=}\ \cdots exp_i \cdots]\!]\,\sigma =_{df} \sigma[\cdots \mathtt{x}_i / [\![exp_i]\!]\,\sigma \cdots]$$

Command sequencing is obviously composition of state transformations:

$$[\![cmd_0\texttt{;}\ cmd_1]\!] =_{df} [\![cmd_1]\!] \circ [\![cmd_0]\!]$$

Blocks introduce local hidden state:

$$[\![\texttt{begin var x;}\ cmd\ \texttt{end}]\!]\,\sigma =_{df} ([\![cmd]\!]\,\sigma)[\mathtt{x}/\sigma.\mathtt{x}]$$

We permit a conditional command:

$$[\![\texttt{if}\ exp\ \texttt{then}\ cmd_0\ \texttt{else}\ cmd_1]\!]\,\sigma =_{df} elim_{\mathbb{B}}([\![exp]\!]\,\sigma)([\![cmd_0]\!]\,\sigma)([\![cmd_1]\!]\,\sigma)$$

Simple procedures are quite straightforward because our model of the state combines both ordinary variables and input parameters.

$$[\![\texttt{proc}\ p[\mathtt{z}]\ cmd]\!] =_{df} [\![cmd]\!]$$

For simple procedures we have the following straightforward result.

Lemma 5.1.

$$[\![p[exp]]\!] = [\![cmd[\mathtt{z}/exp]]\!]$$

Note that the syntactic substitution replaces programming language variables by expressions (right-values). Since the programming language contains an assignment statement we should point out that though this notion of substitution is otherwise obvious, we do not, of course, replace left-value instances of `z` by the expression.
We also wish to provide simple primitive recursive procedures.

$$[\![\texttt{proc}\ p[\mathtt{z}]\ \texttt{cases z in}\ 0 : cmd_0\ \mid\ \mathtt{m}+1 : cmd_1\ \texttt{endcases}]\!] =_{df}$$

$$uncurry_{\mathtt{z}}(elim_{\mathbb{N}}\ [\![cmd_0]\!]\ (\lambda n \bullet \lambda v \bullet [\![cmd_1[\mathtt{m}/n][p[\mathtt{m}]/v]]\!]))$$

And finally procedure calls:

$$[\![p[exp]]\!]\,\sigma =_{df} curry_{\mathtt{z}}\,[\![p]\!]\ ([\![exp]\!]\,\sigma)\ \sigma$$

The expressions of $\mathcal{P}_{\mathbb{N}}$ are given as follows:

$$\begin{array}{rcl} exp & ::= & \mathsf{True} \mid \mathsf{False} \mid exp = exp\ \cdots\ etc. \\ & & \mid\ 0 \mid 1 \mid 2\ \cdots\ etc. \mid exp + exp\ \cdots\ etc. \\ & & \mid\ \mathtt{x} \mid \mathtt{y} \mid \mathtt{z}\ \cdots\ etc. \end{array}$$

The semantics of expressions (their value in a state) is very straightforward since the syntax is arranged to pick out a subset of the syntax of the language of predicates that may appear in a Z schema.

$$[\![exp]\!]\,\sigma =_{df} \sigma.exp$$

Having now completed the semantics for the programming language we can finally make good the semantics for the implementation relation:

$$[\![cmd \Subset U]\!] =_{df} [\![cmd]\!] \in [\![U]\!]$$

It is now possible to introduce a number of rules for program development specifically for $\mathcal{P}_{\mathbb{N}}$.

5.2. Skip

Trivially, we have rules for the `skip` command.

Proposition 5.2. The following rules are derivable for the skip command:

$$\frac{\texttt{skip} \Subset [D \mid P \mid Q] \quad \sigma.P}{\sigma.\sigma'.Q}\ (\Subset^{-}_{\texttt{skip}}) \qquad \frac{\sigma.P \vdash \sigma.\sigma'.Q}{\texttt{skip} \Subset [D \mid P \mid Q]}\ (\Subset^{+}_{\texttt{skip}})$$

Proof.

$$\frac{\dfrac{}{\texttt{skip}\ \sigma = \sigma} \quad \dfrac{\texttt{skip} \Subset U \quad \sigma.P}{\sigma.(\texttt{skip}\ \sigma)'.Q}}{\sigma.\sigma'.Q}$$

and:

$$\frac{\dfrac{}{\texttt{skip}\ \sigma = \sigma} \quad \begin{array}{c}\overline{\sigma.P}\\ \vdots \\ \sigma.\sigma'.Q\end{array}}{\dfrac{\sigma.(\texttt{skip}\ \sigma)'.Q}{\texttt{skip} \Subset U}}$$

Recall that σ' is *not* a variable with a diacritical prime, but a term: the variable σ subject to the priming operation. Thus σ and σ' are the *same* binding (global state) modulo the priming of their observations. We can also show that `skip` acts as a left and right identity for composition, for example:

$$\frac{cmd\ \texttt{;}\ \texttt{skip} \Subset U}{cmd \Subset U}\ (\Subset^{id-}_{\texttt{skip}})$$

5.3. Assignment

Proposition 5.3. The following rule (in which no after state identifier may occur in the expressions exp_i) is derivable for simultaneous assignment to the variables x_i:

$$\frac{\sigma.P \vdash \sigma.\sigma[\cdots \texttt{x}_i/\sigma.exp_i \cdots]'.Q}{\cdots \texttt{x}_i \cdots\ \texttt{:=}\ \cdots exp_i \cdots \Subset [\cdots \texttt{x}_i, \texttt{x}'_i \cdots : \mathbb{N}\ ;\ D \mid P \mid Q]}\ (\Subset^{+}_{\texttt{:=}})$$

5.4. Conditional

Proposition 5.4.

$$\begin{array}{lll}(i) & \texttt{if}\ exp\ \texttt{then}\ cmd_0\ \texttt{else}\ cmd_1 & =_{exp} \quad cmd_0\\ (ii) & \texttt{if}\ exp\ \texttt{then}\ cmd_0\ \texttt{else}\ cmd_1 & =_{\neg exp} \quad cmd_1\end{array}$$

An introduction rule for conditional commands:

Proposition 5.5.

$$\frac{cmd_0 \Subset_{exp} U_0 \quad cmd_1 \Subset_{\neg exp} U_1}{\texttt{if}\ exp\ \texttt{then}\ cmd_0\ \texttt{else}\ cmd_1 \Subset U_0 \wedge U_1}\ (\Subset^{+}_{if})$$

Proof. We will write c for `if` exp `then` cmd_0 `else` cmd_1, c_0 for cmd_0, c_1 for cmd_1 and e for exp.

$$\frac{\dfrac{cmd_0 \Subset_{exp} U_0 \quad \overline{c =_{exp} cmd_0}\ (5{\cdot}4)(i)}{c \Subset U_0} \qquad \dfrac{cmd_1 \Subset_{\neg exp} U_1 \quad \overline{c =_{\neg exp} cmd_1}\ (5{\cdot}4)(ii)}{c \Subset U_1}}{c \Subset U_0 \wedge U_1}$$

5.5. Sequencing

Given the semantics of sequencing in the programming language and that of composition of specifications in the specification language we obtain the following directly:

Proposition 5.6.

$$\frac{cmd_0 \Subset U_0 \quad cmd_1 \Subset U_1}{cmd_0 \ ; \ cmd_1 \Subset U_0 \fatsemi U_1} \ (\Subset^{+}_{;})$$

5.6. Blocks

The implementation of existentially quantified operation schemas, concerned as they are with hiding observations, will be implemented by the block or the introduction of local program variables which, on leaving the block, are returned to their previous denotations. We have, for each programming language variable (Z observation) $\mathtt{z}$:

Proposition 5.7.

$$\frac{cmd \Subset U}{\mathtt{begin\ var\ z\ ;}\ cmd\ \mathtt{end} \Subset \exists \mathtt{z}, \mathtt{z}' : \mathbb{N} \bullet U} \ (\Subset^{+}_{block})$$

5.7. Procedures

5.7.1. Non-recursive procedures

In this section we only deal with the simpler non-recursive case, beginning with an introduction rule.

Proposition 5.8. Let $\mathtt{n}$ be a fresh programming language variable. Then:

$$\frac{p[\mathtt{n}] \Subset (\lambda \mathtt{z} : \mathbb{N} \bullet U)[\mathtt{n}]}{p \Subset U}$$

Proof. In the proof we omit the typing information (for $\mathtt{z}$ and σ and so on) since it is clear from the context and simplifies the presentation. Since p is a simple non-recursive procedure we assume it has the following form: $\mathtt{proc}\ p[\mathtt{z}]\ cmd$. Note that $[\![p[\mathtt{n}]]\!] = \lambda \sigma \bullet curry_{\mathtt{z}}\, [\![p]\!]\, ([\![\mathtt{n}]\!]\, \sigma)\ \sigma = \lambda \sigma \bullet curry_{\mathtt{z}}\, [\![p]\!]\, (\sigma.\mathtt{n})\ \sigma$, and that $[\![(\lambda \mathtt{z} \bullet U)[\mathtt{n}]]\!] = \{\lambda \sigma \bullet f\ (\sigma.\mathtt{n})\ \sigma \mid f \in [\![\lambda \mathtt{z} \bullet U]\!]\}$ or more formally $\{g \mid \exists f \bullet g = \lambda \sigma \bullet f\ (\sigma.\mathtt{n})\ \sigma \wedge f \in [\![\lambda \mathtt{z} \bullet U]\!]\}$. So, from the premise, which amounts to $[\![p[\mathtt{n}]]\!] \in [\![(\lambda \mathtt{z} \bullet U)[\mathtt{n}]]\!]$ we can infer that: $\exists f \bullet \lambda \sigma \bullet curry_{\mathtt{z}}\, [\![p]\!]\, (\sigma.\mathtt{n})\ \sigma = \lambda \sigma \bullet f\ (\sigma.\mathtt{n})\ \sigma \wedge f \in [\![\lambda \mathtt{z} \bullet U]\!]$. So, for some arbitrary y_0 we have:

$$\lambda \sigma \bullet curry_{\mathtt{z}}\, [\![p]\!]\, (\sigma.\mathtt{n})\ \sigma = \lambda \sigma \bullet y_0\ (\sigma.\mathtt{n})\ \sigma \wedge y_0 \in [\![\lambda \mathtt{z} \bullet U]\!] \qquad (1)$$

¿From the second conjunct of (1) we have: $y_0 \in curry_{\mathtt{z}}\, [\![U]\!]$ which is $y_0 \in \{curry_{\mathtt{z}}\ f \mid f \in [\![U]\!]\}$ or more formally $y_0 \in \{h \mid \exists f \bullet h = curry_{\mathtt{z}}\ f \wedge f \in [\![U]\!]\}$ from which we have $\exists f \bullet y_0 = curry_{\mathtt{z}}\ f \wedge f \in [\![U]\!]$. So, for some arbitrary y_1 we have:

$$y_0 = curry_{\mathtt{z}}\ y_1 \wedge y_1 \in [\![U]\!] \qquad (2)$$

Substituting the first conjunct of (2) into the first conjunct of (1) we obtain: $\lambda \sigma \bullet curry_{\mathtt{z}}\, [\![p]\!]\, (\sigma.\mathtt{n})\ \sigma = \lambda \sigma \bullet curry_{\mathtt{z}}\ y_1\ (\sigma.\mathtt{n})\ \sigma$. By extensionality, and the fact that $curry_{\mathtt{z}}$ is injective, we conclude that $y_1 = [\![p]\!]$. Substituting this into the second conjunct of (2) we obtain $[\![p]\!] \in [\![U]\!]$ which is, by definition, $p \Subset U$ as required.

Corollary 5.9.

$$\frac{cmd[\mathtt{z}/\mathtt{n}] \Subset U[\mathtt{z}/\mathtt{n}]}{p \Subset U} \ (proc^{+})$$

Proof. A trivial consequence of proposition 5.8 by lemma 5.1 and proposition 3.25.

Turning now to the elimination rule.

Proposition 5.10. The following rule is derivable:

$$\frac{p \Subset U}{p[exp] \Subset (\lambda \mathtt{z} \bullet U)[exp]}$$

Proof. We have to establish that:

$$[\![p[exp]]\!] \in [\![(\lambda \mathtt{z} \bullet U)[exp]]\!]$$

which amounts to:

$$\exists f \bullet \lambda \sigma \bullet curry_{\mathtt{z}}\, [\![p]\!]\, (\sigma.exp)\ \sigma = \lambda \sigma \bullet f\ (\sigma.exp)\ \sigma \wedge f \in [\![\lambda \mathtt{z} \bullet U]\!]$$

This follows if:

$$curry_{\mathtt{z}}\, [\![p]\!] \in [\![\lambda \mathtt{z} \bullet U]\!]$$

which amounts to:

$$\exists f \bullet curry_{\mathtt{z}}\, [\![p]\!] = curry_{\mathtt{z}}\ f \wedge f \in [\![U]\!]$$

and this follows providing that:

$$[\![p]\!] \in [\![U]\!]$$

which is the premise:

$$p \Subset U$$

Again, in view of lemma 5.1 and proposition 3.25, we have a corollary.

Corollary 5.11.

$$\frac{p \Subset U}{cmd[\mathtt{z}/exp] \Subset U[\mathtt{z}/exp]}\ (proc^{-})$$

5.7.2. Recursive procedures

Now the more complex case: procedures which are primitive recursive. We can still prove the introduction rule for this form of procedure.

Proposition 5.12. Let $\mathtt{n}$ be any programming language variable. Then:

$$\frac{p[\mathtt{n}] \Subset (\lambda \mathtt{z} \bullet U)[\mathtt{n}]}{p \Subset U}$$

Proof. Note that:

$$[\![p[\mathtt{n}]]\!] = \lambda \sigma \bullet elim_{\mathbb{N}}\ [\![cmd_0]\!]\ h\ (\sigma.\mathtt{n})\ \sigma$$

where $h =_{df} \lambda n \bullet \lambda v \bullet [\![cmd_1[\mathtt{m}/n][p[\mathtt{m}]/v]]\!]$ and:

$$[\![(\lambda \mathtt{z} \bullet U)[\mathtt{n}]]\!] = \{g \mid \exists f \bullet g = \lambda \sigma \bullet f\ (\sigma.\mathtt{n})\ \sigma \wedge f \in [\![(\lambda \mathtt{z} \bullet U)]\!]\}$$

Hence, the premise is equivalent to:

$$\exists f \bullet \lambda \sigma \bullet elim_{\mathbb{N}}\ [\![cmd_0]\!]\ h\ (\sigma.\mathtt{n})\ \sigma = \lambda \sigma \bullet f\ (\sigma.\mathtt{n})\ \sigma \wedge f \in [\![(\lambda \mathtt{z} \bullet U)]\!]$$

so, for arbitrary σ, and fresh y_0 we have:

$$elim_{\mathbb{N}}\ [\![c_0]\!]\ h = y_0 \wedge y_0 \in [\![(\lambda \mathtt{z} \bullet U)]\!] \qquad (1)$$

Now, from the second conjunct of (1), we see that:

$$\exists f \bullet y_0 = curry_{\mathtt{z}}\ f \wedge f \in [\![U]\!]$$

or, for fresh y_1:

$$y_0 = curry_{\mathtt{z}}\ y_1 \wedge y_1 \in [\![U]\!] \qquad (2)$$

Substituting the first conjunct of (2) in the first conjunct of (1) yields:

$$elim_{\mathbb{N}}\ [\![cmd_0]\!]\ h\ = curry_{\mathtt{z}}\ y_1$$

So

$$uncurry_{\mathtt{z}}\ (elim_{\mathbb{N}}\ [\![cmd_0]\!]\ h) = uncurry_{\mathtt{z}}\ (curry_{\mathtt{z}}\ y_1)$$

which is $[\![p]\!] = y_1$. But $y_1 \in [\![U]\!]$, from the second conjunct of (2), hence $[\![p]\!] \in [\![U]\!]$, that is, $p \notin U$, as required.

The elimination rule also holds under the more general definition for recursive procedures:

Proposition 5.13. The following rule is derivable:

$$\frac{p \notin U}{p[exp] \notin (\lambda \mathtt{z} \bullet U)[exp]}$$

Proof. We have to establish that: $[\![p[exp]]\!] \in [\![(\lambda \mathtt{z} \bullet U)[exp]]\!]$ which is:

$$\exists f \bullet elim_{\mathbb{N}}\ [\![cmd_0]\!]\ h\ (\sigma.exp)\ \sigma = \lambda \sigma \bullet f\ (\sigma.exp)\ \sigma \wedge f \in [\![\lambda z \bullet U]\!]$$

where $h =_{df} \lambda n \bullet \lambda v \bullet [\![cmd_1[\mathtt{m}/n][p[\mathtt{m}]/v]]\!]$. This follows if we can show that:

$$elim_{\mathbb{N}}\ [\![cmd_0]\!]\ h \in [\![\lambda z \bullet U]\!]$$

in other words, that:

$$\exists f \bullet elim_{\mathbb{N}}\ [\![cmd_0]\!]\ h = curry_{\mathtt{z}}\ f \wedge f \in [\![U]\!]$$

this in turn, follows if:

$$uncurry_{\mathtt{z}}\ (elim_{\mathbb{N}}\ [\![cmd_0]\!]\ h) \in [\![U]\!]$$

but this is just $[\![p]\!] \in [\![U]\!]$, which is the premise $p \notin U$.

As usual, we may combine this with a β-reduction of the schema application term leading to an alternative version of the rule:

Corollary 5.14.

$$\frac{p \notin U}{p[exp] \notin U[\mathtt{z}/exp]}$$

The most important rule is the introduction rule for recursive synthesis.

Proposition 5.15. The following introduction rule for recursive procedure introduction is derivable:

$$\frac{cmd_0 \notin (\lambda \mathtt{z} \bullet U)[0] \quad p[\mathtt{m}] \notin (\lambda \mathtt{z} \bullet U)[\mathtt{m}] \vdash cmd_1 \notin (\lambda \mathtt{z} \bullet U)[\mathtt{m}+1]}{\mathtt{proc}\ p[\mathtt{z}]\ \mathtt{cases}\ \mathtt{z}\ \mathtt{in}\ 0 : cmd_0\ \mid\ \mathtt{m}+1 : cmd_1\ \mathtt{endcases} \notin U}\ (rp^+)$$

Proof. The conclusion is $[\![p]\!] \in [\![U]\!]$, which, by proposition 5.12 will hold if:

$$[\![p[\mathtt{n}]]\!] \in [\![(\lambda \mathtt{z} \bullet U)[\mathtt{n}]]\!]$$

for arbitrary $\mathtt{n}$. Note that:

$$[\![p[\mathtt{n}]]\!] = \lambda \sigma \bullet elim_{\mathbb{N}}\ [\![cmd_0]\!]\ h\ (\sigma.\mathtt{n})\ \sigma$$

where $h =_{df} \lambda n \bullet \lambda v \bullet [\![cmd_1[\mathtt{m}/n][p[\mathtt{m}]/v]]\!]$). Also note that:

$$[\![(\lambda \mathtt{z} \bullet U)[\mathtt{n}]]\!] = \{g \mid \exists f \bullet g = \lambda \sigma \bullet f\ (\sigma.\mathtt{n})\ \sigma \wedge f \in [\![\lambda z \bullet U]\!]\}$$

So we have to show:

$$\exists f \bullet \lambda \sigma \bullet elim_{\mathbb{N}}\ [\![cmd_0]\!]\ h\ (\sigma.\mathtt{n})\ \sigma = \lambda \sigma \bullet f\ (\sigma.\mathtt{n})\ \sigma \wedge f \in [\![\lambda \mathtt{z} \bullet U]\!]$$

We proceed by induction on the number $\sigma.\mathtt{n}$.
In the base case we have to show that:

$$\exists f \bullet \lambda \sigma \bullet elim_{\mathbb{N}}\ [\![cmd_0]\!]\ h\ 0\ \sigma = \lambda \sigma \bullet f\ 0\ \sigma \wedge f \in [\![\lambda \mathtt{z} \bullet U]\!]$$

which follows if:

$$\exists f \bullet [\![cmd_0]\!] = \lambda \sigma \bullet f\ 0\ \sigma \wedge f \in [\![\lambda \mathtt{z} \bullet U]\!]$$

or if:

$$[\![cmd_0]\!] \Subset \{g \mid \exists f \bullet g = \lambda\sigma \bullet f\ 0\ \sigma \wedge f \in [\![\lambda \mathtt{z} \bullet U]\!]\}$$

that is:

$$cmd_0 \Subset (\lambda z \bullet U)[0]$$

which is the first premise.
For the induction case, let $k = \sigma.\mathtt{m}$.
First note the following equivalence:

$$\lambda\sigma \bullet elim_{\mathbb{N}}\ [\![cmd_0]\!]\ h\ (k+1)\ \sigma =$$

$$\lambda\sigma \bullet (\lambda n \bullet \lambda v \bullet [\![cmd_1[\mathtt{m}/n][p[\mathtt{m}]/v]]\!])\ k\ (elim_{\mathbb{N}}\ [\![cmd_0]\!]\ h\ k)\ \sigma =$$

$$\lambda\sigma \bullet (\lambda n \bullet \lambda v \bullet [\![cmd_1[\mathtt{m}/n][p[\mathtt{m}]/v]]\!])\ ([\![\mathtt{m}]\!]\ \sigma)\ [\![p[\mathtt{m}]]\!]\ \sigma =$$

$$\lambda\sigma \bullet [\![cmd_1[\mathtt{m}/n][p[\mathtt{m}]/v]]\!]\ [n/([\![\mathtt{m}]\!]\ \sigma)][v/[\![p[\mathtt{m}]]\!]]\ \sigma$$

$$[\![cmd_1[\mathtt{m}/n][n/\mathtt{m}][p[\mathtt{m}]/v][v/p[\mathtt{m}]]]\!] =$$

$$[\![cmd_1]\!]$$

That is:

$$\lambda\sigma \bullet elim_{\mathbb{N}}\ [\![cmd_0]\!]\ h\ (k+1)\ \sigma = [\![cmd_1]\!] \qquad (1)$$

Now we have to show, assuming:

$$\exists f \bullet \lambda\sigma \bullet elim_{\mathbb{N}}\ [\![cmd_0]\!]\ h\ k\ \sigma = \lambda\sigma \bullet f\ k\ \sigma \wedge f \in [\![\lambda \mathtt{z} \bullet U]\!] \qquad (2)$$

that:

$$\exists f \bullet \lambda\sigma \bullet elim_{\mathbb{N}}\ [\![cmd_0]\!]\ h\ (k+1)\ \sigma = \lambda\sigma \bullet f\ (k+1)\ \sigma \wedge f \in [\![\lambda \mathtt{z} \bullet U]\!] \qquad (3)$$

This follows by showing that (2) follows from the assumption of the second premise of the rule, and that (3) follows from the conclusion of the second premise of the rule. The second premise's conclusion states that:

$$[\![cmd_1]\!] \in [\![(\lambda \mathtt{z} \bullet U)[\mathtt{m}+1]]\!]$$

Which, in view of the fact that $\sigma.(\mathtt{m}+1) = \sigma.\mathtt{m} + 1 = k+1$, is:

$$[\![cmd_1]\!] \in \{g \mid \exists f \bullet g = \lambda\sigma \bullet f\ (k+1)\ \sigma \wedge f \in [\![\lambda \mathtt{z} \bullet U]\!]\}$$

which is:

$$\exists f \bullet [\![cmd_1]\!] = \lambda\sigma \bullet f\ (k+1)\ \sigma \wedge f \in [\![\lambda \mathtt{z} \bullet U]\!]$$

In view of (1) we can write this as:

$$\exists f \bullet \lambda\sigma \bullet elim_{\mathbb{N}}\ [\![cmd_0]\!]\ h\ (k+1)\ \sigma = \lambda\sigma \bullet f\ (k+1)\ \sigma \wedge f \in [\![\lambda \mathtt{z} \bullet U]\!]$$

which is (3) as required.
The second premise assumption states that:

$$[\![p[\mathtt{m}]]\!] \in [\![(\lambda \mathtt{z} \bullet U)[\mathtt{m}]]\!]$$

which, since $\sigma.\mathtt{m} = k$, is:

$$\lambda\sigma \bullet curry_{\mathtt{z}}\ [\![p]\!]\ k\ \sigma \in \{g \mid \exists f \bullet g = \lambda\sigma \bullet f\ k\ \sigma \wedge f \in [\![\lambda \mathtt{z} \bullet U]\!]\}$$

or:

$$\exists f \bullet \lambda\sigma \bullet curry_{\mathtt{z}}\ [\![p]\!]\ k\ \sigma = \lambda\sigma \bullet f\ k\ \sigma \wedge f \in [\![\lambda \mathtt{z} \bullet U]\!]$$

or, using the semantics of the procedure p, is:

$$\exists f \bullet \lambda\sigma \bullet curry_{\mathtt{z}}(uncurry_{\mathtt{z}}(elim_{\mathbb{N}}\ [\![cmd_0]\!]\ h))\ k\ \sigma = \lambda\sigma \bullet f\ k\ \sigma \wedge f \in [\![\lambda \mathtt{z} \bullet U]\!]$$

which simplifies to:

$$\exists f \bullet \lambda\sigma \bullet elim_{\mathbb{N}}\ [\![cmd_0]\!]\ h)\ k\ \sigma = \lambda\sigma \bullet f\ k\ \sigma \wedge f \in [\![\lambda \mathtt{z} \bullet U]\!]$$

which is (2) as required.

At this point our programming logic for $\mathcal{P}_{\mathbb{N}}$ is established, and we finish by showing that it satisfies principle 4.3 the *frame freezing principle*.

Proposition 5.16. The logic for $\mathcal{P}_{\mathbb{N}}$ satisfies principle 4.3.

Proof. This is formally an induction on the structure of derivations. The key point, however, is this: in the semantics for $\mathcal{P}_{\mathbb{N}}$ the only command that can alter the state is the assignment command and the only rule involving the introduction of assignment commands is the rule ($\notin \overset{+}{:=}$). This rule, for an assignment to $\mathtt{x}$, involves the co-observation $\mathtt{x}'$ explicitly. So, if $\mathtt{x}'$ is not visible in the alphabet of an arbitrary derivation δ, then there is no instance of an assignment to $\mathtt{x}$ in that derivation which lies outside a sub-derivation δ_0 whose conclusion has the form $\exists \mathtt{x}, \mathtt{x}' \bullet U$. The semantics of existential hiding ensures that the state is not changed at $\mathtt{x}$ by any of its implementations, hence the state at $\mathtt{x}$ is not altered by any implementation of the derivation δ_0. Since $\mathtt{x}'$ is not visible in δ there can be no other instances of assignment to $\mathtt{x}$ apart from those just considered. So the state is not changed by any implementation of the derivation at $\mathtt{x}$. This establishes the principle.

5.8. Proof-theoretic simplifications

In imperative programming languages, such as $\mathcal{P}_{\mathbb{N}}$, the state is an implicit variable, only making an appearance in the semantics of the language. Similarly, in schemas, the observations have values in an implicit state (binding); this point was discussed in section 2.1 (and discussed at length in our other papers *e.g.* [6], [8]) where we remarked that the states become explicit in derivations which ultimately take place in the underlying logical system $\mathcal{Z}_{\mathcal{C}}$.
It would be very much more pleasant if we could avoid mention of the state entirely, even in derivations which eventually require calculation in the core logic. In fact this is possible, since the states in question are always arbitrary and before and after states are always related by priming. So we may treat observations as terms in the proofs, in the presence of additional axioms that enforce equality between two observations if, and only if, they are identical. Let us now be more precise about this.

Definition 5.17. Let ${\mathcal{Z}_{\mathcal{C}}}^+$ be the system $\mathcal{Z}_{\mathcal{C}}$ with the following additional axioms for all observations $\mathtt{x}$.

$$\overline{P \vdash_{\mathcal{Z}_{\mathcal{C}}^+} \mathtt{x} = \mathtt{x}}$$

Lemma 5.18. For any state σ, and term t:

$$\sigma.\sigma'.t = \sigma.\mathit{unprime}\ t$$

Proof. By induction on the structure of terms.

Proposition 5.19. Let δ_0 be a $\mathcal{Z}_{\mathcal{C}}$ derivation whose conclusion is $\sigma.P \vdash_{\mathcal{Z}_{\mathcal{C}}} \sigma.\sigma'.Q$. There exists a $\mathcal{Z}_{\mathcal{C}}^+$ derivation δ_1 with conclusion $P \vdash_{\mathcal{Z}_{\mathcal{C}}^+} \mathit{unprime}\ Q$. The converse is also true.

Proof. This is an induction on the structure of the derivations (in both directions). We illustrate with two cases of the proof from left to right. First one of the induction cases. Consider a $\mathcal{Z}_{\mathcal{C}}$ derivation whose conclusion is $\sigma.P \vdash_{\mathcal{Z}_{\mathcal{C}}} \sigma.\sigma'.(Q_0 \wedge Q_1)$ by virtue of $(\wedge^+)$. We therefore have two $\mathcal{Z}_{\mathcal{C}}$ derivations with conclusions $\sigma.P \vdash_{\mathcal{Z}_{\mathcal{C}}} \sigma.\sigma'.Q_0$ and $\sigma.P \vdash_{\mathcal{Z}_{\mathcal{C}}} \sigma.\sigma'.Q_1$. *Ex hypothesi* we have two $\mathcal{Z}_{\mathcal{C}}^+$ derivations whose conclusions are $P \vdash_{\mathcal{Z}_{\mathcal{C}}^+} \mathit{unprime}\ Q_0$ and $P \vdash_{\mathcal{Z}_{\mathcal{C}}^+} \mathit{unprime}\ Q_1$, whence, by $(\wedge^+)$ we have a $\mathcal{Z}_{\mathcal{C}}^+$ derivation of $P \vdash_{\mathcal{Z}_{\mathcal{C}}^+} \mathit{unprime}\ (Q_0 \wedge Q_1)$ as required. Now let us look at the crucial atomic case. Suppose we have a $\mathcal{Z}_{\mathcal{C}}$ derivation whose conclusion is $\sigma.P \vdash_{\mathcal{Z}_{\mathcal{C}}} \sigma.\sigma'.(\mathtt{x}_0 = \mathtt{x}_1)$ for some observations (possibly primed) $\mathtt{x}_0$ and $\mathtt{x}_1$. By distribution of the substitution, and lemma 5.18, this can be written as $\sigma.P \vdash_{\mathcal{Z}_{\mathcal{C}}} \sigma.\mathit{unprime}\ \mathtt{x}_0 = \sigma.\mathit{unprime}\ \mathtt{x}_1$. We also have $\sigma.P \vdash_{\mathcal{Z}_{\mathcal{C}}} \sigma.\mathtt{x}_0 = \sigma.\mathtt{x}_1$ without loss of generality, if we suppose that the observations are not primed. Since the state σ is arbitrary the only way this can hold is if the two observations are identical. Hence we have a derivation immediately from the new axioms of $\mathcal{Z}_{\mathcal{C}}^+$ with conclusion $P \vdash_{\mathcal{Z}_{\mathcal{C}}^+} \mathtt{x}_0 = \mathtt{x}_1$ as required.

In the programming logic we have introduced we then obtain simpler rules as follows:

Proposition 5.20. The following rules are derivable for the skip command:

$$\frac{\mathtt{skip} \Subset [D \mid P \mid Q] \quad P}{unprime\ Q}\ (\Subset^{-}_{\mathtt{skip}}) \qquad \frac{P \vdash unprime\ Q}{\mathtt{skip} \Subset [D \mid P \mid Q]}\ (\Subset^{+}_{\mathtt{skip}})$$

Proof. Straightforward, in view of proposition 5.19

Proposition 5.21. The following rule (in which no after state identifier may occur in the expressions exp_i) is derivable for simultaneous assignment to the variables x_i:

$$\frac{P \vdash Q[\cdots \mathtt{x}'_i / exp_i \ \cdots]}{\cdots \mathtt{x}_i \cdots \mathtt{:=}\ \cdots exp_i \cdots \Subset [\cdots \mathtt{x}_i, \mathtt{x}'_i\ \cdots : \mathbb{N};\ D \mid P \mid Q]}\ (\Subset^{+}_{\mathtt{:=}})$$

Proof. Suppose we have a $\mathcal{Z}_C$ derivation whose conclusion is:

$$\sigma.P \vdash_{\mathcal{Z}_C} \sigma.\sigma[\cdots \mathtt{x}_i / \sigma.exp_i \ \cdots]'.Q$$

Then this is equivalently:

$$\sigma.P \vdash_{\mathcal{Z}_C} \sigma.\sigma'.Q[\cdots \mathtt{x}'_i / exp_i \ \cdots]$$

By proposition 5.19 we have an equivalent $\mathcal{Z}^{+}_{C}$ derivation with conclusion:

$$P \vdash Q[\cdots \mathtt{x}'_i / exp_i \ \cdots]$$

as required.

The following are also straightforward.

Proposition 5.22. Weakening preconditions:
(*i*)

$$\frac{P_1 \vdash P_0}{[T \mid P_0 \mid P] \sqsupseteq [T \mid P_1 \mid P]}\ (\sqsupseteq^{+}_{pre})$$

Strengthening postconditions:
(*ii*)

$$\frac{P_0 \vdash P_1}{[T \mid P \mid P_0] \sqsupseteq [T \mid P \mid P_1]}\ (\sqsupseteq^{+}_{post})$$

6. Examples

6.1. Two simple derivations from a single specification

We begin with a trivial example specification, illustrating that it is possible to derive distinct programs which meet it. In this initial example we will show full details. In the later examples we will suppress some of the trivial steps.
Consider the following specification. The state is:

$$S =_{df} [\mathtt{x} : \mathbb{N}]$$

The operation schema is:

Add

ΔS
$\mathtt{z}? : \mathbb{N}$

$\mathtt{x}' = \mathtt{x} + \mathtt{z}?$

6.1.1. Simple procedure

$(Add^{\lambda z^?})[\mathtt{n}]$ reduces, using β-reduction, to the following schema:

*Add**

$$\Delta S$$
$$\mathtt{n} : \mathbb{N}$$
$$\mathtt{x}' = \mathtt{x} + \mathtt{n}$$

Then we can use a simple non-recursive procedure and addition. Formally we would have:

$$\dfrac{\dfrac{\dfrac{}{\mathtt{x} + \mathtt{n} = \mathtt{x} + \mathtt{n}}\ (ref)}{\mathtt{x} := \mathtt{x} + \mathtt{n} \Subset Add^{\lambda z^?}[\mathtt{n}]}\ (\Subset^{+}_{:=})}{p \Subset Add}\ (proc^{+})$$

where:

$$\mathtt{proc}\ p[\mathtt{z}?]\ \mathtt{x} := \mathtt{x} + \mathtt{z}?$$

6.1.2. Recursive procedure

Alternatively, we can obtain a recursive procedure. We need to note the following special cases of $(Add^{\lambda z^?})[\mathtt{n}]$ ($Add[\mathtt{z}?/n]$ or $Add[\mathtt{n}]$ when the parameter is clear from the context).
First $Add[0]$:

$$\Delta S$$
$$\mathtt{x}' = \mathtt{x}$$

and then $Add[\mathtt{n} + 1]$:

$$\Delta S$$
$$\mathtt{n} : \mathbb{N}$$
$$\mathtt{x}' = (\mathtt{x} + \mathtt{n}) + 1$$

The latter can be expressed as the composition of two simpler schemas, $Add[n]$:

$$\Delta S$$
$$\mathtt{n} : \mathbb{N}$$
$$\mathtt{x}' = \mathtt{x} + \mathtt{n}$$

and:

Succ

$$\Delta S$$
$$\mathtt{x}' = \mathtt{x} + 1$$

That is, we can show that:

$$Add[\mathtt{n}] \fatsemi Succ \sqsupseteq Add[\mathtt{n} + 1]$$

Of course, we can also write the derivation formally in the logic:

$$\dfrac{\dfrac{}{\mathtt{x} = \mathtt{x}}\ (ref) \quad \dfrac{}{Add[\mathtt{n}] \fatsemi Succ \sqsupseteq Add[\mathtt{n}+1]} \quad \dfrac{\dfrac{}{p[\mathtt{m}] \in Add[\mathtt{m}]} \quad \dfrac{}{\mathtt{x}+1 = \mathtt{x}+1}}{\mathtt{x}\mathtt{:=}\ \mathtt{x}+1 \in Succ}}{\dfrac{\mathtt{skip} \in Add[0]}{p[\mathtt{m}];\ \mathtt{x}\mathtt{:=}\ \mathtt{x}+1 \in Add[\mathtt{n}] \fatsemi Succ} \quad p[\mathtt{m}];\ \mathtt{x}\mathtt{:=}\ \mathtt{x}+1 \in Add[\mathtt{m}+1]}}{p \in Add}$$

where the procedure p is thus given as follows:

$$\mathtt{proc}\ p[\mathtt{z}?]\ \mathtt{cases}\ 0 : \mathtt{skip}\ \mid\ \mathtt{m}+1 : p[\mathtt{m}];\ \mathtt{x}\mathtt{:=}\ \mathtt{x}+1\ \mathtt{endcases}$$

These derivations are less easy to read than their elaborations, though they are easy to construct from them. So, in future examples we will not provide the formal derivations.
Note that we could define:

$$\mathtt{if}\ \mathtt{n} > 0\ \mathtt{then}\ cmd\ \mathtt{fi} =_{df} \mathtt{cases}\ 0\ : \mathtt{skip}\ \mid\ \mathtt{m}+1\ : cmd\ \mathtt{endcases}$$

in which case the program could be written:

$$\mathtt{proc}\ p[\mathtt{z}?]\ \mathtt{if}\ \mathtt{n} > 0\ \mathtt{then}\ p[\mathtt{m}];\ \mathtt{x}\mathtt{:=}\ \mathtt{x}+1\ \mathtt{fi}$$

The idea here of introducing new syntax by definitional extension is likely to be of importance in the future development of the framework. The example here is very rudimentary. We will see a more significant example in the next section.

6.2. Factorial

In this example we demonstrate how to derive two simple recursive programs from the standard definition. As a consequence we illustrate many of the techniques available in the framework.
The factorial function is, as usual, specified as follows:

$$\begin{array}{lcl} fact(0) & = & 1 \\ fact(n+1) & = & (n+1) * fact(n) \end{array}$$

The initial specification is:

Fact

$\mathtt{x}, \mathtt{x}' : \mathbb{N}$
$\mathtt{z}? : \mathbb{N}$

$\mathtt{x}' = fact(\mathtt{z}?)$

6.2.1. First development

The strategy in this case will be to introduce an accumulator.

Fact*

$\mathtt{x}, \mathtt{x}', \mathtt{y}, \mathtt{y}' : \mathbb{N}$
$\mathtt{z}? : \mathbb{N}$

$\mathtt{y}' = \mathtt{z}? + 1$
$\mathtt{x}' = fact(\mathtt{z}?)$

By 4.2 and 4.1(ii) we have:

$$Fact^* \sqsupseteq Fact$$

Next we formulate the curried version of $Fact^*$, $(\lambda \mathtt{z}? \bullet Fact^*)[\mathtt{n}]$, which, at input $\mathtt{n}$, is, by β-reduction, $Fact^*[\mathtt{n}]$:

$$\mathtt{n}, \mathtt{x}, \mathtt{x}', \mathtt{y}, \mathtt{y}' : \mathbb{N}$$
$$\mathtt{y}' = \mathtt{n} + 1$$
$$\mathtt{x}' = \mathit{fact}(\mathtt{n})$$

The point of this is to prepare the way for the construction of a program by recursion on $\mathtt{n}$, more exactly, by a recursive procedure f. In preparation, we first obtain $\mathit{Fact}^*[0]$:

$$\mathtt{x}, \mathtt{x}', \mathtt{y}, \mathtt{y}' : \mathbb{N}$$
$$\mathtt{y}' = 1$$
$$\mathtt{x}' = 1$$

and, with similar simplification, $\mathit{Fact}^*[\mathtt{m} + 1]$:

$$\mathtt{m}, \mathtt{x}, \mathtt{x}', \mathtt{y}, \mathtt{y}' : \mathbb{N}$$
$$\mathtt{y}' = \mathtt{m} + 2$$
$$\mathtt{x}' = \mathit{fact}(\mathtt{m} + 1)$$

The first of these can be refined to a simultaneous assignment:

$$\mathtt{x}, \mathtt{y} := 1, 1$$

$\mathit{Fact}^*[\mathtt{m} + 1]$ can be further decomposed into the composition of $\mathit{Fact}^*[\mathtt{m}]$ and:

Step

$$\mathtt{x}, \mathtt{x}', \mathtt{y}, \mathtt{y}' : \mathbb{N}$$
$$\mathtt{x}' = \mathtt{x} * \mathtt{y}$$
$$\mathtt{y}' = \mathtt{y} + 1$$

Now, the assumption from rule (rp^+) allows us to conclude that $\mathit{Fact}^*[\mathtt{m}]$ is implemented by $f[\mathtt{m}]$ and *Step* is clearly implemented by just another simultaneous assignment $\mathtt{x}, \mathtt{y} := \mathtt{x} * \mathtt{y}, \mathtt{y} + 1$. The program all this yields is:

$$\mathtt{proc}\ f[\mathtt{z}?]\ \mathtt{cases}\ 0 : x, y := 1, 1 \mid \mathtt{m} + 1 : f[\mathtt{m}];\ \mathtt{x}, \mathtt{y} := \mathtt{x} * \mathtt{y}, \mathtt{y} + 1\ \mathtt{endcases}$$

6.2.2. Second development

Our first development demonstrates something unusual about our framework: the ability to very easily derive a procedure which introduces global side-effects. We can also derive a program which ensures the accumulation variable is part of a local state.

This second development is really very important because it amounts to a signal for the future elaboration of our applications framework: we will require an extension of our programming language to allow *recursive procedures with local variables*. Now these could, of course, be added to the syntax of the programming language, their semantics provided and rules for using them derived. This has not been our methodology in the past: we began our research in this area by introducing a core logic $\mathcal{Z}_C$ which we have not altered despite the construction of a significant logic for the schema calculus and, in this paper, for refinement and program derivation. If we continue to follow our logical strategy in our applications area we should rather see whether or not the new programming constructs we might need can be introduced by *conservative extension by definitions*. If this can be achieved the relevant semantics and necessary rules will be available in a systematic way from the existing base system. In the example we are concerned with here we are interested in adding a new form of procedure which has the form:

$$\mathtt{proc}\ p[\mathtt{z}?]\ \mathtt{begin\ var\ x;\ cases}\ 0 : \mathit{cmd}_0 \mid \mathtt{m} + 1 : \mathit{cmd}_1\ \mathtt{end}$$

Note that this new form of procedure is not a simple complex of existing programming idioms: the body of

the block is not a command for example. We take this new syntax to be defined by means of:

$$\texttt{proc}\ p[\texttt{z?}]\ \texttt{begin var x;}\ p_0[\texttt{z?}]\ \texttt{end}$$

where:

$$\texttt{proc}\ p_0[\texttt{z?}]\ \texttt{cases}\ 0 : cmd_0\ \mid\ \texttt{m} + 1 : cmd_1[p/p_0]\ \texttt{end}$$

Rather than derive general rules for this new idiom in this paper we will now simply derive our program along the lines of its definition above. This will indicate in general terms, by means of a special case, how those rules can be formed. We will cover this in detail, alongside other similar novel features, in future publications.
So we begin by deciding to implement *Fact* by means of a simple procedure p:

$$p \Subset Fact$$

By $(proc^+)$, we will need to find a command *cmd* such that:

$$cmd[\texttt{n}] \Subset Fact[\texttt{n}]$$

We now make the observation that:

$$\exists\, u, v \bullet v = \texttt{n} + 1$$

holds for any n. We use this as the premise for 4.13 and obtain the refinement:

$$Fact^\dagger[\texttt{n}] \sqsupseteq Fact[\texttt{n}]$$

where $Fact^\dagger$ is:

$$\exists\, \texttt{y}, \texttt{y}' \bullet [\texttt{x}, \texttt{x}', \texttt{y}, \texttt{y}', \texttt{z?} : \mathbb{N} \mid \texttt{x}' = fact(\texttt{z?}) \wedge \texttt{y}' = \texttt{z?} + 1]$$

$Fact^\dagger[\texttt{n}]$ can be implemented by means of a block as usual. That is:

$$\texttt{begin var y;}\ cmd_0\ \texttt{end} \Subset Fact^\dagger[\texttt{n}]$$

and it remains for us to construct a command cmd_0 so that:

$$cmd_0 \Subset [\texttt{x}, \texttt{x}', \texttt{y}, \texttt{y}', \texttt{n} : \mathbb{N} \mid \texttt{x}' = fact(\texttt{n}) \wedge \texttt{y}' = \texttt{n} + 1]$$

It so happens that our previous derivation comes to our aid at this point, because the schema here is simply $Fact^*[\texttt{n}]$. Now we know that $f \Subset Fact$, and therefore, by (rp^-), $f[\texttt{n}] \Subset Fact[\texttt{n}]$. So we can take cmd_0 to be $f[\texttt{n}]$.
Summarising our development we have:

$$\texttt{proc}\ p[\texttt{z?}]\ \texttt{begin var y;}\ f[\texttt{z?}]\ \texttt{end}$$

where

$$\texttt{proc}\ f[\texttt{n}]\ \texttt{cases}\ 0 : \texttt{x, y:=}\ 1, 1\ \mid\ \texttt{m} + 1 : f[\texttt{m}]\texttt{; x, y:=}\ \texttt{x} * \texttt{y}, \texttt{y} + 1\ \texttt{endcases}$$

which, according to our new idiom, is:

$$\texttt{proc}\ p[\texttt{z?}]\ \texttt{begin var y; cases}\ 0 : \texttt{x, y:=}\ 1, 1\ \mid\ \texttt{m} + 1 : p[\texttt{m}]\texttt{; x, y:=}\ \texttt{x} * \texttt{y}, \texttt{y} + 1\ \texttt{end}$$

As we mentioned above, it would be appropriate to introduce special tailored rules for this idiom. This example indicates how that could be done, but we will leave further investigation for the future.

6.3. Using choice

In this example we demonstrate how to derive a simply recursive program from a course of values definition. This example illustrates the use of disjunctive choice.
The Fibonacci numbers are, as usual, specified as follows:

$$\begin{array}{lcl} \mathit{fib}(0) & = & 1 \\ \mathit{fib}(1) & = & 1 \\ \mathit{fib}(n+2) & = & \mathit{fib}(n+1) + \mathit{fib}(n) \end{array}$$

The initial specification is:

Fib

$$\mathtt{y}, \mathtt{y}' : \mathbb{N}$$
$$\mathtt{z}? : \mathbb{N}$$
$$\mathtt{y}' = \mathit{fib}(\mathtt{z}?)$$

The strategy for obtaining a simply recursive program for this specification is to introduce another accumulator:

Fib*

$$\mathtt{x}, \mathtt{x}', \mathtt{y}, \mathtt{y}' : \mathbb{N}$$
$$\mathtt{z}? : \mathbb{N}$$
$$\mathtt{y}' = \mathit{fib}(\mathtt{z}?)$$
$$\mathtt{x}' = \mathit{fib}(\mathtt{z}? - 1)$$

Clearly we have:

$$\mathit{Fib}^* \sqsupseteq \mathit{Fib}$$

by frame expansion and strengthening postconditions.
Preparing to derive a recursive program by (rp^+) we next we formulate the curried version of Fib^*, $(\lambda \mathtt{z}? \bullet \mathit{Fib}^*)[\mathtt{n}]$ or, by β-reduction, $\mathit{Fib}^*[\mathtt{n}]$ which is:

$$[\mathtt{x}, \mathtt{x}', \mathtt{y}, \mathtt{y}', \mathtt{n} : \mathbb{N} \mid \mathtt{y}' = \mathit{fib}(\mathtt{n}) \land \mathtt{x}' = \mathit{fib}(\mathtt{n} - 1)]$$

Leading to $\mathit{Fib}^*[0]$:

$$[\mathtt{x}, \mathtt{x}', \mathtt{y}, \mathtt{y}' : \mathbb{N} \mid \mathtt{y}' = 1 \land \mathtt{x}' = 1]$$

and: $\mathit{Fib}^*[\mathtt{m} + 1]$:

$$[\mathtt{x}, \mathtt{x}', \mathtt{y}, \mathtt{y}', \mathtt{m} : \mathbb{N} \mid \mathtt{y}' = \mathit{fib}(\mathtt{m} + 1) \land \mathtt{x}' = \mathit{fib}(\mathtt{m})]$$

The first of these can be refined to a simultaneous assignment:

$$\mathtt{x}, \mathtt{y} := 1, 1$$

The second schema can now be expressed as a disjunction by splitting the (implicit) true precondition into two cases: zero and successor. We use inequation 4.6 for this.

$$U_0 \lor U_1 \sqsupseteq \mathit{Fib}^*[\mathtt{m} + 1]$$

where U_0 is:

$$[\mathtt{x}, \mathtt{x}', \mathtt{y}, \mathtt{y}', \mathtt{m} : \mathbb{N} \mid \mathtt{m} = 0 \mid \mathtt{y}' = 1 \land \mathtt{x}' = 1]$$

and U_1 is:

$$[\mathtt{x}, \mathtt{x}', \mathtt{y}, \mathtt{y}', \mathtt{m} : \mathbb{N} \mid \exists\, u \bullet \mathtt{m} = u + 1 \mid \mathtt{y}' = \mathit{fib}(\mathtt{m} + 1) \land \mathtt{x}' = \mathit{fib}(\mathtt{m})]$$

For commands cmd_0 and cmd_1 which meet U_0 and U_1 we implement the disjunction by:

$$\mathtt{if}\ \mathtt{m} = 0\ \mathtt{then}\ \mathit{cmd}_0\ \mathtt{else}\ \mathit{cmd}_1$$

We can weaken the precondition of U_0 to obtain:

$$U_2 =_{df} [\mathtt{x}, \mathtt{x}', \mathtt{y}, \mathtt{y}', \mathtt{m} : \mathbb{N} \mid \mathtt{y}' = 1 \land \mathtt{x}' = 1]$$

which we know can be implemented by a simultaneous assignment, as above. U_1 can be further refined into the composition of:

$$U_3 =_{df} [\mathtt{x}, \mathtt{x}', \mathtt{y}, \mathtt{y}', \mathtt{m} : \mathbb{N} \mid \exists\, u \bullet \mathtt{m} = u + 1 \land \mathtt{y}' = \mathit{fib}(\mathtt{m}) \land \mathtt{x}' = \mathit{fib}(\mathtt{m} - 1)]$$

and:

$$\mathit{Step} =_{df} [\mathtt{x}, \mathtt{x}', \mathtt{y}, \mathtt{y}', \mathtt{m} : \mathbb{N} \mid \mathtt{x}' = \mathtt{y} \land \mathtt{y}' = \mathtt{x} + \mathtt{y}]$$

We can now introduce a sequence of commands into our program in which the obvious simultaneous assignment:

$$\mathtt{x}, \mathtt{y} := \mathtt{y}, \mathtt{x} + \mathtt{y}$$

appears as the second component. That is:

$$U_3 \fatsemi \mathit{Step} \sqsupseteq U_1$$

And this will, for a suitable command cmd_2, be refined to:

$$cmd_2 \ ; \ \mathtt{x}, \mathtt{y} := \mathtt{y}, \mathtt{x} + \mathtt{y}$$

We demonstrate this in more detail by observing that:

$$\exists\, u \bullet \mathtt{m} = u + 1 \land \mathtt{y}' = \mathit{fib}(\mathtt{m}) \land \mathtt{x}' = \mathit{fib}(\mathtt{m} - 1)$$

is:

$$\exists\, u \bullet \mathtt{m} = u + 1 \land \mathtt{y}' = \mathit{fib}(u + 1) \land \mathtt{x}' = \mathit{fib}(u)$$

Composing with *Step* yields:

$$[\mathtt{x}, \mathtt{x}', \mathtt{y}, \mathtt{y}', \mathtt{m} : \mathbb{N} \mid \exists\, u \bullet n = u + 1 \land \mathtt{y}' = \mathit{fib}(u + 1) + \mathit{fib}(u) \land \mathtt{x}' = \mathit{fib}(u + 1)]$$

or:

$$[\mathtt{x}, \mathtt{x}', \mathtt{y}, \mathtt{y}', \mathtt{m} : \mathbb{N} \mid \exists\, u \bullet \mathtt{n} = u + 1 \land \mathtt{y}' = \mathit{fib}(u + 2) \land \mathtt{x}' = \mathit{fib}(u + 1)]$$

or:

$$\mathtt{x}, \mathtt{x}', \mathtt{y}, \mathtt{y}', \mathtt{m} : \mathbb{N} \mid \exists\, u \bullet \mathtt{m} = u + 1 \land \mathtt{y}' = \mathit{fib}(\mathtt{m} + 1) \land \mathtt{x}' = \mathit{fib}(\mathtt{m})]$$

which is U_1, as required.
We similarly weaken the precondition of U_3 to obtain:

$$[\mathtt{x}, \mathtt{x}', \mathtt{y}, \mathtt{y}', \mathtt{m} : \mathbb{N} \mid \mathtt{y}' = \mathit{fib}(\mathtt{m} + 1) \land \mathtt{x}' = \mathit{fib}(\mathtt{m})]$$

which is just $\mathit{Fib}^*[\mathtt{m}]$, and can be implemented by the recursive call that is available as an assumption from the second premise of the rule (rp^+).
The program this yields is:

```
proc   fibonacci[z?]
       cases        0 : x, y := 1, 1
                    m + 1 : if m = 0  then x, y := 1, 1
                                      else fibonacci[m] ; x, y := y, x + y
       endcases
```

Alternatively, this can be combined with a similar analysis to our derivation in section 6.2.2 to make the accumulator in this procedure local rather than global.

6.4. An example using promotion

6.4.1. Specification

We now wish to use the specification *Fact* above, together with its implementation, to specify and then implement an operation over a global state. In the global state we have two numbers. This can be represented by the cartesian product $\mathbb{N} \times \mathbb{N}$.
The global operation simply generalises the local operation by applying it to the first of the pair. The promotion schema as usual explains *how* the local and global state spaces are to be connected:

Promote

$$\mathtt{x}, \mathtt{x}' : \mathbb{N}$$
$$\mathtt{w}, \mathtt{w}' : \mathbb{N} \times \mathbb{N}$$

$$\mathtt{w}'.1 = \mathtt{x}'$$
$$\mathtt{w}'.2 = \mathtt{w}.2$$

and the global operation is:

$$\mathit{GlobalFact} =_{df} \exists\, \mathtt{x}, \mathtt{x}' : \mathbb{N} \bullet \mathit{Fact}^*[\mathtt{z}?/\mathtt{w}.1] \land \mathit{Promote}$$

6.4.2. Refinement

$Fact^*[\mathtt{w}.1] \wedge Promote$ can be refined, by 4.5, to:

FP

$$\mathtt{x}, \mathtt{x}' : \mathbb{N}$$
$$\mathtt{w}, \mathtt{w}' : \mathbb{N} \times \mathbb{N}$$
$$\mathtt{x}' = fact(\mathtt{w}.1)$$
$$\mathtt{w}'.1 = \mathtt{x}'$$
$$\mathtt{w}'.2 = \mathtt{w}.2$$

This can then be refined, by 4.8, to the composition of:

FP_0

$$\mathtt{x}, \mathtt{x}' : \mathbb{N}$$
$$\mathtt{w}, \mathtt{w}' : \mathbb{N} \times \mathbb{N}$$
$$\mathtt{x}' = fact(\mathtt{w}.1)$$
$$\mathtt{w}' = \mathtt{w}$$

and:

FP_1

$$\mathtt{x}, \mathtt{x}' : \mathbb{N}$$
$$\mathtt{w}, \mathtt{w}' : \mathbb{N} \times \mathbb{N}$$
$$\mathtt{w}'.1 = \mathtt{x}$$
$$\mathtt{w}'.2 = \mathtt{w}.2$$
$$\mathtt{x} = \mathtt{x}'$$

Now we know from the previous example that $f \Subset Fact$, and by frame freezing we know that $f \Subset \Psi\mathtt{w} \bullet Fact$. Note that $(\Psi\mathtt{w} \bullet Fact)[\mathtt{w}.1] = FP_0$. So, by 5.14:

$$f[\mathtt{w}.1] \Subset FP_0$$

We also have:

$$\mathtt{w}.1 := \mathtt{x} \Subset FP_1$$

so, by further use of $(\Subset {}^{+}_{\circ\,9})$ and the refinement we know that:

$$f[\mathtt{w}.1]\,;\ \mathtt{w}.1 := \mathtt{x} \Subset FP$$

and finally, by using $(\Subset {}^{+}_{block})$ we have:

$$\mathtt{begin\ var\ x};\ f[\mathtt{w}.1]\,;\ \mathtt{w}.1 := \mathtt{x}\ \mathtt{end} \Subset GlobalFact$$

6.5. Promotion continued

An related but distinct example of promotion along the lines of the previous example is now explored. Suppose that the global operation is specified by means of:

$$GF =_{df} \exists\, \mathtt{x}, \mathtt{x}' \bullet Fact \wedge Promote$$

We aim to implement this using a simple procedure:

$$\mathtt{proc}\ p[\mathtt{z}?]\, cmd$$

for some command *cmd*. This can be achieved, using $(proc^{-})$, if we can derive:

$$cmd[\mathtt{n}] \Subset (\lambda \mathtt{z}? \bullet GF)[\mathtt{n}]$$

Now using β-reduction and the definition of substitution, this is equivalent to:

$$cmd[\mathtt{n}] \in \exists\, \mathtt{x}, \mathtt{x}' \bullet Fact[\mathtt{n}] \land Promote$$

since $\mathtt{z}?$ only appears in *Fact*. Now this can be achieved, using $(\in^{+}_{block})$, taking $cmd[\mathtt{n}]$ to be the block `begin var x;` cmd_0 `end` and providing that:

$$cmd_0 \in Fact[\mathtt{n}] \land Promote$$

We next observe that this specification can be refined, by 4.5 to:

*FP**

$\mathtt{x}, \mathtt{x}', \mathtt{n} : \mathbb{N}$
$\mathtt{w}, \mathtt{w}' \in \mathbb{N} \times \mathbb{N}$

$\mathtt{x}' = fact(\mathtt{n})$
$\mathtt{w}'.1 = \mathtt{x}'$
$\mathtt{w}'.2 = \mathtt{w}.2$

Now, by 4.8, this can be refined to:

$$\Psi \mathtt{w} \bullet Fact[\mathtt{n}] \fatsemi FP_1$$

where FP_1 was defined in the previous example; hence we know that:

$$\mathtt{w}.1 := \mathtt{x} \in FP_1$$

We also know, by the frame freezing principle and $(proc^{-})$, that:

$$f[\mathtt{n}] \in \Psi \mathtt{w} \bullet Fact[\mathtt{n}]$$

since $f \in Fact$. So, using $(\in^{+}_{\fatsemi})$ and the refinement, we have:

$$f[\mathtt{n}]\,;\ \mathtt{w}.1 := \mathtt{x} \in \Psi \mathtt{w} \bullet Fact[\mathtt{n}] \fatsemi FP_1$$

Assembling this program from the derivation leads to:

$$\mathtt{proc}\ p[\mathtt{z}?]\ \mathtt{begin\ var\ x;}\ f[\mathtt{z}?]\,;\ \mathtt{w}.1 := x\ \mathtt{end} \in GF$$

and the example is complete.

6.6. Comments on the examples

We hope these few examples serve to illustrate our approach to refinement and program development. Although the reader will need to see, or to undertake, many more examples in order to fully evaluate our framework, our examples do involve a reasonable use of schema algebra and the facilities permitted in the programming language. What we are very happy to have achieved is a satisfactory integration of the refinement and derivation logics within our earlier general logic for the schema calculus. All this is achieved on the basis of the core logic $\mathcal{Z}_{\mathcal{C}}$, which remains unmodified.

Naturally, in providing such a logical framework the nature of the specifications, or at least their role, changes. In the core logic, or even in that logic extended as it was in [8] for the schema calculus, specifications remain essentially a formalisation of *requirements*. The logic then permits those requirements to be analysed and their consequences understood and explored. In moving to include logics for refinement and program derivation, specifications become, additionally, records of *design*: the structures introduced or eliminated along the way dictating something of the structure which leads eventually to an implementation.

Another important feature of the framework is the *modularity* it permits. In our example concerning promotion it is evident that the program development, just like the specification itself, can be factored into entirely separate components; the development of an implementation of a local operation can proceed quite independently of the development of the promotion. This fulfills two important properties: *extensionally*, the independent program development fragments come together to produce a correct implementation of the entire specification; and *intensionally*, the algorithmic choices made in the development of the local operation carry over into the implementation of the specification as a whole.

Finally we should comment on the portfolio of rules for refinement and derivation that are at our disposal in this paper. We have only included the most basic and critical rules. One of the reasons we have chosen the examples we have is to at least show our methodology for further extending our set of rules. This was most clearly indicated in our second derivation of the factorial program. As far as possible we will aim to introduce new constructs, as we have relentlessly done in this paper and our previous work, by conservative extension by definitions. In this way the basic system remains small and tractable, and additional semantics and rules are easily derived from that.

7. Conclusions and future work

The first four sections of this paper comprise a general framework for schema-based specification refinement and program development. We describe a novel semantics based on sets of implementations which we argue is suitable for refinement in the presence of schema operators. We further show that this semantics can be applied to specifications in a standard Z notation, and also in a manner more reminiscent of the refinement calculus. It is the latter approach that we deal with in most detail, though retaining a Z-like notation for the presentation and development. The two final technical sections of the paper are devoted to applications of the framework. We firstly connected the framework to a simple programming language and established a programming logic for it. We then explored some simple examples in this application area.

Much remains to be explored before definitive conclusions can be drawn. We can, at this stage, point to the novel semantics and the integrated approach to specification refinement and program development it leads to. A thorough comparison of the new semantics with other approaches, in particular the standard semantics based on weakest preconditions, must be undertaken, possibly making use of or connections with relevant sections of [3]. It is likely that these comparisons will benefit from consideration in an abstract setting, quite separate from details of the framework as described here, and in examining these issues more generally, including the variety of relational approaches to refinement.

More prosaically, though importantly, we have, in the applications sections of the paper, only explored a very impoverished programming language and undertaken rather simple illustrative examples. There is much work of generalisation to undertake in this area.

Finally we should point out that the refinement relation explored in this paper is restricted to operation refinement only (simulations are absent because they are identity functions). Naturally the framework should be generalised to include data refinement. Again, the sensible precursor to this will be a thorough investigation of data refinement in the context of the novel semantics, and in comparison to other approaches, in a more abstract setting.

8. Acknowledgements

We would like to thank the EPSRC (grant reference: GR/L57913), the Royal Society of Great Britain and the New Zealand Foundation for Research, Science and Technology (grant reference: UOWX0011) for financially supporting this research.

This work has been influenced in its development by too many people to name explicitly. However, special thanks for particularly important discussions and comments go to Eerke Boiten, Moshe Deutsch, Lindsay Groves, Greg Reeve, Grant Anderson, Ray Turner and Jim Woodcock.

References

[1] A. Cavalcanti. A refinement calculus for Z. Technical report, D. Phil. thesis, University of Oxford, 1997.

[2] A. Cavalcanti and J. Woodcock. ZRC—a refinement calculus for Z. *Formal Aspects of Computing*, 10(3):267—289, 1998.

[3] W. P. DeRoever and K. Engelhardt. *Data refinement: model-oriented proof methods and their comparison*. Prentice Hall International, 1998.

[4] L. Groves. Adapting program derivations using program conjunction. In J. Grundy, M. Schwenke, and T. Vickers, editors, *International Refinement Workshop and Formal Methods Pacific'98*, Springer series in discrete mathematics and theoretical computer science, pages 145—164. Springer, 1998.

[5] M. C. Henson and S. Reeves. New foundations for Z. In J. Grundy, M. Schwenke, and T. Vickers, editors, *Proc. International Refinement Workshop and Formal Methods Pacific '98*, pages 165–179. Springer, 1998.

[6] M. C. Henson and S. Reeves. Revising Z: I - logic and semantics. *Formal Aspects of Computing Journal*, 11(4):359–380, 1999.
[7] M. C. Henson and S. Reeves. Revising Z: II - logical development. *Formal Aspects of Computing Journal*, 11(4):381–401, 1999.
[8] M. C. Henson and S. Reeves. Investigating Z. *Journal of Logic and Computation*, 10(1):1–30, 2000.
[9] S. King. Z and the Refinement Calculus. In D. Bjørner, C. A. R. Hoare, and H. Langmaack, editors, *VDM '90 VDM and Z—Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*, pages 164–188. Springer-Verlag, April 1990.
[10] P. Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science VI*, pages 153–175. North Holland, 1982.
[11] C. Morgan. *Programming from Specifications*. Prentice Hall International, 2nd. edition, 1998.
[12] J. Woodcock and S. Brien. $\mathcal{W}$: A logic for *Z*. In *Proceedings of ZUM '91, 6th Conf. on Z*. Springer Verlag, 1992.
[13] J. Woodcock and J. Davies. *Using Z: Specification, Refinement and Proof*. Prentice Hall, 1996.

A. The specification logic $\mathcal{Z}_\mathcal{C}$

A.1. Language

In this appendix we shall describe the simple specification logic $\mathcal{Z}_\mathcal{C}$ from [8]. This is included for convenience only and the reader may need to consult the earlier paper at least in order to fully understand our notational and meta-notational conventions.
$\mathcal{Z}_\mathcal{C}$ is a typed theory in which the types of higher-order logic are extended with *schema types* which are unordered, label-indexed tuples. For example, if the T_i are types and the $\mathbf{z}_i$ are labels (constants) then:

$$[\cdots \mathbf{z}_i : T_i \cdots]$$

is a (schema) type. The symbols $\preceq$, $\curlywedge$ and $\curlyvee$ denote the *schema subtype* relation, and the operations of *schema type intersection* and (compatible) *schema type union*.
The terms of $\mathcal{Z}_\mathcal{C}$ are evident from the logic below. We write t^T to indicate that the term t has type T. We use the meta-variable C for terms which are sets. Of particular note are the *bindings*, the terms of schema type. The bindings of type $[\cdots \mathbf{z}_i : T_i \cdots]$ have the form $\langle\!| \cdots \mathbf{z}_i \Rrightarrow t_i^{T_i} \cdots |\!\rangle$. We use the $\star$ operation to denote *binding concatenation*: it is only defined when the alphabets of its two argument bindings are disjoint. We also make use of a meta-language substitution for bindings:

$$b[\mathbf{z}_0/v].\mathbf{z}_1 =_{df} \begin{cases} v & \text{when} \quad \mathbf{z}_0 = \mathbf{z}_1 \\ b.\mathbf{z}_1 & \text{otherwise} \end{cases}$$

We employ the notation $b.P$ and $b.t$ (generalising binding selection) which is adapted from [12]. Suppose that $\{\mathbf{z}_0 \cdots \mathbf{z}_n\}$ is the alphabet set of t, then $t.P$ is $P[\mathbf{z}_0/t.\mathbf{z}_0] \cdots [\mathbf{z}_n/t.\mathbf{z}_n]$. An important lemma for us in this paper is:

Lemma A.1.

$$b.P[\mathbf{z}/t] \Leftrightarrow b[\mathbf{z}/b.t].P$$

Proof. By induction over the structure of propositions and terms.

If the binding b has type $[\cdots \mathbf{z} : T \cdots]$ then the priming operation means that the binding b' has type $[\cdots \mathbf{z}' : T \cdots]$. We extend this operation to function applications over schema types: if $f \in [\cdots \mathbf{z} : T_0 \cdots] \to [\cdots m : T_1 \cdots]$ and $z \in T_0$ then $f\ z \in [\cdots m : T_1 \cdots]$ as expected, but $(f\ z)' \in [\cdots m' : T_1 \cdots]$ (note the prime here). Finally, we need to define *unprime* P which is the proposition P in which all primed observations are unprimed.

A.2. Logic

The judgements of the logic have the form $\Gamma \vdash_{\mathcal{Z}_\mathcal{C}} P$ where Γ is a set of formulæ.
The logic is presented as a natural deduction system *in sequent form*. Derivations in the logic, above, were presented in *pure* natural deduction form.
All data (entailment symbol, contexts, type *etc.*) which remains unchanged by a rule are omitted. In the rule

$(\exists^-)$, the variable y may not occur in C, P_0, P_1 nor any other assumption. We begin with the usual rules for $\mathcal{Z}_\mathcal{C}$.

$$\frac{P_0}{P_0 \vee P_1}\ (\vee^+_{\mathrm{o}}) \qquad \frac{P_1}{P_0 \vee P_1}\ (\vee^+_{\mathrm{1}}) \qquad \frac{P_0 \vee P_1 \quad P_0 \vdash P_2 \quad P_1 \vdash P_2}{P_2}\ (\vee^-)$$

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

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

$$\frac{}{\Gamma, P \vdash P}\ (ass) \qquad \frac{\Gamma \vdash P_1}{\Gamma, P_0 \vdash P_1}\ (wk)$$

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

$$\frac{}{\langle\!|\, \cdots \mathbf{z}_i \Rrightarrow t_i \cdots \,|\!\rangle.\mathbf{z}_i = t_i}\ (\Rrightarrow^=_{\mathrm{o}}) \qquad \frac{}{\langle\!|\, \cdots \mathbf{z}_i \Rrightarrow t.\mathbf{z}_i \cdots \,|\!\rangle = t^{[\cdots \mathbf{z}_i : T_i \cdots]}}\ (\Rrightarrow^=_{\mathrm{1}})$$

$$\frac{}{(t, t').1 = t}\ ((,)^=_{\mathrm{o}}) \qquad \frac{}{(t, t').2 = t'}\ ((,)^=_{\mathrm{1}}) \qquad \frac{}{(t.1, t.2) = t}\ ((,)^=_{\mathbf{2}})$$

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

$$\frac{t_0 = t_1}{t_0 = t_1}\ (ext) \qquad \frac{t^{\,T}.\mathbf{z}_i = t_i}{(t \restriction T').\mathbf{z}_i = t_i}\ (\restriction^=) \quad (\mathbf{z}_i \in \alpha T';\ T' \preceq T)$$

$$\frac{}{\mathit{elim}_{\mathbb{N}}\ t_0\ t_1\ \mathsf{Zero} = t_0}\ (\mathit{elim}_{\mathbb{N}}{}^{\mathrm{o}}) \qquad \frac{}{\mathit{elim}_{\mathbb{N}}\ t_0\ t_1\ (\mathsf{Succ}\ z) = t_1\ z\ (\mathit{elim}_{\mathbb{N}}\ t_0\ t_1\ z)}\ (\mathit{elim}_{\mathbb{N}}{}^{\mathrm{1}})$$

$$\frac{}{\mathit{elim}_{\mathbb{B}}\ \mathsf{True}\ t_0\ t_1\ = t_0}\ (\mathit{elim}_{\mathbb{B}}{}^{\mathrm{o}}) \qquad \frac{}{\mathit{elim}_{\mathbb{B}}\ \mathsf{False}\ t_0\ t_1\ = t_1}\ (\mathit{elim}_{\mathbb{B}}{}^{\mathrm{1}})$$

where:

$$t_0 = t_1 =_{df} \forall\, z : t_0 \bullet z \in t_1 \wedge \forall\, z : t_1 \bullet z \in t_0$$

The usual side-conditions apply to rule $(\exists^-)$.
The symmetry and transitivity of equality and numerous *equality congruence* rules for the various term forming operations are all derivable in view of rule (*sub*).

A.3. Carrier sets

We need to introduce a carrier set for each type.
We begin with the type of natural numbers. Since we will wish to exploit the inductive structure of this basic type we will make the following definition:[4]

Definition A.2.

$$\mathbb{N} =_{df} \{x^{\mathbb{N}} \mid \forall\, z^{\mathbb{P}\mathbb{N}} \bullet \mathsf{Zero} \in z \wedge (\forall\, y^{\mathbb{N}} \bullet y \in z \Rightarrow \mathsf{Succ}\ y \in z) \Rightarrow x \in z\}$$

Then the rules for the *set* of natural numbers will then be as expected.

Proposition A.3. The following introduction and elimination rules for natural numbers are derivable:

$$\frac{}{\mathsf{Zero} \in \mathbb{N}} \qquad \frac{z \in \mathbb{N}}{\mathsf{Succ}\ z \in \mathbb{N}} \qquad \frac{P[z/\mathsf{Zero}] \quad x \in \mathbb{N}, P[z/x] \vdash P[z/\mathsf{Succ}\ x]}{z \in \mathbb{N} \vdash P}$$

[4] The notational ambiguity heralds no danger, since only *sets* appear as terms.

For convenience we write 2 for Succ Succ Zero, and so on.
The carrier for the type of booleans is easily given.

Definition A.4.

$$\mathbb{B} =_{df} \{z^{\mathbb{B}} \mid z = \mathsf{True} \vee z = \mathsf{False}\}$$

Proposition A.5. The following introduction and elimination rules for booleans are derivable:

$$\frac{}{\mathsf{True} \in \mathbb{B}} \qquad \frac{}{\mathsf{False} \in \mathbb{B}} \qquad \frac{z \in \mathbb{B} \quad z = \mathsf{True} \vdash P \quad z = \mathsf{False} \vdash P}{P}$$

The carriers of the other types, are then given structurally, in the obvious way.

A.4. Typed set theory

Set-theoretic relations and operators, such as containment, union, intersection and complement, are all easily definable *within each type*, and we will make free use of these when necessary. For example:

Definition A.6.

$$C_0^T \cup_T C_1^T =_{df} \{z^T \mid z \in C_0 \wedge z \in C_1\}$$

Functions are as usual sets of ordered pairs. We write $C_0 \to C_1$ to denote the usual subset of $\mathbb{P}(C_0 \times C_1)$ that satisfies totality and unicity. We need to make extensive use of lambda notation to define particular functions. Our syntax for these is traditional: $\lambda z^{T_0} \bullet t^{T_1}$ is the element of $T_0 \to T_1$ that associates each v in T_0 with the value $t[z/v]$ in T_1. Note that *all our term formation constructors preserve termination*, hence these lambda abstractions denote total functions. In particular, recursion is only available via the *primitive* recursion operator $elim_{\mathbb{N}}$ over the type of natural numbers.
The composition of two functions f_0 and f_1 (of appropriate type) is written $f_0 \circ f_1$ and is defined to be $\lambda z \bullet f_1(f_0 z)$, as usual.

A.5. Filtered Sets

We shall also need to extend filtering from bindings to sets of bindings.

Definition A.7. Let $T_0 \preceq T_1$.

$$C^{\mathbb{P}\,T_1} \restriction T_0 =_{df} \{z^{T_0} \mid \exists\, x^{T_1} \bullet x \in C \wedge z = x \restriction T_0\}$$

Proposition A.8. Let $T_0 \preceq T_1$.

$$\frac{t^{T_1} \in C^{\mathbb{P}\,T_1}}{t \restriction T_0 \in C \restriction T_0}\ (\in^+_{\restriction}) \qquad \frac{t \in C^{\mathbb{P}\,T_1} \restriction T_0 \quad x \in C, t = x \restriction T_0 \vdash P}{P}\ (\in^-_{\restriction})$$

for fresh x.

A.6. Restricted equality

In many contexts we need to compare bindings over a common restricted type.

Definition A.9. Let $T \preceq T_0$ and $T \preceq T_1$.

$$t_0^{T_0} =_T t_1^{T_1} =_{df} t_0 \restriction T = t_1 \restriction T$$

The following versions of reflexivity, symmetry and transitivity are obvious:

Lemma A.10.

$$\frac{}{t \restriction T =_T t}\ (refl_{\restriction}) \qquad \frac{t_1 =_T t_0}{t_0 =_T t_1}\ (sym_{\restriction}) \qquad \frac{t_0 =_{T_0} t_1 \quad t_1 =_{T_1} t_2}{t_0 =_{(T_0 \curlywedge T_1)} t_2}\ (trans_{\restriction})$$

When the type in question is that of one of the terms being compared, we can avoid the subscript.

Definition A.11.

$$t_0^{T_0} \doteq t_1^{T_1} =_{df} t_0 \upharpoonright (T_0 \nwarrow T_1) = t_1 \upharpoonright (T_0 \nwarrow T_1) \qquad (T_1 \preceq T_0 \textit{ or } T_0 \preceq T_1)$$

Proposition A.12. Let $T_0 \preceq T_1$.

$$t_0^{T_0} \doteq t_1^{T_1} \Leftrightarrow t_0 =_{T_0} t_1$$

A similar restricted form of membership is very useful for establishing the state schema calculus.

Definition A.13. Let $T_0 \preceq T_1$.

$$t^{T_1} \ \dot{\in} \ C^{\mathbb{P}\, T_0} =_{df} t \upharpoonright T_0 \in C$$

A.7. Restricted extensional equality

Definition A.14.

$$f =_C g =_{df} \forall z \in C \bullet f\ z = g\ z$$

Whence:

Definition A.15.

$$f =_P g =_{df} f =_{\{z | z.P\}} g$$

Generally we have:

Proposition A.16. The following rules are derivable:

$$\frac{z \in C \vdash f\ z = g\ z}{f =_C g}$$

and

$$\frac{f =_C g \quad t \in C}{f\ t = g\ t}$$

Also:

Proposition A.17. The following rules are derivable:

$$\frac{z.P \vdash f\ z = g\ z}{f =_P g}$$

and

$$\frac{f =_P g \quad t.P}{f\ t = g\ t}$$