Comparing Extended Z with a Heterogeneous Notation for Reasoning about Time and Space

Richard F. Paige

Department of Computer Science, York University, Toronto, Ontario, Canada, M3J 1P3. paige@cs.yorku.ca

Abstract. We contrast using a notation extension with using a combination of notations. Specifically, we compare the use of an extended dialect of Z [10] with a combination of Z and predicative programming notation [6] for algorithm refinement and for reasoning about time and space constraints on systems. We discuss the difficulty of using extended notations versus using heterogeneous notations, and consider when we might prefer to extend or combine notations. We conclude that there exist situations where a heterogeneous notation can be more appropriate to use than an extended notation.

1 Introduction

Notation extension, the process of adding new syntactic or semantic features to a language, is an important topic in formal method and software engineering research. Notations have been extended for a variety of reasons: in order to generalize the notation; in order to use concepts available in other languages that have proven to be useful in practice; or, to compare notations in a systematic manner. Notation extension has also been used for notation *development*, the process for which starts with a concise kernel language, that over time is developed into a general-purpose language.

An alternative approach to extension is to construct *heterogeneous notations*. A heterogeneous notation is manufactured from two or more distinct languages, combining the syntax and semantics of each in some manner so as to produce a new language. Heterogeneous notations are used to write *heterogeneous specifications*, which are compositions of partial specifications written in two or more different notations [9]. A specifier might prefer to use a heterogeneous notation rather than extend a notation for a number of reasons: e.g., to keep individual notations simple; or to be able to write specifications in the most suitable specialized language [5, 9, 13].

In this paper, we are interested in contrasting the use of an extended Z dialect with the use of a heterogeneous notation composed from Z and predicative notation [6]. We do this so as to make a preliminary attempt to determine if there exist situations where heterogeneous notations are *useful*, for writing more understandable (or possibly more concise) specifications, and for producing simpler, more understandable refinement proofs than are possible with extended notations. To this end, we will carry out two case studies in applying an extended language and a heterogeneous notation. The case studies will be as follows.

- *Timing study:* the Z notation, extended with a notion of time [3], will be applied in specification, refinement, and proof of timing constraints. Its use will be contrasted with an application of Z combined with predicative programming [6]—which has a built-in notion of timing.
- *Space study:* the Z notation, extended with a simple notion of space, will be applied in specification, refinement, and proof of space constraints. Its use will be contrasted with an application of Z combined with predicative programming that has built-in notations for space [7].

The applications of the extended notations and heterogeneous notations are not meant to be illustrative of all of the features of the individual approaches; the reader is directed to the references for further examples [3, 9]. Rather, the case studies are intended to demonstrate how heterogeneous notations and extended notations can be used, and how the use of heterogeneous notations can compare to use of extended notations. It is also important to note that our comparisons are given in terms of *existing* and very specific notations, and in terms of eliminating limitations of these notations.

The organization of this paper will be as follows.

- In Section 2, we outline the predicative programming method. We also explain our definition of heterogeneous notations, and how heterogeneous specifications are to be formally manipulated, based on the approach of [9].
- In Section 3, we compare the use of Z extended with reasoning about time with the use of a heterogeneous notation combining Z and predicative programming. We carry out a refinement case study in order to compare the two approaches.
- In Section 4, we compare the use of Z extended with reasoning about space with the use of a heterogeneous notation combining Z and predicative programming. Again, we carry out a refinement case study in order to compare the two approaches.
- In Section 5, we discuss our findings, and attempt to suggest why using a heterogeneous approach may be preferable to an extension approach, and vice versa.

2 Notation and Approaches to Extension and Heterogeneity

2.1 Predicative programming

Predicative programming is due to Hehner [6]. It is a program design calculus like Morgan's refinement calculus [8], but unlike the latter treats programs as specifications. In this approach, programs and specifications are predicates on pre- and poststate (final values of variables are annotated with a prime; initial values of variables are undecorated). The weakest predicate specification is $\top$ ("true"), and the strongest specification is $\perp$ ("false"). Refinement is just boolean implication.

Definition 1. A predicative specification P on prestate σ and poststate σ' is refined by a specification Q if $\forall \sigma, \sigma' \cdot (P \Leftarrow Q)$.

The refinement relation enjoys various properties that allow specifications to be refined by parts, steps, and cases. Since refinement is just implication, carrying out a refinement is equivalent to carrying out a logical proof. Therefore, the refinement rules of predicative programming are laws of boolean logic; see [6] for a complete list.

Predicative specifications can be combined using the familiar operators of boolean theory, along with all the usual program combinators (e.g., '.' is the sequencing combinator), as well as combinators for parallelism and communication through channels. Predicative programming also has a **frame** construct equivalent to that of [8]. The specification **frame** $w \cdot P$ means that predicate P can change variables w, but no other variables; if the state consists of variables w and ρ, then **frame** $w \cdot P$ is equivalent to $(P \land \rho' = \rho)$.

One particular novelty with predicative programming is that *recursive* programs can be developed rather than iterative programs, using recursive refinement rules. It has been suggested that this simplifies the process of developing certain programs [6], since in particular it eliminates the need to construct invariants *before* developing loops.

Predicative specifications do not express constraints on termination. Instead, specifications can include reference to time variables, t and t', which can be used to place time bounds on any implementation. Furthermore, predicative specifications may include constraints on space that must be met by any implementation. Space constraints are expressed through references to a space variable s, which represents the current space usage of a specification, and a maximum space variable m, which represents the maximum usage. Detailed examples of using space and time variables can be found in [6, 7]. In general, non-trivial space usage arises with recursive implementations and programs, so in the case studies that use predicative programming, we typically use recursive refinement techniques, which are very useful in proving space bounds [7].

As an example of a predicative specification, the following specification requires reversal of the order of items in a list L, taking no more than $\#L \text{ div } 2$ units of time (where $\#L$ is list length, and one unit is the time for a recursive call).

$$\#L' = \#L \land \forall n : 0, ..\#L \cdot L'(n) = L(\#L \Leftrightarrow n \Leftrightarrow 1) \land t' \leq t + \#L \text{ div } 2$$

2.2 Approach to extension

In the two case studies using extended notations in Sections 3 and 4, we use standard Z notation [10], extended to specifying and reasoning about *timing constraints* and *space usage constraints*. We now outline the approaches to extension that we will use.

For the timing case study in Section 3, we use the real-time Z extension described in [3]; alternatives are listed in the references [1, 2]. In this approach, Z specifications are extended with fresh variables t and t', which express the time when a computation starts and when a computation finishes. In [3], constraints placed on t and t' can be sets of times; without loss of generality, we will only place deterministic constraints on computation time, in order to keep the examples of manageable size.

In order to reason about time in specifications, all specifications will include the schema $\Delta Time$, as follows ($\mathbb{N}_1$ is all non-zero naturals).

$$\Delta Time \mathrel{\widehat{=}} [\, t, t' : \mathbb{N}_1 \mid t' \geq t \,]$$

The extension in [3] modifies refinement rules from [12] to include constraints on time. These constraints express how long the primitive commands in a simple programming language will take—for example, time constraints for evaluating the truth of guards, or for branching back to the top of a loop after an executing the loop body. The rules include references to a schema-valued function, *age*, which returns instantiations of time-change schemas specifying the passage of a specific amount of time. The primitive time-change schema, *ChangeTime*, which we use in the case study is

$$ChangeTime \mathrel{\widehat=} \left[\, \Delta Time;\ \Xi ProgVar;\ d : \mathbb{N}_1 \mid t' - t = d \,\right]$$

where *d* is an amount of time that some program will take, while *ProgVar* is a schema of program variables of interest. The *age* function is defined as follows.

$$\begin{array}{|l} age : \mathbb{N}_1 \to ChangeTime \\ \hline \forall\, t : \mathbb{N}_1 \cdot age(t).d = t \end{array}$$

Without loss of generality, we will use only two timing constraints: an assignment takes one unit of time (for the store to memory); and a branch back to the top of a **do** loop takes one unit of time. All other commands take no time. We will use the same constraints in both the extension example and the heterogeneous example. This set of timing constraints is unrealistic for some problems, but it is sufficient for the case studies herein. These timing constraints mean that the effect of an assignment statement $x := E$ is defined by the operation schema *AssignEtoX* (where *x* can be many variables).

$$AssignEtoX \mathrel{\widehat=} \left[\, x, x' : T;\ \Xi AllElse;\ \Delta Time \mid x' = E \land t' - t = 1 \,\right]$$

(where *AllElse* is all variables in scope, excepting those in *x*). We use $x := E$ as syntax for the schema *AssignEtoX* when using Z.

The refinement rules from [3] are low-level, in that from them it is possible to derive ‘short-cut’ rules that make the approach more practical (involving shorter proof steps). Thus, the complexity of proofs in the extended Z method that we use can be reduced. In order to be consistent in the comparisons, we use low-level laws in applying the heterogeneous method. In this manner, we have a common basis for comparison.

The extension of Z to reason about space is much simpler; in fact, for the most part standard Z specifications can be written (with inclusion of some extra schema details and syntax). To reason about space, we declare a state schema *∆Space*, which defines instances of two new variables: *s*, the current space usage of a program executing a specification; and *m*, the maximum space usage of a program executing a specification.

$$\Delta Space \mathrel{\widehat=} \left[\, m, m', s, s' : \mathbb{N} \mid m \geq s \land m' \geq s' \land m' \geq m \,\right]$$

∆Space is to be included in any specifications for which we want to reason about space usage. Non-trivial space usage only results with specifications that are to be implemented by recursive programs. So in the space case studies, we write specifications as procedures, with Z specifications for bodies. These specifications will be implemented using recursive calls to the original procedure. We will write procedure interfaces using the notation of [8], and base the development of recursive programs on [8] as well (though modified to Z).

The refinement rules that we use for reasoning about space in Z are unchanged from those in [12], since space usage is represented by two variables. In order to calculate space bounds, in proofs we preface a recursive call to a procedure by $s := s + 1$ and $m := max\{m, s + 1\}$. On return from a recursive call, we decrease space usage by $s := s \Leftrightarrow 1$. With such rules, bounds on space can be proven with respect to a specification and implementation.

2.3 Approach to heterogeneity

We aim to compare the use of an extended version of Z with a heterogeneous notation constructed from Z and predicative programming. Since we are interested in formally reasoning about heterogeneous specifications composed from Z specifications and predicative specifications, we need to give the heterogeneous notation a formal semantics.

The approach to formally defining the meaning of heterogeneous notations that we use is from [9]. Translations are defined between formal notations of interest. The translations provide the mechanisms by which a heterogeneous specification can be given a formal semantics using a homogeneous specification, via mapping the original specification into a single-notation formulation[1]. A set of notations and translations between them, which is to be used to give a formal semantics to heterogeneous specifications, is called a *heterogeneous basis*. The small heterogeneous basis that we use in this paper consists of the Z notation and the predicative notation. It is derived from a much larger basis given in [9], which includes Z and predicative notation, as well as a number of other formal and informal notations. We require only one translation in the basis, a mapping from Z to predicative notation. See [9] for mappings from predicative notation to Z, and for other translations.

To translate from a Z schema $Op \mathrel{\widehat{=}} [\, \Delta S;\ i? : I;\ o! : O \mid P \,]$ to a predicative specification, we use the translation *ZToPP*, defined as follows.

$$ZToPP(Op) \mathrel{\widehat{=}} \textbf{frame}\ w \cdot ((\exists\, w' \cdot P) \Rightarrow P)$$

The frame w consists of the variables in S and the operation outputs. Inputs $i?$ are mapped to state variables or to parameters (if the resulting specification is to be encompassed in a procedure or function). The existential quantifier is necessary in the translation to extract the precondition of the operation. We assume that any schema property for S has been expanded and included in the predicate P of Op. Though *ZToPP* is written as a total function, we require that for any Op, $P \neq true$, because predicative programming cannot describe terminating yet arbitrary computations [5].

To include time variables in the result of the translation *ZToPP*, the most we can say in the resulting predicative specification is that time does not decrease. This results in conjoining the predicate term $t' \geq t$ in the result of *ZToPP*.

In the heterogeneous method, we extend Z's definition of *max* to apply to empty sets of integers (we use the standard function *max* when applying pure Z). This is done so as to be able to develop equivalent programs using the extended and the heterogeneous methods. *max* applied to an empty set gives $\Leftrightarrow\infty$, which is smaller than any integer; ∞, correspondingly, is larger than any integer and $\Leftrightarrow\infty$. A full axiomatic definition of

[1] This notation need not be the same as the languages used for writing specifications.

max and 'extended' integers is beyond the scope and space constraints of this paper (but one can be found in [6]). We define $\mathbb{Z}_\infty$, the *extended integers,* as $\mathbb{Z} \cup \{\infty, \Leftrightarrow\infty\}$. The definition of *max* on sets of extended integers is

$$\begin{array}{|l}
max : \mathbb{P}\,\mathbb{Z}_\infty \nrightarrow \mathbb{Z}_\infty \\
\hline
max\{\,\} = \Leftrightarrow\infty \\
\forall\, S : \mathbb{P}_1\,\mathbb{Z}_\infty;\ m : \mathbb{Z}_\infty \bullet \\
\quad (max\ S = m) \Leftrightarrow (m \in S \wedge \forall\, n : S \bullet n \leq m)
\end{array}$$

2.3.1 Semantics of heterogeneous specifications The translation *ZToPP* can be used to formally define the semantics of compositions of Z specifications and predicative specifications, by translating heterogeneous specifications into a homogeneous specification. In this paper, heterogeneous specifications are given a semantics in terms of predicative notation. Details of how the translation process operates over combinators are in [9]. Informally, *ZToPP* applies partwise over predicative combinators. Therefore, we always write Z specifications under the assumption that we can translate them into predicative notation. While we have checked that translation is possible for the specifications in this paper, we do not show the checking in the case studies.

As an example, consider the semantics of the specification $(j = n) \Rightarrow Op$, where $j = n$ is a predicative specification, $\Rightarrow$ is boolean implication, and *Op* is the schema

$$Op \mathrel{\widehat{=}} [\, L, L' : \operatorname{seq}\mathbb{N};\ r, r' : \mathbb{N} \mid L' = L \wedge r' = max\{r, max\{i : j..n \Leftrightarrow 1 \bullet Li\}\}\,]$$

The semantics of this heterogeneous specification is

$$(j = n) \Rightarrow \textbf{frame}\ r, L \cdot (L' = L \wedge r' = max\{r, max\{i : j..n \Leftrightarrow 1 \bullet Li\}\})$$

This can be refined by the specification **ok**, which does nothing.

Some rules for refining heterogeneous specifications composed from Z and predicates were given in [9]. A useful rule that we will need in the timing case study is the following. Informally, the rule states the conditions to be checked in order to refine a schema by a predicative specification.

Rule 1. For a prestate σ and poststate σ', a Z schema with property P is refined by a predicative specification Q if

$$\forall\, \sigma, \sigma' \cdot ((\exists\, \sigma' \cdot P) \Rightarrow P) \Leftarrow Q$$

We introduce one new rule here, for the purposes of simplifying the process of refinement of heterogeneous specifications. It generalizes the *substitution rule* from [6].

Rule 2. Let x be a variable and E an expression, where '.' is predicative sequencing. If S is a schema with property P, then

$$(x := E.\ S) = S'[E/x]$$

where S' is the same as S except with property $(\exists\, \sigma' \cdot P) \Rightarrow P$. Read $S'[E/x]$ as "substitute E for x in the property of schema S'".

Informally, Rule 2 means that we can apply the predicative substitution rule when using Z schemas in refinements. The property of S' changes under the substitution due to the translation of Z into predicative notation: the existential quantification in the property is necessary because the predicative notation can express miraculous computations, while Z cannot.

In order to be able to use Z and predicative notations together, we must also be able to parse specification compositions. This means that we have to eliminate any syntactic ambiguity that arises by combining the two notations. For this reason, we do the following when writing heterogeneous specifications.

– We use standard Z notation for types, e.g., $\mathbb{Z}, \mathbb{N}$, and to describe ranges.
– Refinement in Z follows the approach of Wordsworth [12], and the refinement relation will be written as $\sqsubseteq$. Refinement of Z will be to the guarded command language of Dijkstra.
– We use Morgan's notation for writing procedures that have Z specification bodies [8]. Procedure syntax will be necessary for reasoning about space usage [7].

3 Reasoning about Time

Existing refinement methods, such as those presented in [6, 8, 12], provide the means for rigorously developing correct programs from specifications. In critical applications, correctness may also be measured in terms of performing actions at the right time. Therefore, it is necessary to reason about timing, too.

In this section, we compare the use of Z, extended with refinement rules for timing, with the use of Z combined with predicative programming. The comparison is done in the context of a small case study. We first outline the problem for the case study, then develop a solution using extended Z, and thereafter Z combined with predicates.

3.1 The problem

The very simple problem that we consider is taken from [3]; we choose it because it is small enough to allow comparisons of two separate solutions in the space available. We want to calculate the maximum r of a non-empty list of integers L in time that is proportional (within a constant factor) to the length, n, of the list.

3.2 Using extended Z

We start the example by specifying the state of the system. The system state contains three variables: the list L, the result r, and a counter variable j.

$$State \mathrel{\widehat{=}} [\, L : 0..n \Leftrightarrow 1 \rightarrow \mathbb{N};\; j : \mathbb{N};\; r : \mathbb{Z} \mid j \leq n \,]$$

The problem, S, that we want to solve, is therefore

$$\begin{array}{|l}
\multicolumn{1}{l}{S} \\ \hline
\Delta State;\ \Delta Time \\ \hline
L' = L \\
r' = max\{i : 0..n \Leftrightarrow 1 \bullet Li\} \\
t' \Leftrightarrow t = 2n \Leftrightarrow 1 \\ \hline
\end{array}$$

The time constraint, $t' \Leftrightarrow t = 2n \Leftrightarrow 1$, requires that any implementation of this specification takes $2n \Leftrightarrow 1$ units of time to execute.

We implement the problem S using the real-time refinement rules of [3]. The first step is to refine S into a sequence of two schemas, where the first initializes j and r (taking one unit of time). We show that $S \sqsubseteq A_1 ;\ A_2$, where

$$\begin{array}{|l}
\multicolumn{1}{l}{A_1} \\ \hline
\Delta State;\ \Delta Time \\ \hline
j' = 1 \land r' = L0 \\
L' = L \\
t' \Leftrightarrow t = 1 \\ \hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{A_2} \\ \hline
\Delta State;\ \Delta Time \\ \hline
j = 1 \land r = L0 \land L' = L \\
r' = max\{r, max\{i : j..n \Leftrightarrow 1 \bullet Li\}\} \\
t' \Leftrightarrow t = 2(n \Leftrightarrow 1) \\ \hline
\end{array}$$

To show that $S \sqsubseteq A_1 ;\ A_2$, the following obligations must be discharged.

$$\operatorname{pre} S \vdash \operatorname{pre} A_1 \tag{1}$$

$$\operatorname{pre} S \land A_1 \vdash (\operatorname{pre} A_2)' \tag{2}$$

$$\operatorname{pre} S \land A_1 \land A_2' \vdash S[\_''/\_'] \tag{3}$$

In (1), it is shown that the sequence precondition can be established by the precondition of S. In (2), we show that the precondition of the second step in the sequence, A_2, can be established by A_1 in a precondition of S. In the final step, we show that the sequence establishes the problem. (The priming notations are described in [12]. The notation $S[\_''/\_']$ means "substitute doubly primed variables for primed variables in S".) Steps (1) and (2) are straightforward (because $\operatorname{pre} S$ and $\operatorname{pre} A_1$ are *true*). The last step, (3), is the liveness condition, and it is also straightforward once we substitute primed values of variables from A_1 for unprimed values of variables in A_2, and see that this entails S.

A_1 is a schema expression for an assignment statement $r, j := L0, 1$ which takes one unit of time and needs no further refinement. The next step is to refine A_2. We see that A_2 determines the maximum of L for all elements except the first; therefore, it needs to be implemented by a loop which takes time $2(n \Leftrightarrow 1)$. An invariant for a loop that implements A_2 is as follows.

$$\begin{array}{|l}
\multicolumn{1}{l}{Inv} \\ \hline
\Delta State;\ \Delta Time \\ \hline
r' = max\{i : 0..j' \Leftrightarrow 1 \bullet L'i\} \\
L' = L \\
t' \Leftrightarrow t = 2(j' \Leftrightarrow 1) \\ \hline
\end{array}$$

A loop variant is $n \Leftrightarrow j$. The time constraint in the invariant is $t' \Leftrightarrow t = 2(j' \Leftrightarrow 1)$, since this is the time that has been taken by the loop after $j' \Leftrightarrow 1$ iterations. We claim that $A_2 \sqsubseteq \textbf{do}\ j < n \rightarrow A_3\ \textbf{od}$, where

$$\begin{array}{|l}
\hline
A_3 \\
\Delta State;\ \Delta Time \\
\hline
L' = L \\
j' = j + 1 \\
r' = max\{r, Lj\} \\
t' \Leftrightarrow t = 1 \\
\hline
\end{array}$$

To show this, we must verify the following proof obligations (where $B = j < n$). The first obligations require showing that the invariant is properly initialized (4), and that the invariant and guard B together establish the loop body (5). Showing these two steps is straightforward.

$$\text{pre}\, A_2 \wedge\ \textbf{skip} \vdash Inv \tag{4}$$

$$Inv \wedge \text{pre}\, A_2 \wedge (age(T(\textbf{do})) \wedge B')' \vdash (\text{pre}\, A_3)'' \tag{5}$$

where $T(\textbf{do})$ is the time required to evaluate the guard and conditionally branch, and $T(\textbf{od})$ is the time for the branch-back. In our example, these values are 0 and 1 respectively. Thus, $age(T(\textbf{do}))$ and $age(T(\textbf{od}))$ are

$$\begin{array}{|l}
\hline
age(T(\textbf{od})) \\
\Delta Time;\ \Xi State \\
\hline
t' \Leftrightarrow t = 1 \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
age(T(\textbf{do})) \\
\Delta Time;\ \Xi State \\
\hline
t' = t \\
\hline
\end{array}$$

In our example, only the branch-back takes non-zero time. But the refinement rules of [3] require us to include the time constraints for branch *into* the loop as well, in order for the proof to discharge.

The next two proof steps are as follows.

$$\text{pre}\, A_2 \wedge (Inv \,⨾\, age(T(\textbf{do}))) \wedge \neg B' \vdash A_2 \tag{6}$$

$$\text{pre}\, A_2 \wedge Inv \wedge (age(T(\textbf{do})) \wedge B')' \wedge (A_3 \,⨾\, age(T(\textbf{od})))'' \vdash Inv[\_'''/\_'] \tag{7}$$

Step (6) is straightforward, by the definition of *max* and of ranges. The final step, (7), showing that the invariant is maintained by the loop ('body liveness'), is more complex. The left hand side of the proof resolves to the conjunction of the following two schemas.

$$\begin{array}{|l}
\hline
\Delta State;\ \Delta Time \\
\hline
j = 1 \wedge r = L0 \\
r' = max\{i : 0..j' \Leftrightarrow 1 \bullet L'i\} \\
L' = L \wedge L'' = L' \\
j'' = j' \wedge t'' = t' \wedge r'' = r' \\
t' \Leftrightarrow t = 2(j' \Leftrightarrow 1) \\
j'' \leq n \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
\Delta State;\ \Delta Time \\
\hline
L''' = L'' \\
j''' = j'' + 1 \\
r''' = max\{r'', L''(j'')\} \\
t''' \Leftrightarrow t'' = 2 \\
\hline
\end{array}$$

while the right hand side resolves to

$$\begin{array}{|l}
\Delta State;\ \Delta Time \\
\hline
L''' = L \\
j''' \leq n \\
r''' = max\{i : 0..j''' \Leftrightarrow 1 \bullet L'''(i)\} \\
t''' \Leftrightarrow t = 2(j''' \Leftrightarrow 1) \\
\hline
\end{array}$$

and the proof obligation is then straightforward to discharge, by substitution and simple comparison.

In order to complete the proof, it is necessary to discharge termination obligations. We must therefore show that

$$\text{pre}\, A_2 \wedge Inv \wedge (age(T(\mathbf{do})) \wedge B')' \vdash n - j' > 0 \tag{8}$$

$$\text{pre}\, A_2 \wedge Inv \wedge (age(T(\mathbf{do})) \wedge B')' \wedge (A_3 \mathbin{\text{;}} age(T(\mathbf{od})))'' \vdash n - j''' < n - j' \tag{9}$$

Both obligations follow directly from the conjuncts in their hypotheses, which relate j'' to j' and j''' to j'.

Finally, we note that A_3 is a schema expression for the simultaneous assignment $j, r := j + 1, max\{r, Lj\}$, and so the calculation is complete.

3.3 Using the heterogeneous method

We now use the combination of Z and predicative programming. Predicative programming has techniques for dealing with timing constraints built-in. We propose to specify the state-based aspects of our system using Z, and the timing aspects using predicative programming. The predicative refinement rules, extended to heterogeneous specifications, will be used to refine the heterogeneous specification to an implementation that meets the time constraints.

The approach to constructing and manipulating heterogeneous specifications that we use was outlined in Section 2.3. We use predicative refinement rules on heterogeneous specifications taken from [9]. These refinement rules are syntactically very similar to standard predicative refinement rules, and they provably maintain the key properties of predicative refinement, e.g., the ability to refine specifications by parts and steps [9]. It is these properties that can make the heterogeneous approach to proving time constraints very attractive.

The problem specification in the heterogeneous case is very similar to that for the extended Z setting. The system state is

$$State \mathrel{\widehat{=}} \left[L : 0..n \Leftrightarrow 1 \to \mathbb{N};\ j : \mathbb{N};\ r : \mathbb{Z}_\infty \mid j \leq n \right]$$

We do not include time constraints in the Z schema. Instead, time constraints are expressed as predicates. The initial problem specification is $S \wedge Time$, where

$$S \mathrel{\widehat{=}} \left[\Delta State \mid L' = L \wedge r' = max\{j : 0..n \Leftrightarrow 1 \bullet Lj\} \right]$$

and $Time \mathrel{\widehat=} t' \Leftrightarrow t = 2n \Leftrightarrow 1$. In the refinement, we assume that an assignment statement and a tail-recursive call (equivalent to a loop branch-back) take one unit of time each, and that no other operations take any time.

The refinement proceeds as follows. Since we have composed S and *Time* via a predicative combinator, we use predicative refinement to implement the specification. We do this in two steps, as is standard in predicative refinement. We first refine the heterogeneous specification into code, ignoring all timing issues. Then, once complete, we prove that the time constraints written in the initial specification are satisfied by the implementation. We prove this last step by reusing the refinement tree of the first part, considering only time variables. Then, due to refinement by parts over predicative combinators, the composition of the timing and correctness proofs satisfies the original (timed) specification.

The refinement without time is straightforward. The first step is:

$$S \quad \Leftarrow \quad j, r := 1, L0.\ \ S1$$

where $S1$ is a Z schema, defined as follows.

$$S1 \mathrel{\widehat=} [\, \Delta State \mid L' = L \wedge r' = max\{r, max\{i : j..n \Leftrightarrow 1 \bullet Li\}\} \,]$$

This is trivial to prove using Rule 2. Note that the value of r is not constrained in the schema; it will be constrained by the recursive refinement. If we were developing a looping program, the value of r would have to be constrained. We next refine $S1$ into a two-branch **if** statement, using the *refinement by cases* rule of [6].

$$S1 \Leftarrow \mathbf{if}\ j = n\ \mathbf{then\ ok\ else}\ (j \neq n \Rightarrow S1)$$

(**ok** is the empty program that does nothing.) To prove the **then** branch, we use Rule 1, and in the process substitute n for j in the property of $S1$, so as to show that $S1[n/j]$ reduces to the schema that changes no variables. In doing this, we follow Section 2.3.1 and define the meaning of $j = n \Rightarrow S1$ as a predicate (which is justifiable in this case, because $S1$ is expressible in predicative notation). Substituting n for j gives us

$$\begin{array}{|l}
\hline S1[n/j] \\
\Delta State \\
\hline
L' = L \\
r' = max\{r, max\{i : n..n \Leftrightarrow 1 \bullet Li\}\} \\
\hline
\end{array}$$

The last line of the schema reduces to $r' = max\{r, \Leftrightarrow\infty\}$, which simplifies to $r' = r$. This schema can be implemented by **ok**. To prove the **else** branch, we apply the boolean laws of specialization and discharge, which prove the obligation in two lines. We omit the details.

The proof continues by refining the **else** branch.

$$(j \neq n \Rightarrow S1) \Leftarrow j, r := j + 1, max\{r, Lj\}.\ \ S1 \tag{10}$$

This refinement says that we implement $j \neq n \Rightarrow S1$ by first setting r to the maximum of the previous value of r and the value of Lj, and simultaneously increase j by 1. Then we behave like schema $S1$ again. This style of recursive refinement is a standard technique in predicative programming; it is done in lieu of developing looping programs with invariants (though in fact an invariant is present, but it is buried within the proof step).

To verify that this proof step is correct, we apply Rule 2 to the right-hand-side of (10), and show that it implies $(j \neq n) \Rightarrow S1$, by Rule 1. Applying Rule 2 to the right-hand-side of (10) gives the anonymous schema

$$\begin{array}{|l}
\Delta State \\
\hline
L' = L \\
r' = max\{max\{r, Lj\}, max\{i : j + 1..n \Leftrightarrow 1 \bullet Li\}\}
\end{array}$$

By definition of *max* and ranges, this simplifies to

$$\left[\, \Delta State \mid L' = L \wedge r' = max\{r, max\{i : j..n \Leftrightarrow 1 \bullet Li\}\} \,\right]$$

which is $S1$, and so the refinement holds by specialization.

Now for the timing proof. Since predicative refinement can be done by parts (and since S and *Time* are composed predicatively), we can prove that the timing constraints placed on the initial specification are maintained by the implementation, separate from the correctness proof. This requires us to show

$$Time \Leftarrow j, r, t := 1, L0, t + 1 \;.\; U \tag{11}$$

$$U \Leftarrow \textbf{if } j = n \textbf{ then ok else } Q \tag{12}$$

$$Q \Leftarrow j, r, t := j + 1, max\{r, Lj\}, t + 1 \;.\; t := t + 1 \;.\; U \tag{13}$$

for suitable timing predicates U and Q. Notice that the timing proof structure reuses the structure of the correctness proof.

We conjecture that

$$U = (j = n \Rightarrow t' = t) \wedge (j < n \Rightarrow t' = t + 2(n \Leftrightarrow j))$$
$$Q = j < n \Rightarrow t' = t + 2(n \Leftrightarrow j)$$

and now verify the steps. The proof of time is done in the predicative notation only; therefore, standard predicative rules apply. The proof of step (11) is straightforward by the substitution rule of [6]. Step (12) is also straightforward, applying the law of refinement by cases. The third step, (13), is the most complicated. (13) simplifies to, after three applications of the substitution rule

$$\begin{aligned} Q \Leftarrow\ & (j + 1 = n \Rightarrow t' = t + 2) \wedge \\ & \quad (j + 1 < n \Rightarrow t' = t + 2 + 2(n \Leftrightarrow j \Leftrightarrow 1)) \end{aligned}$$

which, by further simplification, is true. We conclude that the timing constraint *Time* is satisfied by the implementation. By monotonicity, $S \wedge Time$ is implemented, and satisfies its time bound.

We discuss the heterogeneous development, and compare it with the extended Z development, in Section 5.

4 Reasoning about Space

Many formal notations and methods have been constructed or extended so as to be able to reason about time for the purposes of developing real-time and reactive systems. There has been less work done on formal reasoning about space and space usage in formal notations and methods. One reason for this might be that space usage is typically a quantity associated with executing a specification, whereas formal notations are typically used for writing specifications that need not be immediately executable.

In this section, we compare Z extended with space variables with a heterogeneous combination of Z and predicative programming. We apply the two techniques to a simple problem, an abstraction of the Towers of Hanoi, to demonstrate some non-trivial space reasoning. We might expect different results than with a comparison in terms of time, mainly because reasoning about space (and extending notations to space) is much simpler than time, and only requires use of conventions.

4.1 The problem

The problem we want to solve is an abstraction of the Towers of Hanoi. We ignore the issue of putting disks on pegs, and instead concentrate on the issue of space use. The standard (exponential) solution to this problem is doubly recursive. Let x be the number of disks. The system state is $State \mathrel{\widehat=} [x : \mathbb{N}]$. A solution to our abstraction of the Towers of Hanoi problem is given in the guarded command language procedure *tower*.

```
procedure tower $\widehat=$ if $x > 0 \rightarrow$
                  $x := x \Leftrightarrow 1;\ tower;\ x := x + 1;$
                  $MoveDisk;$
                  $x := x \Leftrightarrow 1;\ tower;\ x := x + 1$
            [] $x = 0 \rightarrow$ skip fi
```

Procedure *MoveDisk* carries out the moving of disks from peg to peg (we leave its functionality unspecified, since it will not affect space calculations). To move the pile of disks, if there is at least one disk, first, ignore the bottom disk, remove the remaining pile, then reconsider all disks. Now move one disk (the one we previously ignored); then, again ignore the bottom disks, move the remaining pile, then reconsider all disks. If there are no disks, do nothing.

We assume that a recursive call to a procedure costs one unit of space (for holding a return address), and that *MoveDisk* and all other statements require no further space.

In proving bounds on space use, we will prove that the proposed solution satisfies a proposed space bound, i.e., a posit-and-prove approach.

4.2 Using the heterogeneous method

We first want to use the heterogeneous method to prove a bound on the maximum space used by the program *tower*. We specify the goal of the space calculation in Z.

$$TowerSpace \mathrel{\widehat=} [\,\Delta State;\ \Delta Space \mid m' = max\{m, s + x\}\,]$$

That is, we intend to show that the recursive procedure *tower* satisfies the space bound *TowerSpace*. To prove this, we start with specification *tower*, and in it replace the recursive calls to *tower* with calls to *TowerSpace*, performing the translation from Z to predicative notation behind-the-scenes. In doing so, we prefix the calls with changes in variables s and m; on returns from calls, we reset variables s and m. From this specification, we propose that (with some simplification)

$$\begin{array}{l} \mathit{TowerSpace} \Leftarrow \textbf{if } x > 0 \textbf{ then} \\ \qquad (x, s, m := x \Leftrightarrow 1, s + 1, \mathit{max}\{m, s + 1\}. \\ \qquad \mathit{TowerSpace}.\ \mathit{TowerSpace}. \\ \qquad x, s := x + 1, s \Leftrightarrow 1) \\ \quad \textbf{else ok} \end{array}$$

(*max* is predicative programming is a function of two arguments.) The proposed refinement is constructed directly from *tower*. Note that this is a heterogeneous specification, composing Z specification *TowerSpace* with predicative specifications using predicative combinators.

We verify the refinement by cases. The **else** branch is straightforward, and simply requires us to show that

$$\mathit{TowerSpace} \Leftarrow x = 0 \wedge (x' = x \wedge s' = s \wedge m' = m)$$

With the assumption that $m \geq s$ in $\Delta \mathit{Space}$, this implication is clearly *true*. For the **then** branch, we first simplify the body. The first two statements in the sequence of the **then** branch simplify to (by applying the substitution rule from [6], as well as *ZToPP*)

$$m' = \mathit{max}\{m, \mathit{max}\{s + 1, s + x\}\} \wedge x' = x \Leftrightarrow 1 \wedge s' = s + 1$$

The second two statements in sequence simplify to

$$m' = \mathit{max}\{m, \mathit{max}\{s, s + x\}\} \wedge s' = s \wedge x' = x$$

The sequence of these two specifications, followed by the simultaneous assignment $s, x := s \Leftrightarrow 1, x + 1$, is the assignment

$$m := \mathit{max}\{m, \mathit{max}\{s + 1, s + x\}\}$$

In order to complete the proof, we therefore must show that

$$\mathit{TowerSpace} \Leftarrow x > 0 \wedge m := \mathit{max}\{m, \mathit{max}\{s + 1, s + x\}\}$$

which is equivalent to showing

$$\mathit{TowerSpace} \Leftarrow x > 0 \wedge m := \mathit{max}\{m, s + x\}$$

We therefore must prove that

$$(m \geq s \Rightarrow m' = \mathit{max}\{m, s + x\} \wedge x' = x \wedge s' = s) \Leftarrow x > 0 \wedge m := \mathit{max}\{m, s + x\}$$

By expanding the assignment statement with its predicative semantics, and by applying the one-point rule three times, this reduces to *true*, and the proof is done.

4.3 Using Z

We now prove that the space bound is satisfied using the extended Z notation. We again use a *posit-and-prove* approach. We have a proposed solution to the Towers of Hanoi abstraction. We next specify a procedure *recSpace* that calculates the maximum space required for an implementation of *tower*. The body of *recSpace* is a Z schema.

$$\begin{array}{l} \textbf{procedure } \mathit{recSpace} \mathrel{\widehat{=}} \\ \quad \left[\, \Delta \mathit{State};\ \Delta \mathit{Space} \mid m' = \mathit{max}\{m, s + x\} \wedge s' = s \wedge x' = x \,\right] \end{array}$$

We will prove that *recSpace* is implemented by a recursive program. We claim that

$$\begin{array}{l} \mathit{recSpace} \sqsubseteq \textbf{if } x > 0 \rightarrow x, s, m := x \Leftrightarrow 1, s + 1, \mathit{max}\{m, s + 1\}; \\ \qquad\qquad\qquad\qquad\qquad \mathit{recSpace};\ \mathit{recSpace}; \\ \qquad\qquad\qquad\qquad\qquad x, s := x + 1, s \Leftrightarrow 1 \\ \qquad\qquad\qquad\ [] \; x = 0 \rightarrow \textbf{skip fi} \end{array}$$

The proposed refinement is obtained from *tower*, where each call to *recSpace* is prefixed with $s, m := s + 1, \mathit{max}\{m, s + 1\}$. The call to *MoveDisk* is removed since it will not affect space use. We prove the refinement by first simplifying the $x > 0$ branch of the guarded command. For each occurrence of *recSpace*, we substitute the schema definition of its body, and simplify. The first two statements in the sequence of the $x > 0$ branch is equivalent to the schema

$$\begin{array}{|l} \Delta \mathit{State};\ \Delta \mathit{Space} \\ \hline m' = \mathit{max}\{m, \mathit{max}\{s + 1, s + x\}\} \\ x' = x \Leftrightarrow 1 \\ s' = s + 1 \\ \hline \end{array}$$

The second two statements in sequence simplify to the anonymous schema

$$\left[\, \Delta \mathit{State};\ \Delta \mathit{Space} \mid m' = \mathit{max}\{m, \mathit{max}\{s, s + x\}\} \wedge s' = s \wedge x' = x \,\right]$$

The sequence of these two anonymous schemas and the schema for the simultaneous assignment $s, x := s \Leftrightarrow 1, x + 1$ is then

$$\left[\, \Delta \mathit{State};\ \Delta \mathit{Space} \mid m' = \mathit{max}\{m, \mathit{max}\{s + 1, s + x\}\} \wedge s' = s \wedge x' = x \,\right]$$

which is equivalent to $m := \mathit{max}\{m, \mathit{max}\{s + 1, s + x\}\}$. Therefore, we want to prove that

$$\begin{array}{l} \mathit{recSpace} \sqsubseteq \textbf{if } x > 0 \rightarrow m := \mathit{max}\{m, \mathit{max}\{s + 1, s + x\}\} \\ \qquad\qquad\quad\ [] \; x = 0 \rightarrow \textbf{skip} \\ \qquad\qquad\quad\ \textbf{fi} \end{array}$$

To prove the **if** branch, we must show that

$$recSpace \sqsubseteq x > 0 \land m' = max\{m, max\{s+1, s+x\}\}$$

To do so, we assume that $m \geq s$. Then, since $x > 0 \Rightarrow s + x \geq s + 1$, the proof goes through. To prove the **else** branch, we take a similar approach to proving

$$recSpace \sqsubseteq x = 0 \land \mathbf{skip}$$

and by assuming $m \geq s$, we see that $s + x = s$, and because $x = 0$, the proof also goes through. So we have shown that the maximum space bound on the implementation of *tower* is $max\{m, s + x\}$.

5 Discussion and Comparison

We can compare the use of extended Z with a heterogeneous combination of Z and predicative programming in a number of different ways, such as in terms of conciseness of the specifications, or in terms of the simplicity of the refinement rules of each method. Many forms of comparison are subjective. We therefore compare the two methods in terms of complexity of the method steps, i.e., refinement rules, *and* in terms of the complexity of the proofs.

With the extended Z method from [3], two changes must be made from standard Z in order to reason about time: specifications must include references to time variables and timing constraints; and, real-time refinement rules must be constructed from standard rules. The main extension with respect to the refinement rules is to introduce time constraints for programming language statements. The *structure* of refinements itself is not changed; one must still show safety and liveness conditions in order to prove correct refinements. However, the individual refinement rules are made more complicated (and longer, since we have to deal with larger specifications) by the addition of new terms, e.g., *age* schemas. As well, the refinement rules require some ‘place-holder’ terms, e.g., $age(T(\mathbf{do}))$, in order to properly discharge the proofs, even though the specific programming language entities that they describe require no time. Such placeholders, while certainly necessary for the proof, are unintuitive, and make proofs longer than is necessary. Specialization of the refinement rules would seem to be necessary in order to simplify the rules in cases where trivial time bounds are present.

Finally, we mention that the requirement to use refinement rules that contain both timing and state-based constructs simultaneously can make the process of proof in the extended method more complicated than in the heterogeneous method. It does not seem to be possible to separate proof of timing from proof of correctness when using the extended Z method, because this will require use of the schema calculus: a specification of behaviour will have to be (schema) conjoined with a specification of time constraints. Refinement is not in general monotonic over the schema calculus conjunction combinator [11], and so separating proof of correctness from proof of timing is not generally possible.

In the heterogeneous method applied to timing, the refinement rules that are used are simple, generalized rules from [6]. For those rules that are added in order to carry out the proof of partial correctness (but not the proof of timing)—e.g., refinement of a Z schema by a predicate—the additions are small and used in the same manner as existing predicative refinement rules. The added rules are straightforward and for the most part syntactic generalizations of the refinement rules from [6], with minimal restrictions on their use. Importantly, with the heterogeneous method, proofs of correctness and timing can be separated, due to monotonicity properties. This lets us carry out the proof of timing in predicative notation.

If we were to look solely at using extended Z or the heterogeneous approach for writing specifications (with time), we would find many similarities. The initial specifications that we wrote when applying extended Z and the heterogeneous approach were similar in size and complexity. However, the specifications that were *constructed* during the refinement were more complex with extended Z. With the heterogeneous approach, the specifications that were constructed were no more complex than what occurs when writing standard predicative specification, which may be important, in particular, when doing automated proof.

Differences between using the two approaches is less clear when we consider space reasoning. In applying Z method, refinement rules do not have to be changed in order to reason about space, because we treat space as new state components. However, specifications are made modestly larger because of the need to add procedure interfaces to proposed solutions. This is due to the fact that non-trivial space behaviour arises only with recursive procedures. In the heterogeneous approach, we can avoid adding procedure interfaces because of the capability of using recursive refinement techniques [6] on heterogeneous specifications, which negates the requirement to explicitly introduce recursive procedures.

In the heterogeneous method, one new refinement rule must be introduced, for refining Z schemas by predicative specifications. This rule is a simple generalization of standard predicative refinement. But in the extended Z method, refinement as described in [12] suffices for proof of space bounds. Therefore, we might conclude that the extension with respect to space favours the extended Z approach rather than the heterogeneous approach, at least in terms of complexity or any requirements to adapt proof rules to heterogeneous notations. When we look at the proofs of space bounds, in both the extended Z and the heterogeneous case, we see that the proofs are of similar complexity, and take similar steps. Therefore, we might also conclude that the two approaches are too similar in their application to make any overall distinction. Part of the reason for this similarity is due to the fact that extension or notation integration for reasoning about space is much more straightforward than for time: detailed new proof rules do not have to be constructed, and existing proof rules do not generally have to be modified.

In general, then, we found that using the extension of Z to reason about time was more complex and intrusive than using the heterogeneous combination of Z and predicative programming. This intrusiveness manifested itself in terms of both writing specifications, and in terms of proof. On the other hand, we found that using the extension of Z to reason about space was no more complex than using a heterogeneous combination of Z and predicative programming. This was primarily an artifact of reasoning about space constraints, which is simpler to do than reasoning about time constraints, because there are fewer ways to effect a change in space use within a program.

This suggests that whether we should prefer an extension approach over a heterogeneous approach will depend on the task to which we want to apply the method. The timing case study has shown that a heterogeneous approach can be simpler to use than an extension approach. But the space case study has also shown that in some situations—e.g., when reasoning about a system artifact that does not broadly require changes in refinement rules—an extension approach can be just as straightforward to use as a heterogeneous approach.

As an alternative to adding Z to predicative programming for proving time and space bounds, we might consider the reverse situation: where we take predicative programming and add Z notations to it. We could then use the resulting heterogeneous notation to reason about time, space, and refinement. But predicative programming already has built-in mechanisms for reasoning about time and space, and is a wide-spectrum design calculus with a simpler notion of refinement — boolean implication — than Z. Therefore, for this purpose (reasoning about time, space, and refinement), it seems to be redundant to add Z to predicative programming; there of course may be other valid reasons for the addition.

6 Conclusions

We have briefly compared the use of an extension approach to formal methods with a heterogeneous approach to formal methods. Applying Z, extended with specification and reasoning for time and space, and comparing it with applications of Z combined with predicative programming, has been carried out on small examples. We found that the heterogeneous combination of Z and predicative programming was easy to use for timing specification and reasoning, resulting in shorter proofs than was the case with extended Z. And we found that the extension and heterogeneous approaches were similar in complexity for specifying and reasoning about space usage. We therefore can conclude that there exists a situation where heterogeneous notations are useful — and are more useful than an extended notation.

More work remains to be done on comparing extension and heterogeneity. We have only carried out a small pair of case studies here. It seems likely that there will be special cases of applications for which our conclusions will not hold, and discovering such special cases will be of interest, especially with respect to determining fundamental system properties for which extension is simpler to use than heterogeneity (or vice versa). As well, it would be useful to attempt other case studies, beyond time and space. Extension with respect to semantics, e.g., such as is done in [5], would be interesting to study and compare as well.

Acknowledgements. Thanks to Rick Hehner, Jonathan Ostroff, and Phil Brooke for their comments. Special thanks to the anonymous referees for their detailed suggestions, which have improved the quality and presentation of the paper. This work was supported with the help of the National Science and Engineering Research Council of Canada.

References

1. P. Baumann and K. Lerner. A Framework for the Specification of Reactive and Concurrent Systems in Z. In *Proc. 15th Conference on Foundations of Software Technology and Theoretical Computer Science*, LNCS 1026, Springer-Verlag, 1995.
2. J.-M. Bruel, A. Benzekri, and Y. Raymaud. Z and the Specification of Real-time Systems. In *Proc. 7th Int. Conf. on Putting into Practice Methods and Tools for Information System Design*, IRIN, 1995.
3. C.J. Fidge. Real-time Refinement. In *Proc. FME '93*, LNCS 670, Springer-Verlag, 1993.
4. J. Grundy. Predicative Programming—A Survey. In *Proc. Formal Methods in Programming and Their Applications*, LNCS 735, Springer-Verlag, 1993.
5. E.C.R. Hehner and A.J. Malton. Termination Conventions and Comparative Semantics, *Acta Informatica*, 25 (1988).
6. E.C.R. Hehner. *A Practical Theory of Programming*, Springer-Verlag, 1993.
7. E.C.R. Hehner. Formalization of Time and Space, submitted.
8. C.C. Morgan. *Programming from Specifications*, Prentice-Hall, Second Edition, 1994.
9. R.F. Paige. A Meta-Method for Formal Method Integration. In *Proc. FME '97*, LNCS 1313, Springer-Verlag, 1997.
10. J.M. Spivey. *The Z Notation: A Reference Manual*, Prentice-Hall, 1989.
11. N. Ward. Adding specification constructors to the refinement calculus. In *Proc. FME '93*, LNCS 670, Springer-Verlag, 1993.
12. J.B. Wordsworth. *Software Development with Z*, Addison-Wesley, 1992.
13. P. Zave and M. Jackson. Where do operations come from? An approach to multi-paradigm specification, *IEEE Trans. on Software Engineering*, 12(7), July 1996.