FunZ: An Intermediate Specification Language

LINDA B. SHERRELL and DORIS L. CARVER

Department of Computer Science, Louisiana State University, Baton Rouge, LA 70803, USA
Email: sherrell@caddo.astate.edu carver@bit.csc.lsu.edu

During the last few years, the field of software engineering has witnessed an increased interest in formal methods and software reuse. At the same time, functional programming languages, often espoused as rapid prototyping tools, have begun to enjoy more mainstream usage. Assuming that these trends continue, software developers will need improved methods to transform existing specifications into functional implementations. In this paper, we discuss the intermediate specification language FunZ, an integral part of a methodology to produce purely functional programs from Z specifications. To illustrate the concepts of FunZ, we specify the design of a simple software system using both the Z notation and that of FunZ. FunZ itself is best described as an extension of Haskell, yet the language also retains a Z-like flavour in that it contains notational conventions similar to those of standard Z and several object-oriented variants. In addition, software design with FunZ parallels the activity in Z except that each step has functional overtones to better accommodate a final implementation in a purely functional language.

Received September 16, 1994; revised May 5, 1995

1. INTRODUCTION

As the complexity and cost of software have continually increased, researchers have attempted to alleviate the software crisis. Several approaches have been initiated. Formal methods have been introduced into the software lifecycle. Support tools have been implemented to help eliminate human error. Programming paradigms other than the conventional, imperative model have been advocated.

One such paradigm is functional programming. Functional languages offer powerful abstraction facilities, which allow algorithms to be expressed in a natural and concise fashion. As a sample, higher-order functions, list comprehensions, pattern matching and lazy evaluation are devices available to the modern functional programmer.

In the past, although functional languages were frequently used in the realm of rapid prototyping [1], [2] and [3], mainstream applications were somewhat limited. However, over the last few years, this trend has begun to change. For example, Lolita [4], a large system for natural language processing, was developed completely in Miranda™ [5]. Clio and Spectool, two verification tools designed at Odyssey Research Associates, are based on Caliban, a Miranda-like language. These tools played a major role in the first phase of verifying a hardware component for the Fault Tolerant Parallel Processor [6]. More recently, Haskell [7] has been used to program a significant portion of a software system for automatic speech recognition [8]. Additional real-world applications can be found in [9].

The use of functional programming languages should become even more prevalent in the future with ongoing improvements to compilers and advances in computer architecture. At the same time, the current emphasis on formal methods and software reuse is likely to continue. Therefore, we have designed the intermediate language FunZ and an associated methodology to aid software developers in deriving purely functional programs from existing Z specifications.

Several languages have influenced the design of FunZ including Larch [10], COLD-K [11], Extended ML [12] and Object-Z [13]. Ideas from [14] have also played a role in its development. However, FunZ is best described as an extension of Haskell with a Z-like flavour because it preserves many of the notational conventions of standard Z and several object-oriented variants. In addition, software design with FunZ is similar to that with Z, except that each step has functional overtones in order to provide a better match with a final implementation in a purely functional language.

Since FunZ combines features from both Z and Haskell, the language is of benefit to either Z or Haskell aficionados. In particular, for those software developers who know Z but are less familiar with functional programming, FunZ provides a bridge between Z specifications and functional implementations. Similarly, for those Haskell programmers inexperienced with Z, FunZ is an attractive design language because it is a simple extension of Haskell.

In addition, using FunZ for design specifications as opposed to Z offers several advantages to both groups. First, code fragments are derived and verified earlier in the development process. Consequently, the total cost for a software project should decrease. Second, FunZ allows a developer to prove properties about the system...
design using either the Z notation or the programming language Haskell. This freedom means that the notation most applicable to the problem may be selected. And finally, the methodology surrounding FunZ provides a framework for recording design decisions that is useful for future maintenance.

1.1. Why Z and Haskell?

Throughout their respective histories, specification languages and functional programming languages have been closely associated. Hope [15], an early functional language, includes parameterized modules that are based on those of CLEAR [16], the first algebraic specification language. At least two prototyping languages, me too [1] and SAMPL,e [17], combine features from Miranda and VDM, the first formal method to use the model-oriented approach. And now, as a means of bridging the gap between Z specifications and Haskell implementations, FunZ continues this tradition by integrating features from Z and Haskell.

Z [18] is a model-oriented specification language based on typed set theory. Since its initiation in the 1980s, Z has become increasingly popular. This popularity is evidenced not only by the numerous conference and journal articles devoted to software development in Z, but also by the fact that several textbooks (e.g. [19], [20], [21] and [22]) are now available on the subject. We have selected Z as the initial specification language, in large part because of its widespread acceptance but, more importantly, because of its strong mathematical basis.

Haskell [7] is a language that closely resembles Miranda, but also includes additional features such as array comprehensions and type classes. Haskell was designed as a general purpose language appropriate for teaching, research, and building large systems. Although the language is relatively new, we have chosen it as the implementation language because of its endorsement by the functional programming community as a standard for non-strict, purely functional languages.

1.2. Why an intermediate language?

Refinement is a well-established principle when constructing imperative programs from model-oriented specifications. By refining an abstract specification to one that is more concrete, a software developer narrows the gap between specification and implementation. In particular, the problem of proving an implementation correct with respect to its specification is converted into two smaller problems: (i) proving the concrete or lower level specification is consistent with the abstract version; and (ii) verifying the code against the concrete specification. Furthermore, by following refinement guidelines, the developer documents his or her design decisions.

A popular method [22], successfully employed at IBM Hursley, uses Z for specification and design and Dijkstra's guarded command language for algorithm development. An alternate approach is to translate all Z schemas to specification statements in the refinement calculus [23] and then apply the laws of the calculus to derive guarded command programs. In [24] the translation occurs after data reification in Z, while [25] initiates the change in notation one step sooner by converting the abstract specification schemas to the refinement calculus.

We have experimented with changing from Z to Haskell at different stages in the software life cycle. We began with a traditional approach, that is, refining Z specifications to Z designs and then to code. Initial designs focused on the list and its standard list functions, while later designs targeted additional data types, most notably the array. The respective translations revealed a natural correspondence between sequences and functions in Z and lists and arrays in Haskell. The interested reader can find detailed descriptions of some of these designs and translations in [26] and [27].

To formalize this work, we needed to solve two problems: (i) adapt the traditional refinement approach [18] in order to give it a more functional flavour and (ii) define a general set of transformation rules for converting Z designs to Haskell. We considered the first problem to be relatively easy, while the second would require some ingenuity. While working on both problems, we developed a collection3 of mappings of the following form:

\[
\begin{align*}
\text{Z set operators} & \rightarrow \text{Z sequence operators} \\
& \rightarrow \text{Haskell list functions}
\end{align*}
\]

As an example, if we consider lists without duplicate elements, set difference \(\setminus\) maps to range subtraction \(\rightarrow\), which in turn maps to list difference \(\text{\setminus}\). Given the close relationship between sequences and lists, it was often possible to perform a direct translation from sets to lists by using a modified refinement approach. Thus arose the idea of an intermediate language, which would combine features from Z and Haskell, to assist the software designer in translating Z specifications to Haskell programs. FunZ and its associated methodology are the result of these early efforts.

As far as we know, our research is the first to use an intermediate language when translating Z specifications to functional implementations. However, Wood and Place [28] describe a formal development method in which Z specifications are initially transformed to ANNotated Ada (Anna) [29] and then to Ada code. Although the transformation rules of their method are language independent, the advantage in targeting Ada programs is twofold: the existing specification language Anna serves as the intermediate language and the Ada tool set simplifies much of the translation process.

In a similar fashion, FunZ targets Haskell, but the language and associated methodology are applicable to other functional languages by making some simple syntax changes and using the appropriate function names. For instance, ++ separates the constructors of

---

3 A similar collection was developed for arrays.
an algebraic datatype in Hope, whereas \( \mid \) is the correct
Haskell syntax. As another example, \( \_\_\_\_ \) is the list difference
operator in Miranda, while \( \setminus \) is used in Haskell.

By using a case study, this paper illustrates how one
1 can translate an abstract specification written in Z to
Haskell code via the intermediate language FunZ. Furthermore, as a means of comparison, a correspond-
ing Z design is presented along with the FunZ
specification. Throughout the paper, we assume a
working knowledge of Z [18], as well as some familiarity
with the functional programming paradigm [30]. How-
ever, knowledge of Haskell is not required as its concepts
are described where needed. For readers who feel an
introduction to Haskell may be helpful we recommend
[31] or [32].

The organization of the paper is as follows. The next
section discusses other research that combines functional
programming and formal methods. Section 3 gives a
brief introduction to FunZ, while Section 4 contains the
actual case study. Comparisons with other work and our
conclusions appear in Section 5.

2. RELATED RESEARCH

Much of the previous work linking formal specifications
and functional programming has centered around
prototyping. Research has ranged from the simple
animation of specifications to the design of new
programming languages that integrate essential
characteristics of formal methods and functional
languages.

Descriptions of how a software designer can animate
specifications using Miranda appear in [33] and [19].
More recently, [34] discusses a mechanical translation of
VDM specifications into Standard ML [35]. Note that,
strictly speaking, Standard ML (SML) cannot be
classified as a purely functional language owing to its
references (special values similar to the variables in an
imperative language) and its I/O facilities, neither of
which is referentially transparent.

As previously mentioned, the prototyping languages
me too [1] and SAMPAE [17] borrow features from
Miranda and VDM. Compared with animation
techniques, me too forms the foundation of a complete
software design methodology, while SAMPAE is an
integral part of an interactive prototyping environment.
These languages offer the advantage that one medium is
used for both specification and prototyping. However,
SAMPAE mixes the functional and imperative
paradigms by including traditional control structures,
assignment statements, and updatable variables.

Another approach to software development is the
Extended ML methodology [36,12]. Extended ML is a
wide-spectrum language, founded on the principles of
algebraic specifications, which supports the formal
development of Standard ML (SML) programs from
high-level specifications to final implementations within
a single framework. An advantage of the methodology is
that Extended ML contains only two new language
constructs: the axiom and the place-holder \(?\). The
minimum number of constructs that are required is
primarily the result of the rich module facility already
present in SML.

Software developers may soon have the option of
using the two-tiered approach of Larch [10] when
constructing SML programs. The first draft of a Larch
interface language for SML, Larch/ML, appears in [37].
Its designers claim that one benefit from the language is
that it can be used to describe the semantic properties of
basic data types, thus enhancing the formal semantics of
SML. Furthermore, Extended ML is currently unable to
handle references and assignments, while Larch/ML is
suitable for specifying such features.

At British Telecom Research Laboratories, con-
structive Z specifications have been converted into
Lazy ML programs, which were used as prototypes.
Due to the inefficiency of these initial programs, they
were then transformed into more efficient functional
programs using the methodology of Burstall and
Darlington [38]. A case study illustrating the technique
appears in [39]. This research differs from ours in two
important ways. First, in the work of Johnson and
Sanders, a primary goal is to obtain executable code as
soon as possible; the code then drives the design. Second,
the transformation of the functional program into an
equivalent imperative program\(^4\) is seen as part of the
software development process.

3. OVERVIEW OF FUNZ

The primary objective in designing FunZ was to produce
an intermediate specification language to assist soft-
ware developers in transforming Z specifications to
Haskell programs. Specific design constraints were as
follows:

(i) The language should be a simple extension of
Haskell.

(ii) The language should be conducive to specifying the
characteristics of a purely functional programming
language.

(iii) The language should preserve the notational
conventions and structuring facilities of Z.

Furthermore, an associated methodology, patterned
after the Hursley method of IBM [22], was to be
developed simultaneously. Note that a description of
the methodology encompassing FunZ appears in [40].

To satisfy the design constraints of FunZ, features
from both Haskell and Z were integrated. The result is a
specification language that looks like an extension of
Haskell, yet whose semantics parallels that of Z. In
addition, for those designers and implementors who
prefer functional programing languages, FunZ allows
software developers to model state operations more

\(^4\) Johnson and Sanders had not attempted this final translation
procedure when their paper was accepted to the 4th Z User Workshop.
closely to the way that they will be implemented in a final functional program. (As one example, see the definition of the modifies tuple later in this section.)

Similar to Z, the prominent language feature of FunZ is the schema, albeit with a name change. To emphasize the fact that these building blocks serve as a bridge between Z specifications and functional programs, FunZ schemas are called spans. Moreover, the graphical, box-like notation has been abandoned in favour of a syntax patterned after Haskell modules.

As in Z, each FunZ design consists of state and operation spans that refine the corresponding schemas of the Z specification. Unlike Z, the structure of a state span differs significantly from that of an operation span. Motivation for these differences as well as other design decisions are described below. Note that Figure 1 contains a template for a typical state span.

FunZ requires the name of each state span to match the name of the Z schema on which it is based, thus providing an automatic means of schema inclusion. Each state span consists of four components or parts: declaration, invariant, relation, and initialization. As their respective names suggest, the declaration part contains the necessary declarations, while the invariant part describes the invariant on the concrete state. These two components are analogous to the declaration and predicate parts of the state schema in Z. Meanwhile, the relationship between concrete and abstract states, commonly known as the retrieve relation, appears in the relation part. Finally, the initialization part replaces the Z schema that denotes an initial concrete state. In short, the state span groups all items associated with the concrete state into a single specification unit. When compared to Z, this means that several declarations do not have to be repeated. More importantly, it makes the design clearer to have associated parts collected into the same span.

As in Z, a typical operation span consists of a declaration part and a predicate part. Figure 2 delineates the individual components of the declaration part to emphasize the differences between an operation span and its Z counterpart. The most important change is in the modifies declaration, as it represents not only an alteration in syntax, but also one in semantics. Meanwhile, the predicate part is comparable to that of Z.

The modifies clause supersedes the Delta (Δ) convention of Z; all state variables that are allowed to change must be listed explicitly. The "list" itself is known as the modifies tuple. As an example, consider the following modifies clause that forms a part of span Testok, which is described in Section 4.2:

\[
\text{modifies Class (ns, ts)}
\]

The declaration denotes that all the variables and predicates of spans Class and Class' are visible, yet only the values of variables ns and ts may vary. If no state values should change as a result of an operation, then the modifies tuple is simply written as ( ). In other words, the expression modifies(Schema_Name) ( ) replaces the Xi (Ξ) convention of Z.

Another important point concerning the modifies clause has to do with the semantics of functional programming. Recall that there is no assignment statement in a purely functional language; the state must be passed around explicitly with parameters. Therefore, what is meant by a variable changing is that the variable must be passed as a parameter to a function and a new value must be returned as a result of this function call. The notation of the modifies tuple is meant to reflect that the variables will need to be actual parameters in a Haskell implementation.

4. A CASE STUDY

This section traces the development of the Test operation, one component of the class manager's
assistant, from an initial specification written in Z, through a detailed design expressed in FunZ and, finally, to Haskell code. Our intention in selecting this standard example as a case study is to allow the reader to focus on the development process itself, rather than the details of a complex application.

4.1. Requirements

The following description of the class manager’s assistant and the Test operation are from [22]. The corresponding initial Z specification can be found in Appendix A.

Assistant description

A computerized class manager’s assistant is required to keep track of students enrolled on a class, and to record which of them have done the midweek exercises. When a student applies for a class, he or she will be enrolled on it, unless it is full. Such a student will be presumed not to have done the exercises. When a student completes the exercises, the fact is to be recorded. Students may leave a class even if they have not done the exercises, but only the students who have done the exercises are entitled to a completion certificate.

Test operation

This operation records that a student has done the exercises, or warns if the student is not enrolled or has already done the exercises.

4.2. Specification with FunZ

The process of constructing a detailed specification in FunZ parallels a development in Z with the primary difference being that Haskell code fragments constitute a major portion of the final FunZ design. To help clarify the overall procedure, we next present a general, methodological guideline.

(A) Translate the global variables to their FunZ equivalents.

(B) Define a FunZ state span SS that is equivalent to the schema representing the abstract state.

(1) Choose appropriate Haskell data structures or user-defined types to represent the objects of the abstract state. Add the necessary type declarations to SS.

(2) Denote any necessary constraints on the concrete objects. Place the new expressions in the invariant part of SS.

(3) Define a retrieve relation, which maps the concrete FunZ objects to the abstract Z objects, and place this in the relation part of SS. The mapping must be subjective as this function will be used to translate the abstract objects of the Z specification to concrete objects in FunZ.

(4) Specify a concrete initial state in the initialization part of SS. Add the appropriate parameters to the init tuple.

(C) For each Z operation schema, define a corresponding FunZ operation span OS.

(1) Transfer the input and output variables to the declaration part of OS.

(2) Translate each predicate to an equivalent FunZ expression. Install the new expressions in the predicate part of OS.

(3) Put all state variables whose values should change into the modifies tuple of OS.

As we develop the FunZ specification, we compare the evolving spans to a similar design in Z. Accompanying each step is a table depicting the Z text on the abstract state, along with the corresponding text for concrete designs in both FunZ and Z. For the convenience of the reader, the definitions of all Haskell functions from the FunZ specification appear in Appendix B.

Finally, Appendix C contains the complete FunZ specification for the class manager’s assistant. Although we do not include the corresponding Z specification, the reader can easily construct the respective schemas by referring to the concrete Z phrases described below.

Step A: translate global variables

The developer has the option of choosing a representation for any basic set, which was previously defined on the abstract state. In this case, we have decided to postpone the decision so we merely make the appropriate syntax changes. Global constants and enumerated types are easily translated. The variables in the FunZ enumerated type must begin with a capital letter since they correspond to data constructors in Haskell.

In the Z design, this step is not necessary because the global variables are already represented in the Z notation (Table 1). However, a Z designer must eventually make similar translations when he or she develops the program code.

---

2 This classic example appears frequently in the literature and was first published in [41].
Step B: define a state span

Steps B1–B4 correspond to data refinement in Z. In other words, data structures of the target programming language replace the objects in the abstract data space. In a Z design, Z constructs must model primitive types or ADTs of the implementation language, while a FunZ document contains actual Haskell types. Due to the expressive type system of Haskell, the software designer can create high-level data types, similar in form to the abstract Z types, or use built in types such as lists, tuples, and arrays.

Step B1: choose Haskell data structures to represent abstract objects

Although the sets in a model-oriented specification can be implemented by means of an abstract data type [42], we base our design on the Haskell list and its associated list functions. In particular, the set of enrolled students is represented by a list of students who have completed the exercises (ts) and a list of students who have not (ns). Because the arrangement of students in each list is unimportant, the additional ordering information offered by lists as compared to sets can be disregarded. Furthermore, in this example, there is no need to worry about removing duplicate elements after a list insertion since a student is only added to the group of tested students if he or she is not already a member.

Meanwhile, in the Z design (see Table 2), sequences model the Haskell lists. This is a good match as many of the sequence operators from the Z Mathematical Toolkit [18] correspond to basic list functions.

Step B2: place necessary constraints on the concrete objects

Next, we give an informal description of the FunZ expressions that will form the invariant part of the state span Class (see Table 3). Since the concatenation ++ of lists ns and ts corresponds to the set of enrolled students, its length cannot exceed the maximum size of the class. Unlike the invariant on the abstract state, the concrete invariant no longer needs a predicate such as tested ⊆ enrolled since, by construction, the list (ns ++ ts) contains each element of ts.

Lists ns and ts should not contain repetitions since each list represents a group of students. This is easily specified by using the predefined Haskell function nub, which removes duplicate elements from a list. Furthermore, ns and ts should be disjoint since it is impossible for a student to have been tested and not tested at the same time. The last predicate in the FunZ column designates this final requirement.

Symbols ==, &&, 'elem', and 'notElem' are infix operators that correspond to the =, ∧, ∈, and ∉ of Z. In FunZ, Haskell operators are used in predicates, while the logic operators of Z are reserved for combining spans. The reason for using Haskell in each of the predicates is that some software developers may want to implement the invariant. Even when the state invariant is not executed, describing the constraints in Haskell helps the developer to understand the chosen data structure better. As a final note, the keyword implies is not a part of Haskell. This boolean-valued function has been added to FunZ to handle implication, because => is a reserved operator in Haskell.

An important distinction about Haskell's equality operator is that it is one of the methods defined on the type class Eq. The operator can only be applied to objects of the same type, and the corresponding type must be an instance of Eq. Since the list data structure is a predefined instance of the equality class, the FunZ expressions in Table 3 are also legal in Haskell.

The concrete Z and FunZ invariants are quite similar. With respect to the first predicate, sequence operators # and * correspond to the predefined Haskell functions length and (++) . Since sequences are partial functions, the range (ran) operator can be used to express the fact that these particular sequences do not contain repetitions. In other words, the length (#) of each sequence should equal the number of elements in its range. An alternate way of specifying this information would have been to declare sequences nottested and
### TABLE 4. Retrieve relations

<table>
<thead>
<tr>
<th>FunZ</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>abmap : : [Student] —&gt; P Student</td>
<td></td>
</tr>
<tr>
<td>enrolled = set (ns + + ts)</td>
<td>enrolled = ran (testlist ~ nottested)</td>
</tr>
<tr>
<td>tested = set ts</td>
<td>tested = ran testlist</td>
</tr>
</tbody>
</table>

### Step C: define operation spans

Observe that this step denotes an iterative process. Here, we only describe the construction of an operation span for the successful component of the Test operation, namely Testok, since the spans for the error conditions are relatively straightforward. However, the definitions for all the operation spans are included in Appendix C. In particular, the definition for span Test is simply the disjunction of spans Testok, AlreadyTested, and NotEnrolled.

### Step Cl: translate the input and output variables

When transferring the input and output variables to the FunZ span, only a slight change in syntax is required (Table 6). Double colons replace the single colons of Z to match the typing declarations in Haskell. In addition, keywords in and out are used to indicate input and output variables, since the standard Z suffixes ? and ! are not allowed in Haskell identifiers. Meanwhile, in Z the variables are simply copied from the abstract schema to the concrete one.

### Step C2: translate the predicate part

To obtain predicates on the concrete state, one uses the retrieve relation (see step B3) as a guide. Each abstract object in the Z operation schema is replaced with its concrete representation. By using the properties of set theory and theorems provided with FunZ, one then attempts to simplify the resulting expressions. The strategy is to obtain expressions where Z operators map to Haskell functions.

To help clarify the subsequent translation, Table 7 contains the transformation rules for this example. The first three rules specify syntax changes for input and output variables as well as enumerated types, whereas rules 4 and 5 relate set membership symbols to Haskell operators. Finally, rule 6 requires that list xs contain no duplicates in order for the transformation to hold.

A description of the translation process for each of the predicates from schema Testok (see Appendix A or column one of Table 8) follows. Beginning with the first abstract predicate, the derivation of a concrete equivalent is:

\[
s? \in \text{enrolled} \Rightarrow \\
s? \in \text{set (ns + + ts)} \Rightarrow \text{retrieve function} \\
s \in \text{set (ns + + ts)} \Rightarrow \text{rules 1, 4, and 6}
\]

### TABLE 5. Initial states

<table>
<thead>
<tr>
<th>Z (Abstract)</th>
<th>FunZ (Concrete)</th>
</tr>
</thead>
<tbody>
<tr>
<td>enrolled' = 0</td>
<td>(ns, ts)</td>
</tr>
<tr>
<td>tested' = 0</td>
<td>([], [])</td>
</tr>
<tr>
<td>nottested' = {}</td>
<td>{}</td>
</tr>
<tr>
<td>testlist' = {}</td>
<td>{}</td>
</tr>
</tbody>
</table>
TABLE 7. Transformation rules

<table>
<thead>
<tr>
<th>Rules</th>
<th>Z</th>
<th>FunZ/Haskell</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>varname?</td>
<td>varname</td>
</tr>
<tr>
<td>2</td>
<td>varname</td>
<td>set xs</td>
</tr>
<tr>
<td>3</td>
<td>enumid</td>
<td>Enumid</td>
</tr>
<tr>
<td>4</td>
<td>∈</td>
<td>'elem'</td>
</tr>
<tr>
<td>5</td>
<td>\≠</td>
<td>'notElem'</td>
</tr>
<tr>
<td>6</td>
<td>set xs</td>
<td>xs</td>
</tr>
</tbody>
</table>

Similarly, the second predicate translates as:

\[ s? \notin \text{tested} \Rightarrow \]
\[ s? \notin \text{set ts} \Rightarrow \text{retrieve function} \]
\[ s \not\in \text{notElem ts} \]

By jointly considering the above results, an additional simplification is possible. Observe that student \( s \) is a member of the concatenation of lists \( ns \) and \( ts \), but is not contained in \( ts \). Therefore, \( s \) must be an element of \( ns \) or, in FunZ, \( s \not\in \text{elem} ns \). Since this new, simpler predicate implies

\[ s \not\in \text{elem} (ns ++ ts) \]

we select it as the first predicate on the concrete state.

Continuing in the same fashion, the derivation steps for the third concrete predicate are:

\[ \text{tested}' = \text{tested} \cup \{s?\} \Rightarrow \]
\[ \text{set ts}' = \text{set ts} \cup \{s?\} \Rightarrow \text{retrieve function (twice)} \]
\[ \text{set ts}' = \{s?\} \cup \text{set ts} \Rightarrow \text{commutativity of } \cup \]
\[ \text{set ts}' = \text{set (s?;ts)} \Rightarrow \text{definition of set} \]
\[ ts' = s:ts \]

Note that the commutative law was applied so that the new student would be added to the front of the list, an O(1) operation, rather than the end of the list, which requires O(n) time. Furthermore, the substitution of list \( (s:ts) \) for \( \text{set (s:ts)} \) was valid because student \( s \) is not an element of \( ts \) according to the second concrete predicate.

Now consider the fourth abstract predicate. Applying the retrieve relation twice, produces the result:

\[ \text{set (ns' ++ ts')} = \text{set (ns + + ts)} \]

An equivalent predicate can be derived by applying laws of set theory and various theorems provided with FunZ. However, we present a more informal argument that assumes the previous concrete predicates and the invariant.

\[ (s \not\in \text{elem} ns \Rightarrow s \not\in \text{notElem ts}) \]
\[ \&\& (s \not\in \text{elem} ts \Rightarrow s \not\in \text{notElem} ns) \]

from span Class. An intuitive idea of the reasoning is as follows: after student \( s \) is inserted at the front of list \( ts \), \( s \) must be removed from \( ns \) to preserve the invariant. Note that \( \setminus \) is the Haskell operator for list difference. In particular, the expression \( xs \setminus ys \) returns the list \( xs \) with only the first occurrence (if any) of each element in \( ys \) removed. Therefore, \( ns \setminus \{s\} \) has the desired effect of removing student \( s \) from \( ns \), the group of non-tested students.

Finally, to derive a concrete equivalent of the last abstract predicate, the output response, one simply applies rules 2 and 3. In other words, the designer drops the explanation point from the output variable and replaces the enumerated type value in \( Z \) with its corresponding Haskell data constructor.

Note that span Test ok (see Appendix C) does not include both the predicates \( s \not\in \text{elem} ns \) and \( s \not\in \text{notElem} ts \) in its predicate part. Only the first predicate appears, since the second can be derived from the first and the invariant on span Class.

Meanwhile, to translate the abstract predicates to concrete predicates in \( Z \), one follows a similar approach to that described above. In particular, the retrieve function (see the \( Z \) predicates in step B3) serves as a translator for the objects on the abstract state. Rather than present the derivation steps, we simply include the final predicates in Table 8. Note that the removal of student \( s? \) from the set of nottested students is denoted by the range subtraction operator \( \setminus \).

As a final comment, although the translation processes in \( Z \) and FunZ are very similar, there are two major differences in the resulting designs. First, predicates such as \( \text{id_name} = \text{id_name} \) can appear in the \( Z \) schemas. In FunZ, these predicates are deleted due to the semantics of the modifies tuple. (Recall the discussion from Section 3). The other difference is that the designer who uses FunZ as opposed to \( Z \) has a design document with Haskell code fragments after the completion of this step.

Step C3: construct the modifies tuple

To determine which variables should be placed in the modifies tuple, the designer checks the predicates

| TABLE 8. Predicates for the successful component of the Test operation |
|------------------------|------------------------|------------------------|
| (Abstract)             | FunZ                   | (Concrete)             |
| \( s? \in \text{enrolled} \) | \( s \not\in \text{elem} \text{ns} \) | \( s? \in \text{ran nottested} \) |
| \( s? \notin \text{tested} \) | \( s \not\in \text{notElem} \text{ts} \) | \( s? \notin \text{ran testlist} \) |
| \( \text{tested}' = \{s?\} \cup \text{tested} \) | \( \text{testlist}' = \{s?\} \cup \text{testlist} \) | |
| \( \text{enrolled}' = \text{enrolled} \) | \( \text{ns}' = \text{ns} \setminus \{s?\} \) | \( \text{nottested}' = \text{nottested} \setminus \{s?\} \) |
| \( r! = \text{success} \) | \( r = \text{Success} \) | \( r! = \text{success} \) |
obtained in the previous step for decorated variables. In the case of Testok, the relevant predicates are as follows: ns' = ns \ \{s\} and ts' = s:ts. Therefore, the modifies clause for concrete schema Testok is:

modifies Class (ns, ts)

Meanwhile, in the Z design, this step consists of a simple translation. In particular, instead of constructing a modifies clause, the designer translates all schema names included in the abstract operation schema to their respective names on the concrete state.

This completes the last step in the guidelines for translating an abstract specification written in Z to a more concrete one expressed in FunZ. However, for those designers who wish to prove that the resulting FunZ design is a correct refinement of the original Z specification, several proof obligations must be met, as discussed in the next section.

4.3. Proof obligations

The proof obligations in FunZ are analogous to those of Z. However, because the retrieve relation for the class manager's assistant is a function (see Section 4.2, step B3), the comments below are limited to functional refinement [18]. In particular, the discussion includes a brief overview of the proof obligations in a generic framework, followed by the conditions specific to this case study.

A single proof obligation must be fulfilled to show that every concrete initial state corresponds to an abstract initial state. In FunZ, this concrete is stated as follows:

∀Astate; StSpan • StSpan\init' ⇒ Ainit'

Note that Astate and Ainit represent the schemas denoting the abstract state space and the abstract initial state. Moreover, the notation StSpan\init' designates the concrete state after initialization. This means that the actual parameters of the corresponding init tuple replace the formal parameters when the system first becomes operational. Although the above proof obligation does not explicitly mention the retrieve function, its equations are visible due to the declaration of StSpan.

In addition to the condition relating initial states, functional refinement requires that every operation span satisfy two proof obligations. These obligations are known as the safety and liveness conditions [22]. The safety condition ensures that whenever an operation on the abstract state (Aop) terminates, its corresponding operation on the concrete state (OpSpan) will also terminate. This is expressed by

∀Astate; StSpan; x :: inp X • pre Aop
∧ StSpan\init ⇒ pre OpSpan

Meanwhile, the liveness condition guarantees that the concrete state resulting from a concrete operation represents a valid abstract state or, in other words, one that could terminate as a consequence of the corresponding abstract operation. The respective proof obligation is as follows:

∀ΔAstate; ΔStSpan; x :: inp X; y :: out Y •
pre Aop ∧ ΔStSpan\init ∧ OpSpan ⇒ Aop

The safety and liveness conditions of FunZ reflect the fact that a state span contains two additional components when compared to a state schema. In particular, the subterm StSpan\init tells the designer to disregard the initialization part, as the proofs for safety and liveness do not depend on any of the initialization predicates. Furthermore, as in the proof obligation for initial states, these conditions do not explicitly mention the retrieve function. Note that the FunZ proof obligations use the Δ symbol to denote before and after state schemas, as well as state spans.

In conclusion, the proof obligations for the initial states and test operation from the class manager's assistant follow. Here we simply list the required conditions, while the actual proofs appear in [40].

4.4. Haskell code

The class manager's assistant is a suitable candidate for an abstract data type since it consists of a datatype corresponding to the classroll and an associated set of operations which act on this type. In Haskell, each ADT is represented by a module. A skeleton of the module that implements the class manager's assistant follows.

```haskell
module ClassADT (Classroll, enroll, test, leave, enquire) where
import BasicDef
type Classroll = (Nottested, Tested)
type Nottested = [Student]
type Tested = [Student]
enroll :: Classroll -> Student -> (Classroll, Response)
-- Definition of enroll would appear here
test :: Classroll -> Student -> (Classroll, Response)
-- Definition of test (described below)
leave :: Classroll -> Student -> (Classroll, Response)
-- Definition of leave would appear here
enquire :: Classroll -> Student -> (Classroll, Response)
-- Definition of enquire would appear here
```

Module ClassADT exports the type Classroll and its associated operations: enroll, test, leave, and enquire. As is customary for an ADT, the representation of the type and the implementation of its operations are hidden from the user. The module does not contain definitions for the Student and Response types, as we are assuming that the module BasicDef, which appears in the import declaration, contains these definitions. Note that type Classroll could simply be defined as ([Student], [Student]). However, we introduce the
TABLE 9. Preconditions and postconditions for Test spans

<table>
<thead>
<tr>
<th>Span</th>
<th>Precondition</th>
<th>Postcondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testok</td>
<td>s 'elem' ns</td>
<td>ts' = s;ts</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ns' = ns \ {s}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r = Success</td>
</tr>
<tr>
<td>AlreadyTested</td>
<td>s 'elem' ts</td>
<td>r = Alreadytested</td>
</tr>
<tr>
<td>NotEnrolled</td>
<td>s 'notElem' ns</td>
<td>r = Notenrolled</td>
</tr>
<tr>
<td></td>
<td>s 'notElem' ts</td>
<td></td>
</tr>
</tbody>
</table>

We now translate the FunZ specification for the Test operation into a corresponding Haskell function. Recall the definition of Test:

\[
\text{Test} \triangleq \text{Testok} \lor \text{AlreadyTested} \lor \text{NotEnrolled}
\]

The disjunction of spans indicates that a case analysis is in order. To help clarify the analysis, Table 9 contains the pre- and post-conditions for each component of the operation.

Note that the single precondition \( s \ 'notElem' (ns + + ts) \) from span NotEnrolled has been replaced by two equivalent preconditions. By recalling the predicate

\[
(s \ 'elem' ns \ implies \ s \ 'notElem' ts) \\
& \& (s \ 'elem' ts \ implies \ s \ 'notElem' ns)
\]

from the state invariant of span Class (step B2 in Section 4.2), it is clear that the preconditions for each component are mutually exclusive. Therefore, the following implementation of the test function employs a sequence of guards.

\[
\begin{align*}
test :: \text{Classroll} \rightarrow \text{Student} \rightarrow (\text{Classroll}, \text{Response}) \\
test (ns, ts) s & | s \ 'elem' ns \ = \ ((ns \ \{s\}, s;ts), \text{Success}) \\
| s \ 'elem' ts & \ = \ ((ns, ts), \text{Alreadytested}) \\
| \text{otherwise} \ & \ = \ ((ns, ts), \text{Notenrolled})
\end{align*}
\]

The semantics of pattern matching indicate that the guards will be evaluated top to bottom until one returns the value True. In this case, if both the first and second guards should fail, then the precondition of NotEnrolled is guaranteed. To avoid the unnecessary evaluation of predicates \( s \ 'notElem' ns \) and \( s \ 'notElem' ts \), the expression otherwise comprises the last guard. Note that otherwise is simply syntactic sugar for the Boolean value True.

It is easy to see that the function above is a realization of its FunZ specification. In addition to the guards corresponding to the preconditions, notice that the first argument of test corresponds to the modifies tuple of span Testok (step C3 in Section 4.2).

5. CONCLUSIONS

We have defined an intermediate specification language, named FunZ, that can be used to produce purely functional programs from Z specifications. The formal text of a FunZ document looks like extended Haskell code, whereas the development process itself emulates that of Z. In particular, FunZ preserves the features of Z that contribute to the incremental development of specifications by maintaining the ideas of the schema calculus and schema inclusion. Furthermore, FunZ communicates the characteristics of a purely functional programming language to an implementor through special language constructs such as the modifies and init tuples.

In this paper, we gave a general overview of the FunZ notation and used a case study to illustrate how one can translate Z specifications to Haskell code. As a means of comparison, a corresponding Z design was developed simultaneously. We now summarize the differences between Z and FunZ as design languages by explaining how the usage of FunZ affects its two types of users: specifiers and implementors.

The major difference to the specifier/designer who uses FunZ has to do with the construction of the predicates on the concrete state. As illustrated in this paper, the expressions in a FunZ design are sometimes more complex (e.g. see the invariants in Table 3). However, the advantage gained in applying FunZ is that the designer is better able to describe certain aspects of the system, those unique to functional languages, since FunZ is expressly designed for this purpose.

The major difference to the implementor/programmer who works from a FunZ specification is that he or she actually begins with a skeleton of an implementation. Moreover, if the FunZ document includes formal proofs, the programmer has an even better understanding of the software design as it applies to Haskell. The overall effect is that the distance, and therefore, the time, from design to code is reduced.

Likewise, when comparing FunZ with traditional animation techniques [33], [44], and [43], two properties distinguish the approach encompassing FunZ: (i) the FunZ document serves as a record of the design process; and (ii) the user may prove the final implementation correct with respect to its initial specification. Furthermore, prototyping is still feasible within the framework of FunZ since code fragments are obtained relatively early in the process. A user may, in fact, provide the customer with a working prototype before satisfying any proof obligations.

Finally, the intermediate specification language FunZ and its associated methodology offer several advantages to software developers who wish to translate their Z specifications to functional programs. First, the specification language integrates features from Z with those of the functional programming paradigm to provide a bridge between Z specifications and functional implementations. At the same time, the methodology provides a framework for recording design decisions, which is useful for future maintenance. Within this framework, the user may select a development procedure ranging from an intuitive style, similar to that presented...
in this paper, to a fully formal approach as discussed in [40]. In conclusion, FunZ is an attractive software development method for both designers and implementors who prefer purely functional programming languages.

REFERENCES

L. B. SHERRELL AND D. L. CARVER


APPENDIX A

The initial Z specification for the class manager’s assistant follows. All schemas are from [22].

The abstract state

To describe the class manager’s assistant, we need three global variables in addition to the names of schemas. Since we are uninterested in the representation of a student, we use a given set to introduce the type Student.

\[ \text{Student} \]

A system user will be required to specify the maximum size of a class, so the variable size is declared.

\[ \text{size : N} \]

An enumerated type Response is defined in order that appropriate warnings or messages can be delivered at the conclusion of an operation.

\[ \text{Response ::= success | already tested | notenrolled ...} \]

The abstract state for the class manager’s assistant is defined in schema Class. The class roll is represented by set enrolled, while the set of students who have completed the exercises is designated by tested.

\[ \text{Class} \]

\[ \text{enrolled, tested} : P \text{ Student} \]

\[ \#\text{enrolled} \leq \text{size} \]

\[ \text{tested} \subseteq \text{enrolled} \]

When the class manager’s assistant is first activated, no students will be enrolled. The schema representing the abstract initial state follows.

\[ \text{ClassInit} \triangleq [\text{Class}'] | \text{enrolled}' = \emptyset \]

Notice that there is no need for a predicate stating that tested is initially empty. This fact can be derived from above since the tested students are a subset of the enrolled students.

Schemas for Test

Schema Testok specifies the successful recording of a student who has finished the exercises. In particular, it represents the case when the following two conditions hold: (i) a user enters an input student \( s? \) who has been tested or, in other words, completed the exercises; and (ii) the database contains no record that this student has been tested. After student \( s? \) is added to the set of tested students, the success response should be generated.

\[ \text{Testok} \]

\[ \Delta \text{Class} \]

\[ s? : \text{Student}; r! : \text{Response} \]

\[ s? \in \text{enrolled} \]

\[ s? \notin \text{tested} \]

\[ \text{tested}' = \text{tested} \cup \{s?\} \]

\[ \text{enrolled}' = \text{enrolled} \]

\[ r! = \text{success} \]

Schema AlreadyTested represents an error condition: the user enters a student \( s? \) who has previously been recorded as completing the exercises. The appropriate output response is alreadytested.

\[ \text{AlreadyTested} \]

\[ \forall \text{Class} \]

\[ s? : \text{Student}; r! : \text{Response} \]

\[ s? \in \text{tested} \]

\[ r! = \text{alreadytested} \]

Schema NotEnrolled also designates an error condition. In this case, the user enters student \( s? \) who is not a member of the class. The corresponding output response is notenrolled.

\[ \text{NotEnrolled} \]

\[ \forall \text{Class} \]

\[ s? : \text{Student}; r! : \text{Response} \]

\[ s? \notin \text{enrolled} \]

\[ r! = \text{notenrolled} \]

Finally, the disjunction of the previous schemas comprises the definition of operation Test.

\[ \text{Test} \triangleq \text{Testok} \lor \text{AlreadyTested} \lor \text{NotEnrolled} \]

APPENDIX B

The following function definitions, along with their comments, are adapted from the Standard Prelude as published in the Haskell Report [7]. Note that the symbol \( . \), which appears in several definitions, is the Haskell infix operator for function composition. Also \( /= \), which is defined in class Eq, is the symbol for inequality.

\[ ++ \quad :: [a] \rightarrow [a] \rightarrow [a] \]

\[ x + y = \text{foldr} (\lambda) y x \]

APPENDIX B

The following function definitions, along with their comments, are adapted from the Standard Prelude as published in the Haskell Report [7]. Note that the symbol \( . \), which appears in several definitions, is the Haskell infix operator for function composition. Also \( /= \), which is defined in class Eq, is the symbol for inequality.

\[ ++ \quad :: [a] \rightarrow [a] \rightarrow [a] \]

\[ x + y = \text{foldr} (\lambda) y x \]
foldr, applied to a binary operator, a starting value (typically the right-identity of the operator), and a list, reduces the list using the binary operator, from right to left:

\[
\text{foldr} \ f \ z \ [x_1, x_2, \ldots, x_n] = x_1 \ 'f' \ (x_2 \ 'f' \ (\ldots (x_n \ 'f' \ z) \ldots))
\]

\[
\text{foldr} \ f \ z \ [] = z
\]

\[
\text{foldr} \ f \ z \ (x : x_\text{s}x) = f \ x \ (\text{foldr} \ f \ z \ x_\text{sx})
\]

length returns the length of a finite list as an Int; it is an instance of the more general genericLength, the result type of which may be any kind of number.

\[
\text{genericLength} :: (\text{Num} \ a) \Rightarrow [\text{b}] \rightarrow \text{a}
\]

\[
\text{length} :: [\text{a}] \rightarrow \text{Int}
\]

\[
\text{length} = \text{genericLength}
\]

foldl is the left-to-right dual of foldr.

\[
\text{foldl} \ f \ z \ [] = z
\]

\[
\text{foldl} \ f \ z \ (x : x_\text{s}x) = \text{foldl} \ f \ (f \ z \ x) \ x_\text{s}x
\]

equality class

\[
\text{class} \ Eq \ a \ where
\]

\[
(==), (/=) :: \ a \rightarrow \ a \rightarrow \text{Bool}
\]

\[
x /= y = \neg (x == y)
\]

Boolean function for negation

\[
\text{not} :: \text{Bool} \rightarrow \text{Bool}
\]

\[
\text{not} \ True = False
\]

\[
\text{not} \ False = True
\]

nub (meaning "essence") removes duplicate elements from its list argument

\[
\text{nub} :: (\text{Eq} \ a) \Rightarrow [\text{a}] \rightarrow [\text{a}]
\]

\[
\text{nub} [] = []
\]

\[
\text{nub} \ (x : x_\text{s}x) = x : \text{nub} \ (\text{filter} \ (/= x) \ x_\text{s}x)
\]

filter, applied to a predicate and a list, returns the list of those elements that satisfy the predicate; i.e.,

\[
\text{filter} \ p \ x_\text{s}x = [x | x \in x_\text{s}x, p x]
\]

\[
\text{filter} \ p = \text{foldr} \ (\lambda \ x \ x_\text{s}x \rightarrow \text{if} \ p \ x
\]

\[
\text{then} \ x : x_\text{s}x \ \	ext{else} \ x_\text{s}x) \ []
\]

Boolean function for conjunction

\[
(\&\&): \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}
\]

\[
\text{True} \&\& \ x = x
\]

\[
\text{False} \&\& \ _ = \text{False}
\]

Boolean function for disjunction

\[
(||) :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}
\]

\[
\text{True} || \ _ = \text{True}
\]

\[
\text{False} || x = x
\]

elem is the list membership predicate, usually written in infix form, e.g., \(x \ 'elem' \ xs\).

\[
\text{elem} :: (\text{Eq} \ a) \Rightarrow a \rightarrow [a] \rightarrow \text{Bool}
\]

\[
\text{elem} = \text{any} \ . \ (==)
\]

notElem is the negation of predicate elem.

\[
\text{notElem} :: (\text{Eq} \ a) \Rightarrow a \rightarrow [a] \rightarrow \text{Bool}
\]

\[
\text{notElem} = \text{all} \ . \ (==)
\]

Applied to a predicate and a list, any determines if any element of the list satisfies the predicate.

\[
\text{any} :: (\text{a} \rightarrow \text{Bool}) \rightarrow [\text{a}] \rightarrow \text{Bool}
\]

\[
\text{any} \ p = \text{or} \ . \ \text{map} \ p
\]

map f xs applies f to each element of xs; i.e.,

\[
\text{map} \ f \ x_\text{s}x = [f \ x | x \in x_\text{s}x]
\]

\[
\text{map} \ f \ [] = []
\]

\[
\text{map} \ f \ (x : x_\text{s}x) = f \ x : \text{map} \ f \ x_\text{s}x
\]

or returns the disjunction of a Boolean list.

\[
\text{or} :: [\text{Bool}] \rightarrow \text{Bool}
\]

\[
\text{or} = \text{foldr} \ (||) \ False
\]

and is the conjunctive dual of or.

\[
\text{and} :: [\text{Bool}] \rightarrow \text{Bool}
\]

\[
\text{and} = \text{foldr} \ (\&\&) \ True
\]

list difference (non-associative). In the result of \(x_\text{s}x \setminus y_\text{s}x\), the first occurrence of each element of \(y_\text{s}x\) in turn (if any) has been removed from \(x_\text{s}x\). Thus, \((x_\text{s}x + y_\text{s}x) \setminus y_\text{s}x = y_\text{s}x\).

\[
\text{\setminus} :: (\text{Eq} \ a) \Rightarrow [a] \rightarrow [a] \rightarrow [a]
\]

\[
\text{\setminus} = \text{foldl} \ \text{del}
\]

del takes a list and an element and returns an identical list except the first occurrence of the specified element is removed.

\[
\text{del} :: (\text{Eq} \ a) \Rightarrow [a] \rightarrow a \rightarrow [a]
\]

\[
\text{del} \ [] = []
\]

\[
\text{del} \ (x : x_\text{s}x) | x == y = x_\text{s}x
\]

\[
\text{otherwise} = x : x_\text{s}x \ 'del' \ y
\]
APPENDIX C

The following design spans are based on the schemas in Appendix A. For additional details concerning spans Class and Testok see Section 4.2.

Span Class
span Class where
  ts :: [Student]
  ns :: [Student]

inv is
  length (ns + +ts) <= size
  (ns == nubns)&&(ts == nub ts)
  Vs :: Student • (s 'elem'  ns implies s 'notElem' ts)
  Vs :: Student • (s 'elem' ts implies s 'notElem' ns)

rel is
  abmap :: [Student] —> P Student
  tested = set ts
  enrolled = set (ns ++ ts)

init is
  (ns, ts)
  ([ ], [ ])
end span Class

Span Testok
span Testok where
  modifies Class (ns, ts)

  s :: in Student
  r :: out Response

pred is
  s 'elem' ns
ts' = s:ts
ns' = ns \ \[s]
s = Success
end span Testok

Span AlreadyTested
span AlreadyTested where
  modifies Class ( )

  s :: in Student
  r :: out Response

pred is
  s 'elem' ts
  r = Alreadytested
end span AlreadyTested

Span NotEnrolled
span NotEnrolled where
  modifies Class ( )

  s :: in Student
  r :: out Response

pred is
  s 'notElem' (ns ++ ts)
  r = Notenrolled
end span NotEnrolled

Span Test
span Test where
  Test = Testok V AlreadyTested V NotEnrolled
end span Test