A Formal Methodology for Deriving Purely Functional Programs From Z Specifications via the Intermediate Specification Language FunZ.

Linda Bostwick Sherrell
Louisiana State University and Agricultural & Mechanical College

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_disstheses

Recommended Citation
https://digitalcommons.lsu.edu/gradschool_disstheses/5981

This Dissertation is brought to you for free and open access by the Graduate School at LSU Digital Commons. It has been accepted for inclusion in LSU Historical Dissertations and Theses by an authorized administrator of LSU Digital Commons. For more information, please contact gradetd@lsu.edu.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
A FORMAL METHODOLOGY FOR DERIVING PURELY FUNCTIONAL
PROGRAMS FROM Z SPECIFICATIONS
VIA THE INTERMEDIATE SPECIFICATION LANGUAGE FUNZ

A Dissertation
Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in
The Department of Computer Science

by
Linda Bostwick Sherrell
B.S., Auburn University, 1972
M.S., Auburn University, 1974
May 1995
Acknowledgements

This dissertation would not have been possible without the support of numerous individuals. First, I would like to thank Dr. Doris L. Carver, my major professor, for encouraging me to submit my results to international conferences and journals early in the process. The comments from reviewers have helped to improve this work and to inspire me to continue on my path. Furthermore, Dr. Carver has presented an excellent role model. I only hope that I will be able to show the same concern and understanding towards my own students, while balancing the demands of teaching, research, and family.

Second, I would like to thank each of the members of my committee for their contributions to my success. Dr. Harriet Taylor was my first computer science teacher at Louisiana State University. Needless to say if her programming course had not been enjoyable, I would not be here today. Dr. J. Bush Jones has provided me with an excellent background in compilers and has always been accessible for questions. Dr. Robert F. Lax, my minor professor, has continually shown an interest in my career, beginning with my days as an Instructor in the Department of Mathematics at LSU and later as my modern algebra professor. I would also like to thank Dr. Lai-Him Chan, from Physics & Astronomy, for agreeing to serve as the Dean’s Representative.

Two other professors from the Department of Computer Science deserve a special thanks. Dr. Andrzej Hoppe, who was one of my early advisors, first introduced me to the functional programming paradigm by emphasizing the pure aspects of Lisp. Dr. Donald H. Kraft has consistently demonstrated his support throughout the years even though he was not a member of my committee.
I would like to thank all the members of the Software Engineering Group for their friendship and encouragement. In addition, I would like to thank David Hoelzeman with whom I shared an office for several years. His continued support after accepting an academic job have helped to provide me with a different perspective, and I appreciate his willingness to listen to my problems.

I would also like to thank a special friend and colleague, Gerry Vidrine. Gerry and I became good friends when we were both instructors in the Mathematics Department at LSU. Even though our lives have taken unexpected turns over the years, she has always found occasions to check on my progress. I am grateful that she has been so understanding about the limitations on my time.

Finally, I would like to thank various members of my family. First, I would like to thank my father, Leonard C. Bostwick, for giving me the confidence in myself to pursue a degree in the sciences when it was unfashionable for women. I would like to thank my mother, Nellie Rose Ann, for allowing me to participate in numerous activities throughout my school years.

Second, I would like to thank my two sons, Robert Lawson and David Jeffrey, for understanding that I needed to pursue an advanced degree at this time in my life. Even when I missed one of their baseball games or special school functions, they never made me feel guilty. In fact, their support and encouragement sometimes made me wonder who was the parent and who was the child.

Last, but most of all, I would like to thank my husband Dan. Without his untiring patience, love, and reassurance, none of this would have been possible. I know that at times it must have seemed like I would never finish my degree, and his life would become an endless discussion about my dissertation. I want to take this opportunity to tell him how much I appreciate all the sacrifices that he has made for me.
# Table of Contents

Acknowledgements .............................................................................................................. iii
List of Tables ....................................................................................................................... vii
List of Figures ...................................................................................................................... viii
Abstract ................................................................................................................................ ix

Chapter
1. Introduction ............................................................................................................  1
  1.1 Overview .............................................................................................................  1
  1.2 Functional Programming ................................................................................ 2
    1.2.1 Historical Influences ............................................................................  3
    1.2.2 Language Features .............................................................................  4
    1.2.3 The Programming Language Haskell .................................................. 9
  1.3 Formal Methods .............................................................................................. 13
    1.3.1 A Taxonomy for Formal Specification Languages .............................. 14
    1.3.2 The Z Specification Language ............................................................. 16
  1.4 Outline of the Dissertation .............................................................................. 18

2. Related Research ................................................................  21
  2.1 Prototyping ...................................................................................................... 21
  2.2 Formal Program Development ....................................................................... 24
    2.2.1 Extended ML ....................................................................................... 25
    2.2.2 Larch/ML .............................................................................................. 26
    2.2.3 From Z to Lazy ML ............................................................................. 28
  2.3 A Frame of Reference for FunZ ................................................................. 30

3. FunZ: The Language and Associated Methodology ............................................... 36
  3.1 Motivation ....................................................................................................... 36
    3.1.1 Why Z and Haskell? ............................................................................ 38
    3.1.2 Why an Intermediate Language? .......................................................... 40
  3.2 Definition of FunZ ....................................................................................... 42
  3.3 FunZ Syntax .................................................................................................... 45
List of Tables

1. A Comparison of FunZ to Related Approaches .......................................................... 33
2. Transformation Rules ............................................................................................ 57
3. Abstract and Concrete Predicates for Schema Leaveok .............................................. 69
4. Preconditions and Postconditions for Concrete Leave Schemas ............................... 72
5. Z Predicates and Haskell Expressions ......................................................................... 73
6. Z and Haskell Operators ............................................................................................. 73
7. Preconditions and Postconditions for Test Spans ...................................................... 88
8. Objects of the State Space ......................................................................................... 91
9. Predicates for the Successful Component of the Test Operation ............................ 91
10. Invariants on the State Space .................................................................................. 92
List of Figures

1. Template for a FunZ State Span ................................................................. 43
2. Template for a FunZ Operation Span ......................................................... 44
3. Basic Phases of the Methodology ............................................................. 49
4. Formal Text for Span Class ..................................................................... 76
5. Predicates from Schema Testok ............................................................... 78
6. Retrieve Function from Span Class ......................................................... 79
7. Predicates for Span Testok ..................................................................... 81
Abstract

In recent years, both formal methods and software reuse have been increasingly advocated as a means of alleviating the ills of the software crisis. During this same time period, purely functional programming languages, which have a long history in the realm of rapid prototyping, have emerged as a viable medium for real-world applications. Since these trends are likely to continue, this work describes a methodology that facilitates the derivation of purely functional programs from existing Z specifications. A unique aspect of the methodology is its incorporation of an intermediate specification language (FunZ) during the design phase of software development.

Most of the previous techniques for translating Z specifications to functional programs were designed primarily to expedite rapid prototyping. In contrast, the FunZ methodology, which is an adapted form of the IBM Hursley method, is a comprehensive approach, spanning the software life cycle from specification through design to final implementation. Due to its greater scope, the FunZ methodology offers several advantages over existing approaches. 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. Since FunZ is expressly designed to target functional languages, the implementor's job is simplified. In fact, a FunZ document looks like extended Haskell code, so an obvious effect in applying FunZ is that the distance from design to code is reduced. Second, the methodology provides a framework for recording design decisions, which is useful for future maintenance. Within this framework, users may select a development path ranging from an intuitive
style to a fully formal approach that includes the proofs of functional refinement. Furthermore, FunZ allows software developers to prove properties about a system design within the realm of Z or Haskell. This means that proofs can be performed throughout software development and the designer is free to select the most appropriate notation.

In summary, the intermediate specification language FunZ and its related methodology provide software developers with a complete, formal approach for translating Z specifications to Haskell implementations. Previously, such extensive methods were only available for traditional, imperative languages.
Chapter 1

Introduction

1.1 Overview

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 life cycle. 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. Its proponents claim several advantages including shorter programs, code that is easier to understand, and faster software development times. Undoubtedly, functional languages, which comprise a subclass of the family of declarative languages, are more abstract than their imperative counterparts. These languages allow the programmer to concentrate on what a program should do as opposed to how, and simultaneously free the implementor from low-level concerns such as flow of control and memory management.

Although functional languages have a long history in the realm of rapid prototyping [Hend86; Hekm88; Joos89], mainstream usage has been somewhat limited due to efficiency concerns. However, with improvements in compilers and advances in computer architecture, this trend is beginning to change. [Sand91] has shown the execution speeds of some functional programs compare favorably with those of C. Meanwhile, the FLARE Project [Sand93] is a concerted effort between academia and industry to tap the potential of functional languages and to promote their use in real applications.
As another means to diminish the problems associated with constructing reliable software, the field of software engineering is currently experiencing a renewed interest in software reuse [Krue92]. Although software reuse includes the modification of source code, more generally it refers to the use of any preexisting software component as an aid in developing new software [Free83].

This research integrates software reuse, formal methods, and functional programming by defining a formal methodology for deriving purely functional programs from existing Z [Spiv89] specifications. A unique aspect of the methodology is its usage of an intermediate specification language, named FunZ [Sher95], during the design phase of the software life cycle. FunZ is the first intermediate specification language to target the functional programming paradigm. By combining features from Z and the functional programming language Haskell [Huda92b], FunZ provides a natural link between Z specifications and Haskell implementations. As background for the dissertation, the remainder of this chapter contains a summary of functional programming followed by a brief introduction to formal methods. The chapter concludes with an outline describing the organization of the entire dissertation.

1.2 Functional Programming

Functional, also known as applicative, programming derives its name from its method of computation, function application. A function in a purely functional program is equivalent to a mathematical function. The argument or variable of a function does not change once given a value, and a function's only effect is to evaluate its definition, an expression. Because functions have no side effects, expressions can be evaluated at any time and subexpressions with equal values can be substituted one for the other. This inherent property of mathematical functions is known as referential transparency.

Referential transparency is the most distinguishing characteristic of purely functional languages and its influence permeates the entire software life cycle. At the design stage, the
programmer is freed from low-level concerns such as flow of control. During verification, algebraic laws can be used to prove program correctness. When corrective maintenance is required, debugging is easier.

1.2.1 Historical Influences

Lisp [McC60], the first functional programming language, was designed by McCarthy for artificial intelligence applications. Although the original implementation was a purely functional language, imperative features such as assignment statements and sequencing constructs were later added. Nevertheless, three features of Lisp are an intrinsic part of present-day functional languages and their implementations [Huda89]:

1. The use of the conditional expression to write recursive functions.

2. Lists and the use of higher-order functions over lists.

3. System storage management including garbage collection.

After Backus introduced the functional language FP in his Turing Award lecture [Back78], increased importance was given to the functional programming paradigm. In this landmark paper, Backus praised the virtues of functional programming and, at the same time, exposed the limitations of imperative programming. He argued that the programming constructs of imperative languages too closely model the operations and architecture of the von Neumann computer. Additional weight was given to Backus' position, since he was the person most responsible for the design and implementation of FORTRAN [Huda89]. In fact, Backus was presented the Turing Award primarily for his contributions to the design of two imperative languages: FORTRAN and ALGOL.

Backus borrowed ideas from APL [Iver72] for his language FP. Just one indication of this is the notation of FP's functional forms, programming constructs that are used to form new functions. A distinguishing feature of these combining forms is the absence of parameters. In other words, arguments are not named. Even though this design was not adopted by
most functional languages, several program transformation techniques are founded on the FP algebraic approach [Fiel88].

Another language which has had a tremendous impact on functional languages is ML, which evolved into Standard ML [Miln90]. Standard ML is not a purely functional language due to its references (special values similar to variables in an imperative language) and its I/O facilities, neither of which is referentially transparent. However, its designers promote the functional style of programming and recommend the sparing use of references [Toft91].

ML's most important contribution to language design is its Hindley-Milner type system, as evidenced by the fact that it is currently used in all statically typed functional languages [Huda89]. This type system allows polymorphic functions (functions which may be applied to objects of any type) and uses a type inference algorithm to infer types rather than requiring type declarations.

1.2.2 Language Features

Some of the abstraction facilities provided by modern functional languages include list comprehensions, pattern matching, and lazy evaluation. Collectively, these features allow algorithms to be expressed in a natural and concise fashion. Furthermore, higher-order functions and polymorphism promote the reusability of code. The discussion below elaborates on some of these aspects by providing examples written in Haskell, an emerging standard for non-strict (also known as lazy), purely functional languages. Note that the basic list functions are from the Standard Prelude of Haskell as published in [Huda92b].

Lists

The list, an ordered collection of elements, has served as the primary data structure for functional languages beginning with the list-processing language Lisp. In modern functional languages, the elements of a list are all of one type, and the following list notation is
employed: \([\ ]\) represents the empty list and \((x:xs)\) the nonempty list, where \(x\) denotes the head of the list, \(xs\) its tail, and \(:\) is the infix list constructor also known as the cons function.

When enumerating the elements of a list, commas separate each element and square brackets surround the list itself. As an example of enumeration and list construction, the expression \(1: [5, 10, 25, 50]\) returns the list \([1, 5, 10, 25, 50]\). Some useful abbreviations for lists include \([a..b]\), which designates the list of integers between \(a\) and \(b\), inclusive, and \([a..]\), which denotes the infinite list of integers beginning with the number \(a\).

List comprehensions

List comprehensions is the popular name for ZF expressions, which were first implemented by Turner in his language KRC [Turn81]. The notation for a list comprehension closely resembles that of a set abstraction as illustrated by the following example:

\[
[3*y \mid y <- [1..10], \text{even } y]
\]

In the above expression, \(y <- [1..10]\) is a generator that generates the numbers between 1 and 10 inclusive, while \(\text{even } y\) is a guard that assures that only the even elements are bound to \(y\) in the subexpression \(3*y\). Therefore, the value returned by evaluating the entire list comprehension is a list of triples of the even numbers from 1 to 10.

In general, a list comprehension can have any finite number of generators and Boolean guards, which are collectively referred to as qualifiers. The qualifiers are evaluated left to right, with the rightmost generator producing values in a depth-first fashion.

Pattern matching

Languages that support pattern matching allow a function definition to contain several equations. Consider the definition of function \(\text{take}\), which returns a prefix of a list:

\[
\begin{align*}
\text{take} & \quad 0 \quad _{} = \quad [\ ] \\
\text{take} & \quad _{} \quad [\ ] = \quad [\ ] \\
\text{take} & \quad (n+1) \quad (x:xs) = \quad x : \text{take} \quad n \quad xs
\end{align*}
\]
Due to the semantics of pattern matching, the left-hand sides of the equations are evaluated top to bottom, left to right, until one pattern succeeds or diverges, or they all fail. If a pattern succeeds, the associated right-hand side is evaluated and returned as the result of the function. Observe that some of the equations in the previous example contain underscores or wildcards. A wildcard matches any actual parameter and saves the programmer time by eliminating the need to create variable names.

Besides pattern matching, this definition also demonstrates the concept of currying. In a curried function, arguments appear as a sequence of simple arguments rather than as a single structured argument. This results in far fewer parentheses. For example without currying, each left-hand side of the above equations requires additional bracketing and the right-hand side corresponding to the last equation becomes \( x : \text{take} \ (n, \ xs) \).

Finally, there are several advantages to the use of pattern matching. Pattern matching complements the use of equational reasoning in the design and verification of programs. Furthermore, it greatly enhances the readability of a program. To emphasize the point, compare the previous definition of the function \( \text{take} \) with its conditional expression equivalent:

\[
\begin{align*}
\text{take} \ n \ l & = \text{if } n == 0 \ \text{then } [] \\
& \quad \text{else if } l == [] \ \text{then } [] \\
& \quad \text{else } (\text{hd} \ l) : (\text{take} \ (n-1) \ (\text{tl} \ l))
\end{align*}
\]

The subexpression \( \text{hd} \ l \) returns the head of list \( l \), whereas \( \text{tl} \ l \) yields its tail. These functions are unnecessary in the first definition because the formal parameter \( (x:xs) \) automatically deconstructs each actual argument into its corresponding head and tail when a successful match occurs.

**Higher-order functions and polymorphism**

A higher-order function is one that accepts a function as an argument or returns a function as a result. As an example, consider function \( \text{map} \) that applies a function to every element of a list:
Function \texttt{map} is a standard list function for functional languages. Another such function is \texttt{foldr}. When \texttt{foldr} is applied to a binary operator, an initial value, and a list, it returns a new list where each element is the result of the binary operator applied to consecutive elements in the original list. The initial value is necessary because the list could be empty or contain a single element. The definition of \texttt{foldr} follows:

\[
\begin{align*}
\text{foldr } f &\ z \ [\ ] \ = \ z \\
\text{foldr } f &\ z \ (x:xs) \ = \ f \ x \ (\text{foldr } f \ z \ xs)
\end{align*}
\]

By applying \texttt{foldr} to the infix operator and identity element for addition, one can compute the sum of a list of numbers:

\[
\text{sumlist } = \ \text{foldr } (+) \ 0
\]

Similarly, the next function calculates the product of a list of numbers:

\[
\text{prodlist } = \ \text{foldr } (*) \ 1
\]

The last two definitions demonstrate another property of currying, namely that function applications may contain fewer arguments than their respective definitions. In particular, if a function contains \(n\) arguments and is applied to only \(m\) of them, the result is a new function with \(n - m\) arguments [Hugh89]. By allowing unnecessary arguments to be omitted, currying promotes program readability.

Finally, functions \texttt{map} and \texttt{foldr} are polymorphic so they can be applied to lists containing any element type. In a language that does not exhibit polymorphism, e.g., Pascal, a function corresponding to every element type must be defined. As these examples illustrate, polymorphism and higher-order functions both facilitate modular programming. For a more in-depth discussion of higher-order functions with respect to modularity, see [Hugh89].
Lazy evaluation

In order to execute programs, many of the popular functional programming languages employ a strategy known as lazy evaluation. The term lazy refers to the fact that an expression is only computed if its value is required by the surrounding environment. Furthermore, the strategy allows common subexpressions to share the same reduction graph, thus limiting the number of reduction steps to evaluate an expression.

Due to lazy evaluation, the programmer is no longer encumbered by efficiency concerns and can focus on producing better code. In addition, the call by need evaluation strategy permits the definition of infinite streams. For example,

$$\text{posints} = 1 : \text{map (+1) posints}$$

denotes the set of positive integers. A more interesting example adapted from [Turn82] is the following definition of primes:

$$\text{primes} = \text{sieve [2..]}$$

where

$$\text{sieve (p:xs)} = p : \text{sieve [y | y <- xs, y \mod p > 0]}$$

Finally, the power of infinite lists is that they allow a program to remain at a very high level and they promote reusability. Of course, an actual program can only compute finite prefixes of such lists or the program would never terminate. As a typical example of how one might apply function primes, the expression $\text{take 100 primes}$ produces the first 100 primes.

User-defined types

In functional languages, users can define either abstract data types or concrete types. Whereas an abstract data type (ADT) defines a type and a group of operations associated with the type, a concrete type describes how to construct values of a type via data constructors. In Haskell, ADTs are defined by using modules. Since modules are common to most programming languages, specific modules are not presented here. However, section 4.5.4 describes a
Haskell module for a particular implementation, and the interested reader can refer to [Huda92b] for additional examples. Meanwhile, the subsequent definition of a binary tree represents a typical concrete type:

```haskell
data Btree e = Empty | Node e (Btree e) (Btree e)
```

The reserved word `data` introduces new types. In this case, `Btree` is a type constructor, and `Empty` and `Node` are the associated data constructors. The difference between applying type and data constructors is that the former takes place at compile time to ensure type safety, whereas the latter is a run-time activity to yield values [Huda92a].

Notice that the definition for `Btree` is both recursive and polymorphic. To define a binary tree whose nodes contain real floating-point numbers, one can use a type synonym:

```haskell
type Fl_btree = Btree Float
```

By allowing the programmer to assign more meaningful names to existing types, type synonyms promote readability.

In summary, functional languages not only provide a mechanism for defining abstract data types, but they also support concrete types. Since concrete types can be polymorphic and function definitions can be designed to pattern match against the associated data constructors, concrete types present another means for constructing reusable code.

1.2.3 The Programming Language Haskell

Prior to the development of Haskell, a preponderance of purely functional languages had appeared on the scene. According to [Huda89], these included Hope (Edinburgh), FEL (Utah), Lazy ML (Chalmers), Alfl (Yale), Ponder (Cambridge), Orwell (Oxford), Daisy (Indiana), Twentel (University of Twente), and Tui (Victoria University). Furthermore, there were also the three languages developed by David Turner, namely SASL (St. Andrews Static Language), KRC (Kent Recursive Calculator), and Miranda™ (a commercial product). Although

™ Miranda is a trademark of Research Software Ltd.
the semantics of these languages were quite similar, many researchers felt that a common language was needed if functional programming were to become more mainstream. Therefore, near the end of 1987, an international committee was formed to design a language that would include the most popular features of modern functional languages. This new language, named Haskell in honor of the logician and mathematician Haskell B. Curry, would be promoted as the standard for non-strict, purely functional languages.

Although Haskell was first introduced to the general public in an ACM Computing Surveys article about functional programming languages [Huda89], it received its biggest boost when Sigplan Notices devoted an entire issue to its coverage. (The latter periodical included both a tutorial [Huda92a] and the official Haskell Report [Huda92b].) Shortly thereafter, a functional programming textbook [Davi92] emphasizing Haskell also appeared on the market. Until just recently, the majority of other publications involving Haskell dealt with implementation concerns [Hamm90; Peyt91] or enhancements to existing language features [Chen92].

Haskell was designed as a general purpose language appropriate for teaching, research, and building large systems. Compared to previous functional languages, Haskell most closely resembles Miranda. However, its designers have added some original facilities to the functional programmer's repertoire and broadened others. In particular, arrays and type classes are the major additions to Haskell. The following discussion elaborates on these new language features.

Arrays

To define an array in Haskell, one can use the predefined function array. As an example, the following definition from [Huda92b] computes the first n Fibonacci numbers and returns them in an array a. An important point is that the array itself acts as a cache transforming the traditional, exponential algorithm into a linear one [Huda89].
fibs n = a where a =
  array (0, n) ([0 := 1, 1 := 1] ++
    [i := a!(i-2) + a!(i-1) | i <- [2..n]])

The first argument of function array is a pair corresponding to the bounds for the array, while the second argument is a list of associations written as index := value. In this case, the second argument is formed by first concatenating two lists with the infix operator ++. Note that the symbol ! is the subscript operator for arrays.

Theoretically, arrays do not provide the functional programmer with additional expressive power since they can be modeled with lists or more complex data types that associate index types with value types [Davi92]. However, the syntax for arrays complements that of list comprehensions, thus reinforcing the mathematical style of functional programming. Furthermore, the non-strict semantics of Haskell means that the elements in the association list of each array can occur in any order since they will be evaluated as needed. This is especially helpful when programmers are constructing algorithms that involve matrices and recurrence equations since data dependencies can be ignored [Huda89].

Type classes

The most novel addition to Haskell is its extension to the Hindley-Milner type system, namely type classes, in order to systematically handle overloading or ad hoc polymorphism. Previously, there was no standard technique for dealing with overloaded functions such as equality (=), arithmetic (+), and string conversion (show). In fact, even within a single language overloading was not handled in a uniform manner. For example, Miranda invokes three different approaches: equality is defined on all types, there is a single numeric type, and yet a third definition is used for string conversion. Similarly, Standard ML has two approaches to manage overloading. For mathematical operators and string conversion, overloading is resolved at the point of occurrence, whereas in the case of equality, only special type variables can be compared [Hall92].
In Haskell, type classes for equality, arithmetic, and string conversion are predefined. Furthermore, Haskell not only allows programmers to add extra types to these and other standard classes, but users can also create their own type classes. To illustrate the basic concepts, an example from [Huda92b] follows:

```haskell
class Num a where -- simplified class declaration for Num
    (+)      :: a -> a -> a
    negate   :: a -> a

instance Num Int where -- simplified instance of Num Int
    x + y   = addInt x y
    negate x = negateInt x

instance Num Float where -- simplified instance of Num Float
    x + y   = addFloat x y
    negate x = negateFloat x
```

In this example, the operations (+) and negate are overloaded on types Int and Float. First, the class declaration for Num expresses the fact that each instance or type in this class must define methods corresponding to (+) and negate. Next, the type declarations for Int and Float state that these types are instances of the type class Num. Finally, the identifiers addInt, negateInt, addFloat, and negateFloat are the names of the primitive functions that implement the required methods.

As the reader has probably noticed, type classes are similar to the classes of object-oriented programming. In fact, Haskell supports both single and multiple inheritance within its type classes. However, there are two primary differences: 1) Haskell types are not objects so there is no internal mutable state and 2) Haskell classes are entirely type-safe. Since the security of type classes is an important advantage, a more complete explanation from [Huda92a] of why type classes are secure follows:

Any attempt to apply a method to a value whose type is not in the required class will be detected at compile time instead of runtime. In other words, methods are not "looked up" at runtime but are simply passed as higher-order functions.
In summary, type classes are a valuable addition to the Haskell programming language. The consistent treatment of overloading for both predefined and user-defined operations makes the language easier to learn and use in practice. Furthermore, the fact that programmers can group types with common operations together into a single class provides an extra means of organizing large software projects. For an actual industrial application that employs type classes, see [Sand93].

1.3 Formal Methods

Due to the size and complexity of current software projects, suitable methods must be employed in order to produce software that is both cost-effective and reliable. Whereas informal methods quickly become unwieldy in large applications, formal methods are usually more concise and offer other advantages as well.

These advantages are directly related to the mathematical underpinnings of each formal method as embodied in an associated specification language. Formal methods enforce a necessary precision that helps to expose ambiguities, inconsistencies, and incompleteness in a system [Wing90]. Furthermore, by following the guidelines of a formal method, the software developer automatically creates a record of each design decision, which is valuable for both verification and maintenance activities. In general, formal methods foster software quality assurance.

Despite the fact that formal methods have generally been accepted as necessary tools in developing safety-critical software, usage in other application areas has often lagged behind. In some cases, companies were reluctant to invest the time and expense to reeducate personnel. Often support tools needed for the application of formal methods were lacking. In addition, common perceptions concerning the mathematical expertise required to understand formal methods sometimes discouraged their use.
However, the last few years have witnessed an increased interest in formal methods as more practitioners have documented their experiences in producing software for real systems. As a sampling, [Word89] describes a communications programming interface, [Deli90] discusses the role of specifications in designing oscilloscopes, and [Spiv90] presents the results from a case study that specified the kernel for an X-ray machine. Additional applications from a report [Gerh93] on industrial usage of formal methods include an automatic train protection system, an air traffic control system, a restructuring tool for COBOL code, and a real-time database for patient monitoring. Furthermore, [Hall90] dispels some common misconceptions concerning formal methods by relating his experiences in designing a tool set to complement SSADM, a structured systems analysis and design method.

Some of the points established in [Hall90] are the following:

1. Formal methods should not be restricted to safety-critical software; they are useful in almost any application area.
2. Formal specifications do not contain complex mathematics; they are actually easier to understand than programs.
3. Formal methods need not increase development costs. In fact, since errors are often exposed early in the development process, costs can actually decrease.
4. Formal methods do mean more time is spent on specification. However, the implementation, integration, and testing stages are shorter.

In the end, formal methods alone will not remedy all the problems associated with software development. However, their adoption can improve the general effectiveness of each stage in the software life cycle.

1.3.1 A Taxonomy for Formal Specification Languages

Formal specification languages are often classified according to their characteristics. One leading taxonomy divides specification languages into two primary groups: model-oriented
and property-oriented. Specifiers who use a model-oriented language explicitly describe a system's behavior by building a model with mathematical objects, such as mappings, sequences, and sets. Meanwhile, specifiers who employ a property-oriented language define a system's behavior implicitly by listing a collection of properties that the system must satisfy [Wing90]. Property-oriented languages can be further subdivided into axiomatic and algebraic specification languages. With an axiomatic language, a software developer use preconditions and postconditions to specify the operations of a data type, whereas with an algebraic language, he or she represents the abstract data types as many-sorted algebras [VanH89]. Some representative examples follow.

The two most widely known model-oriented specification languages are VDM [Jone80; Jone86] and Z [Haye87; Spiv89]. Meanwhile, an important algebraic specification language is CLEAR [Burs77b, Burs80a] since it was the first specification language to use mathematical descriptions of abstract data types. Furthermore, CLEAR influenced the design of several other specification languages. In particular, the specification language OBJ [Gogu79; Gogu84] and its successors, OBJ 2 [Futa85; Futa87] and OBJ 3 [Gogu88], borrow features from CLEAR and Hope [Burs80b], an early functional programming language. More recently, UMIST OBJ [Gall89] implements an executable subset of OBJ. Finally, an example of a language based on the axiomatic approach to specification is Anna [Luck87]. A unique characteristic of this language is that it targets a single programming language, namely Ada.

It should be noted that the classification for specification languages presented here is by no means clear-cut; several languages support more than one style of specification. For instance, OBJ is often described as an algebraic language, but [Wing90] includes the language in a listing of axiomatic specification languages. Similarly, it is possible to construct property-oriented specifications with the model-oriented language Z. (As an illustration, see Samp90), which presents property-oriented descriptions of both the natural numbers and binary trees.
using a modular extension of Z.) Perhaps the most striking example of a language that supports more than one style of specification is Larch [Gutt85] because each specification has two parts, one written in the axiomatic form and another expressed in the algebraic mode.

1.3.2 The Z Specification Language

Z is a formal specification language based on typed set theory and predicate logic. The language was developed by the Programming Research Group at Oxford University in the 1980’s and has since become one of the more popular specification languages as evidenced by several conferences on formal methods [Bjor90; Preh91; Nich92; Bowe93; Naft94].

Each Z document is a combination of formal mathematical text and informal prose. The informal statements are necessary to explain the formal notation, whereas the formal text provides the required precision for an unambiguous specification. To highlight important sections of formal text, Z provides a graphical notation called a schema.

Schemas are the building blocks of Z specifications. Typically, a Z document will contain a state schema to specify the system state, and several operation schemas to describe state transitions. The structure of each schema consists of two components: 1) a declaration part containing mathematical variables and 2) a predicate part expressing relationships between these variables. Schemas may be written in either a vertical or a horizontal format as illustrated by the following templates:

```
Schema_Name __________________________________________
| declaration part
| predicate part _______________________________________
```

`Schema_Name ≜ [declaration part | predicate part]`

Since specifications can become quite lengthy, Z users usually apply conventions that help to make their documents more manageable and easier to read. Some of the more
common conventions are schema inclusion, identifier decoration, and the Delta (Δ) and Xi (Ξ) conventions. Definitions for these conventions follow.

Schema inclusion

Schema inclusion is when the name of one schema appears in the declaration part of another. The effect is all of the declarations and predicates of the first schema are then visible in the second. Among the reasons that schema inclusion is beneficial is that it creates shorter specifications while helping to reduce typographical errors [Mona91].

Identifier decoration

Special symbols can appear as suffixes of both variable names and schema names. The standard characters are ? to specify inputs, ! to designate outputs, and ' (prime) to indicate final states, the values after an operation has been performed. Note that unprimed identifiers denote the corresponding values before an operation.

Meanwhile, schema decoration creates a new schema from an existing one by replacing all the variable names of the original schema with the appropriate suffix. Schema decoration complements schema inclusion and the two are often used in combination.

Delta and Xi conventions

Let S be the name of a schema with n variables. Then ΔS is an abbreviation for the schema formed by two declarations, namely S and S'. Similarly, ΞS is a shorthand notation for a schema that includes S and S', as well as n new predicates, where each predicate has the form var_name' = var_name, 1 ≤ i ≤ n.

An easy way to remember these conventions is the following:

Δ symbolizes change – the change in value of one or more state components after an operation has been completed.
Ξ represents stability – there is no change to the state when the operation is performed.

Schema calculus

Another important feature of Z is its associated schema calculus because the calculus supports the incremental development of specifications. In particular, by applying operators from propositional logic to existing schemas, new schemas are created.

Z supports several schema operations. As a typical example, consider the following description of schema disjunction (\(\lor\)). By disjoining two or more schemas, a specifier designates a new schema whose declaration part is the union of all declarations from the original schemas and whose predicate part is the disjunction of the respective predicate parts. Note that the following definition defines a schema \(V\), which is the disjunction of schemas \(S\) and \(T\):

\[
V = S \lor T
\]

As a rule, one builds a specification by first specifying the normal conditions of an operation and then adding the error conditions. Schema disjunction is the last step to combine all the conditions into a single operation.

In summary, schemas provide specifiers with the power to structure their specifications. Moreover, the schema calculus and conventions such as schema inclusion promote the reuse of existing specification units.

1.4 Outline of the Dissertation

Due to the ever increasing complexity of software systems, the field of software engineering is currently undergoing a period of heightened interest in formal methods and software reuse. At the same time, functional programming languages, often espoused as rapid prototyping tools, are beginning to enjoy more mainstream usage. Assuming that these trends
continue, software developers will need improved methods to transform existing specifications into purely functional programs.

Therefore, this research prescribes a methodology for translating Z specifications to Haskell implementations. The methodology is based on three methods:

(1) Hursley method [Word92], an established method for Z refinement

(2) FunZ, a new intermediate specification language defined in this work

(3) Dijkstra's guarded command language [1975]

This is the only methodology to use an intermediate specification language to bridge the gap between Z specifications and purely functional programs. Much of the previous work in transforming Z to code was conducted in order to validate user requirements and to quickly produce prototypes. However, now that functional programming languages are being used for larger applications, a more comprehensive approach is also required. The FunZ methodology meets this need by providing a complete and formal methodology.

To ease the job of both the designer and the implementor, FunZ integrates features from Z and Haskell. Furthermore, the associated methodology spans the entire software life cycle, from specification through design to implementation, thereby affording the software developer with a systematic means of recording all design and algorithmic decisions. As a result, the methodology surrounding FunZ is additionally of benefit during the maintenance phases of software development.

A review of each of the chapters in the dissertation follows:

Chapter 1 has presented the preliminaries needed to understand this dissertation. In particular, summaries of both functional programming and formal methods have been communicated. Furthermore, because Haskell and Z are the respective programming and specification
languages used in the research, more detailed information about these languages has been furnished.

Chapter 2 discusses other research that combines formal methods and functional programming. The chapter opens with a description of several methods developed primarily for rapid prototyping. Next, three formal techniques, each unique in its own way, are described. The last section depicts how FunZ and its associated methodology fit into the overall scheme of existing approaches that target the functional programming paradigm.

Chapter 3 relates the primary results of the research. In other words, it provides the reader with a definition of both the intermediate specification language FunZ and the general methodology. Furthermore, the chapter includes additional insights into the reasons for selecting Z as the initial specification language and Haskell as the final implementation language.

Chapter 4 describes two case studies from their initial specifications to their final implementations. The first study uses Z during the specification and design stages of software development, whereas the second example illustrates the FunZ methodology as described in Chapter 3. The reason for including the first study is to provide an understanding of the evolution of the intermediate language FunZ. In addition, by writing one design in Z and the other in FunZ, it is possible to demonstrate many of the advantages that FunZ holds over Z as a design language for purely functional programs.

Finally, chapter 5 begins with a summary of the dissertation. Moreover, the chapter describes the significance of the research and discusses several ideas for future investigation.
Chapter 2

Related Research

2.1 Prototyping

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.

In software engineering, the term animation usually refers to the quick conversion of existing formal specifications into executable prototypes. One of the reasons that animation techniques have proved popular is that they represent an attractive means to validate user requirements. Furthermore, other advantages of prototyping (e.g., increased confidence among software development team members and early detection of errors) are often achieved with animation.

Recently, several examples of animation have appeared in the literature. For instance, [Dill90] translates the Z specification of a telephone database to Miranda, whereas [O'Ne89] animates a VDM specification for an address book, also in Miranda. Meanwhile, in related work, [Nort90] implements VDM sets and maps as Miranda abstract data types to aid software developers in creating prototypes that more closely resemble their original VDM specifications.

As another example of animation, O'Neill [1989] uses Standard ML (SML) as a platform for illustrating VDM specifications. Because SML includes references, special values similar to the variables of an imperative language, an SML program can represent the state changes of
a model-based specification better than a corresponding prototype in Miranda. Moreover, [O'Ne92] describes how to mechanically translate VDM specifications into SML code by means of a VDM syntax-directed editor. Although this method offers advantages such as freedom from implementation bias and consistency among translations, the target language itself is not a purely functional language, nor does it possess a lazy semantics.

For those software developers who prefer modern, lazy functional languages, Goodman offers an alternative animation technique based on monads [Wadl92]. In [Good93], he defines a monad to handle the input, output, and state features of Z specifications, implements the monad in Haskell, and then demonstrates his technique by translating a simple specification into a corresponding Haskell program. Despite the fact that a monad is well suited to specifications in which the state plays a major role (primarily because the state variables no longer have to be passed around explicitly), its usage does have one drawback. In particular, guarded function definitions as supported by Haskell can not be employed to animate the preconditions of Z operation schemas. Therefore, using a monad for animation purposes results in a trade-off; to achieve code that more closely models state changes, one must make significant alterations to the format of the predicates in the original specification.

In contrast to animation where the resulting prototype is most likely a throwaway, some software developers prefer an evolutionary approach. Joosten describes a methodology [Joos89] based on the iterative production of prototypes in a lazy, functional programming language. With this method, the first step is to write both assertive and constructive specifications* using basic mathematical notation as opposed to a dedicated specification language. Then, from the constructive specification, the software developer codes an initial prototype, and from this prototype a successor. The propagation of prototypes continues, where each

* An assertive specification is an intuitive, possibly nonexecutable one, whereas a constructive specification describes a way to achieve an implementation.
new prototype improves on the computational efficiency of its predecessor, until a final prototype containing sufficient detail for an actual implementation is achieved. An important aspect of this method is that the specification evolves along with the prototype, and every effort is made to keep the two as similar as possible. In fact, Joosten uses several symbols from the Bird-Meertens algebra [Bird87; Bird90], a notation and associated theory for calculating functional programs from their specifications. Although the mathematical symbols cultivate concise specifications and corresponding proofs, the symbols themselves are not executable. Furthermore, those uninitiated to such calculi may feel somewhat intimidated by the amount of material that must be mastered in order to become proficient at deriving algorithms from their specifications.

A common feature of both animation and Joosten's method is that a prototype is derived from the initial specification. Meanwhile, in the more traditional approach to prototyping, the final prototype serves as a foundation for developing the overall system specification [Somm92].

One established technique for prototyping is to use executable specification languages. Because of their mathematical basis, functional languages are often recommended [Turn85a; Somm92], but languages that combine features from both specification languages and lazy, functional languages are also available. Two such languages are SAMPÆE and me too.

SAMPÆE [Jäge88] is a prototyping language whose design was influenced primarily by Miranda and META IV of VDM. A unique feature of the language is that it forms the basis of a complete, interactive prototyping environment [Henh91], which includes facilities for editing, type checking, interpreting, compiling, and debugging programs. Furthermore, SAMPÆE supports the reuse of software components by providing a general interface for functions and data types written in C. On the other hand, one disadvantage of SAMPÆE is that the language is referentially opaque since it contains imperative features such as traditional control
structures and assignment statements. Therefore, correctness proofs are more difficult in SAMPLE than with a purely functional programming language.

Interestingly, the language me too is also based on Miranda and META IV. However, unlike SAMPLE, me too forms the foundation of a general software design methodology. As described in [Hend86], the methodology is an iterative process comprised of three steps:

(1) The model step. The software developer chooses abstract data objects and operations to represent the structure and behavior of the software system.

(2) The specify step. The developer uses abstract data types and recursion equations to precisely describe the objects and operations from the model step.

(3) The prototype step. The prototype is executed in order to validate the system design.

In summary, me too is an example of evolutionary prototyping where the final prototype serves as the system specification. The fact that one language is used for both specification and prototyping can be seen as both an advantage and a disadvantage. The advantage is that a single language expedites the tracing of each formal requirement to its respective implementation. The disadvantage is that the prototype may eventually contain too much algorithmic detail, thereby curbing its overall effectiveness as a specification [O’Ne89].

2.2 Formal Program Development

This section describes three formal software development techniques that target functional programming languages. Each method has a distinct characteristic. In particular, the first uses a wide-spectrum language, the second is an example of a Larch interface language, and the third combines properties of animation with formal transformation techniques. Compared to the approaches of the previous section, prototyping plays a less prominent role during the software development process.
2.2.1 Extended ML

Extended ML [Sann85, Sann91] is a formal specification language and associated methodology for developing Standard ML (SML) programs. The language, which is based on the principles of algebraic specifications, is classified as wide-spectrum because it allows high-level specifications and executable programs to be expressed in the same framework [Sann90].

One of the advantages of Extended ML is that it extends SML with just two new language constructs, namely the axiom and the place-holder ?. The minimum number of additional features required is primarily due to the rich module facility of SML which supports a top-down approach to software development by allowing modules, known as structures, to be constructed from existing modules. Furthermore, a programmer can write the interface or signature of a structure before deciding on its actual implementation because signatures and structures are separate entities in an SML program.

Building on the module facility of SML, an initial specification in Extended ML consists of a functor, the name for a parameterized module in SML, but the functor’s signatures contain axioms and the functor itself returns a place-holder instead of a structure. Axioms and place-holders continue to play a prominent role throughout the intermediate stages of software development. In particular, the software developer is able to delay implementation decisions by using the symbol ? in place of type expressions, value expressions, or structure expressions. At the same time, he or she can write axioms for the structure bodies as well as their signatures. Note that the signature axioms specify properties of types and functions that will be implemented later in a corresponding body, whereas structure axioms take the place of code, thereby providing another means of postponing design decisions.

Finally, as outlined in [Sann90], the methodology of Extended ML consists of three primary steps:
\begin{itemize}
\item \textit{Decomposition step.} Decompose the functor into subsidiary functors that are specifications of smaller programming tasks.
\item \textit{Coding step.} Provide a functor body containing type and value declarations. The body may contain a mixture of code and axioms to define the declarations.
\item \textit{Refinement step.} Refine the functor body by replacing some axioms or place-holders.
\end{itemize}

To obtain an executable program, the software developer repeatedly selects one of the steps above until there are no longer any axioms or place-holders remaining in the specification or, in other words, only Standard ML code remains. Furthermore, to prove that the derived program is a correct implementation of its original specification, each step requires one or more proof obligations. As pointed out in [Sann90], these proofs may be performed following the completion of each individual step or after the construction of the entire program.

\subsection*{2.2.2 Larch/ML}

Larch/ML [Zare92] is one of several interface languages [Wing87; Jone91; Gutt91] from the Larch family of specification languages [Gutt85]. Each Larch interface language targets a specific programming language, incorporating its notation in order to better communicate state transformations such as side effects and resource allocations, as well as unique language features such as exception handling and concurrency. One benefit from using a specialized interface language over a generic one is that the resulting specification is usually more concise. Furthermore, because the specification targets a particular language, it is easier for the software developer who implements the program unit and clearer to the programmer who uses it [Gutt90]. Before describing Larch/ML, the following discussion continues with a brief overview of Larch.
Larch is known as the two-tiered specification method because each specification consists of two components: one written in the appropriate interface language and one written in the Larch Shared Language (LSL), an algebraic specification language. Whereas the interface language portion contains the necessary information to implement a module and to use it, the LSL component describes the basic types and operations of the interface part. Furthermore, due to its easier semantics, Larch designers recommend LSL for specifying the more difficult parts of a specification [Gutt90].

The two-tiered approach to specification offers several advantages. First, a component for state transformations and one for the underlying mathematical abstractions provides an automatic means of organizing specifications by separating concerns. Second, because LSL components and interface components can be written independently, it is easier to subdivide specification tasks among software development team members. Third, LSL components promote reusability since they can be used by different interface components [Wing87]. Moreover, the definitions for many commonly used concepts (e.g., lists, sets, stacks, queues, arrays, and partial orders) appear in LSL handbooks [Gutt90] as predefined traits. Finally, the Larch Prover [Garl90, Garl91] allows users to debug their LSL specifications.

A Larch/ML specification, like an Extended ML one, is based on the module facility of Standard ML. In particular, each specification consists of an SML signature with additional information provided in the interface. As is common with an interface component, the specification starts with a using clause that lists all traits required for the interface. Next, Larch/ML associates each SML type with an LSL sort by means of a based on clause. Meanwhile, requires and ensures clauses specify pre- and postconditions of all declared operations. Furthermore, each modifies clause designates the variables whose values may change as a result of a particular operation.
To handle possible naming conflicts among the traits, Larch/ML extends SML's mechanism of qualified identifiers by permitting trait names to qualify operation identifiers. Moreover, all specification statements appear as special comments delimited by (*+ and +*), an extension of the usual comment notation of SML, namely (* ... *). By applying this simple syntactic convention, a standard SML compiler is sufficient to compile any Larch/ML specification.

The designers of Larch/ML describe some of its advantages in [Zare92]. For example, the fact that Larch/ML highlights the state changes of its corresponding SML module is considered helpful to programmers who want to reuse existing modules. In particular, the extra information in each interface means that programmers are less likely to overlook unusual side effects and state changes, a distinct possibility since many SML programmers are accustomed to using a purely, functional subset of SML. Another benefit of Larch/ML is that it can be used to describe the semantic properties of basic data types, thus enhancing the formal semantics of SML. Furthermore, Larch/ML has at least one advantage over the specification language Extended ML in that Larch/ML is suitable for specifying references and assignments, while Extended ML only supports a small, purely functional subset of SML [Sann91].

2.2.3 From Z to Lazy ML

The Systems and Software Engineering Division at British Telecom Research Laboratories has developed a method for converting Z specifications into Lazy ML [Augu89] programs. As described in [John90], the method is a hybrid approach that blends prototyping with traditional transformation techniques in order to reap the advantages of both, as well as to eliminate their respective drawbacks. An outline of the method, which consists of five primary stages, follows.

As in the traditional software life cycle, the first stage involves the analysis of both user and system requirements, followed by the composition of the specification document. When
the resulting specification is written in an implicit style, the second stage consists of refining this initial specification to a constructive form. Although the usual proof obligations [Spiv89] must be met to show that the constructive specification is equivalent to the original, the refinement itself should be minimal. In particular, to facilitate rapid prototyping, abstract types should remain unchanged and a simple algorithm should be selected. Note that if the specification is already in an explicit form, stage two can be omitted.

The transliteration from Z to Lazy ML constitutes the third stage. Although the transliteration process itself has not been formalized, several important properties about the overall process appear in [John90]. Among these are the following:

1. Generic schemas can be implemented by employing polymorphism.
2. Functions from the Z Mathematical Toolkit [Spiv89] can be expressed in a functional language.
3. Certain set expressions can be translated to list comprehensions.
4. Nondeterministic Z specifications can be simulated with functions that return a list of solutions.

At the completion of this stage, the software developer has an executable prototype that serves as the initial program. However, if this initial prototype fails to meet the necessary space and time requirements, better performance can be obtained by applying the transformation rules of Burstall and Darlington [1977a]. The application of these traditional transformation rules corresponds to stage four of the development process. Lastly, stage five consists of the translation of the final functional program to an imperative implementation. (Note that Johnson and Sanders had not attempted this definitive translation procedure at the time their paper was accepted to the 4th Z User Workshop.)
2.3 A Frame of Reference for FunZ

The common denominator of the software development approaches presented in section 2.1 is the interconnection of formal specifications and functional programming primarily to simplify rapid prototyping. Traditionally, prototypes have been inefficient and imperative programs were usually produced at some later date. In contrast, a distinguishing characteristic of the FunZ methodology is that it targets software developers who prefer a purely functional programming language as the final implementation language. Although prototyping is possible within the framework of FunZ, it was not the motivating factor behind its design; rather, FunZ blends features from Z and Haskell to form a bridge between Z specifications and Haskell programs.

As a rule, animation approaches are largely informal—their link with formal methods merely the fact that the initial specification is written in a formal specification language. In the case of Joosten's method [1990], specifications are composed with mathematical notation, but the overall design process is somewhat ad hoc. Although some software developers may consider this an advantage in that they are not constrained by the syntax of a particular specification language and corresponding refinement rules, the current direction in software engineering is the application of formal methods throughout the software life cycle. Therefore, the methodology associated with FunZ assists software developers by providing systematic guidelines at each stage of the development process.

It is hard to compare SAMPLE to FunZ since their respective design goals are entirely different. Suffice it to say that in spite of the fact that SAMPLE is an excellent prototyping medium, it does not have a corresponding methodology. Meanwhile, me too is a formal methodology that employs an executable specification language of the same name as its primary support mechanism. Although many applications have been successfully designed using me too [Alex90], the methodology is limited to the specification and design stages of software
Unlike me too, the FunZ methodology not only directs users from an abstract specification to a concrete design, it also guides them from the design stage to a final implementation.

In section 2.2, three formal approaches to software development were discussed. Of these three, Extended ML is the most complete. The Extended ML methodology spans the entire life cycle, while the language itself has a firm foundation in algebraic semantics [Sann89]. One of the advantages of Extended ML is that the specification language is a small extension of Standard ML, but the corresponding methodology relies heavily on the notion of SML functors. Since SML is the only programming language to have such a sophisticated module system, it is difficult to adapt the Extended ML methodology to different languages. In contrast, FunZ targets Haskell, but the approach itself is general and easily tailored to other purely functional languages.

Like Extended ML, Larch/ML also targets the programming language Standard ML. As explained in section 2.2.2, the two-tiered approach of Larch offers several benefits. Notwithstanding such advantages as separation of concerns and support for reuse, Larch does not include a general methodology for writing an interface specification. One could perhaps argue that the methodology is implicit—that is by learning a Larch interface language, one inevitably learns how to compose a specification for the corresponding target language. However, most software engineers would agree that a methodology is a step-by-step guide to solving a problem. The FunZ approach satisfies this definition.

Another important point to remember is that SML is not a purely functional language. In fact, the designers of Larch/ML maintain that the "impure" features of SML are crucial to their current application domain, which involves concurrency and persistence [Zare92]. At any rate, no existing Larch interface language corresponds to a purely functional programming language.
Of all the related approaches, Johnson and Sander's work [1990] (referred to as the JS procedure hereinafter) is the most relevant to FunZ, because both the JS procedure and the FunZ methodology explain how to translate an initial abstract Z specification to a purely functional programming language. However, there are several significant differences between the two approaches. First, with the JS procedure, minimal changes are made to the Z specification because one of the goals of this procedure is to obtain a working prototype as quickly as possible. In contrast, with the FunZ approach, users first refine an abstract Z specification to a concrete FunZ specification, which forms the skeleton of a Haskell program, and then make minimal changes to the FunZ specification to achieve the final program. Second, the translation phase from specification to code is not formalized in the JS procedure, whereas with the FunZ methodology, both of the translation phases (from Z to FunZ and FunZ to Haskell) use a formal approach. Third, the traditional fold/unfold transformation rules comprise the theoretical foundation of the JS procedure. Meanwhile, the FunZ methodology is an adapted form of the Hursley method [Word92] (a Z refinement method developed at IBM) and Dijkstra's guarded command language [1975].

Table 1 (see next page) provides a comparison of the current approaches that combine formal methods and functional programming. The formalism present in these procedures spans the gamut from animation to Extended ML. Although many of the approaches either target particular programming languages or borrow features from popular specification languages, only a few (namely Extended ML, the JS Procedure, and FunZ) furnish a complete methodology.

Compared to the existing software development approaches that target purely functional programming languages, the FunZ methodology is the most comprehensive. Of considerable importance is the fact that the methodology provides a systematic means for recording design decisions throughout the software life cycle. Since FunZ and its methodology are expressly
<table>
<thead>
<tr>
<th>Approach</th>
<th>Distinguishing Characteristic</th>
<th>Language Basis</th>
<th>Formalism</th>
<th>Complete Methodology</th>
<th>Advantage</th>
<th>Weakness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Animation</td>
<td>throw-away prototyping</td>
<td>—</td>
<td>starts with formal</td>
<td>no</td>
<td>quick prototype</td>
<td>informality</td>
</tr>
<tr>
<td>Joosten</td>
<td>evolutionary prototyping</td>
<td>—</td>
<td>mathematical notation</td>
<td>no</td>
<td>concise notation &amp; programmer</td>
<td>limited use of formal methods</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VDM &amp; Miranda</td>
<td>VDM-like language</td>
<td></td>
<td>environment &amp; interface to C</td>
<td>imperative features</td>
</tr>
<tr>
<td>me too</td>
<td>prototyping language</td>
<td>VDM &amp; Miranda</td>
<td>VDM-like language</td>
<td>design</td>
<td>excellent method for prototyping</td>
<td>final code is not mainstream</td>
</tr>
<tr>
<td>Extended ML</td>
<td>wide-spectrum language</td>
<td>Standard ML</td>
<td>algebraic specifications</td>
<td>yes</td>
<td>small extension of Standard ML</td>
<td>dependent on functors</td>
</tr>
<tr>
<td>Larch/ML</td>
<td>specification language</td>
<td>Larch &amp; Standard ML</td>
<td>two tiers of Larch</td>
<td>no</td>
<td>Larch Prover for Shared Language</td>
<td>no associated methodology</td>
</tr>
<tr>
<td>JS Procedure</td>
<td>prototyping &amp; transformation</td>
<td>Z &amp; Lazy ML</td>
<td>fold/unfold trans. rules</td>
<td>yes</td>
<td>extends simple animation</td>
<td>Z → design (informal)</td>
</tr>
<tr>
<td>FunZ</td>
<td>intermediate spec. lang.</td>
<td>Z &amp; Haskell</td>
<td>adapted form of Hursley method</td>
<td>yes</td>
<td>formalism at each stage of life cycle</td>
<td>lack of tool support</td>
</tr>
</tbody>
</table>

Table 1. A Comparison of FunZ to Related Approaches
designed for the functional programming paradigm, both design and implementation are simplified. Furthermore, the associated documentation supports maintenance steps, as well as the reuse of FunZ design components.

The Hursley method is an established procedure for refining Z specifications into a more concrete form. Furthermore, Dijkstra's guarded command language has a successful history as a representation form for algorithms that target imperative programs. Some researchers have experimented with enhancing these methods, often with the purpose of easing the translation from Z to a particular programming language. [Neil89] describes a rigorous refinement method for Z specifications written in a hierarchical style. Although this article only describes the process from Z to guarded command code, the author's dissertation includes an example from Z to C. Meanwhile, [Wood91] presents a method for transforming Z specifications to Ada programs via the intermediate language ANNotated Ada [Luck87]. In yet another approach, [Senn92] integrates the ideas of the refinement calculus [Morg90] and Knuth's literate programming to verify that Ada programs are correct with respect to their Z specifications.

Research on deriving imperative code from Z specifications has not been limited to standard Z. Cases relating object-oriented extensions of Z and object-oriented programming languages have recently appeared in the literature. For example, [Rafs93] defines a structural mapping from Object Z [Carr89] to C++. Note that this approach does not include a translator for predicates. Meanwhile, [Cord94] describes a much more extensive procedure for translating MooZ [Meir91] specifications to Eiffel [Meye92] programs. With this method, an abstract MooZ specification is first converted into a more concrete MooZ specification using the Hursley method. The next stage is a structural refinement that produces a combination of Eiffel code and specification statements from the refinement calculus. Finally, the last stage includes the application of refinement calculus rules to obtain only Eiffel code.
In conclusion, research continues on how to best translate an abstract Z specification into a more concrete form and finally to an imperative program. The most popular approaches are the Hursley method, the refinement calculus, or combinations of the two. Previously, there was no similar method for functional programs. Now that functional programming compilers are efficient enough for mainstream applications, a formal methodology is needed for constructing purely functional programs from Z specifications. FunZ and its associated methodology meet this need.
Chapter 3

FunZ: The Language and Associated Methodology

3.1 Motivation

As discussed in the previous chapter, functional programming languages have established themselves as excellent tools for rapid prototyping. In addition, the mathematical basis of purely functional languages has proved complementary to the derivation of programs from initial specifications by means of correctness-preserving transformations [Boit92].

With the increased emphasis on formal methods throughout the software life cycle, the use of purely functional languages should become even more prevalent in the future. Furthermore, as compilers continue to improve, these languages are likely to serve not only as interim, design tools but also as final implementation languages. In fact, several medium to large applications are currently under development or have already been programmed in a purely functional language, as evidenced by a recent conference on functional programming in the real world [Gieg94]. Among the examples presented at the conference were a spreadsheet program written in Clean and a functional programming environment called Natural Expert that supports the construction of knowledge based systems. Additional applications, which have appeared recently in the literature, follow.

Lolita [Haza93], a large system for natural language processing, was originally programmed in Miranda but has since been translated to Haskell. Clio and Spectool, two verification tools developed at Odyssey Research Associates, are based on Caliban [Kell89], a Miranda-like language. Both tools played a major role in the first phase of verifying a hardware component for the Fault Tolerant Parallel Processor [Sriv92]. As another example, a
significant portion of a software system for automatic speech recognition [Gobl94] was coded in Haskell.

The emergence of purely functional programming languages for real-world applications along with the heightened interest in formal methods and software reuse helped to inspire the research presented in this dissertation. In particular, the intermediate specification language FunZ [Sher95] and an associated methodology were designed to aid software developers in deriving purely functional programs from existing Z specifications.

Several languages have influenced the design of FunZ including Larch [Gutt85], COLD-K [Feij92], Extended ML [Sann91], and Object-Z [Carr89]. Ideas from [Schu87] have also played a role in its development. However, FunZ is best described as an extension of Haskell with a Z-like flavor because it preserves many of the notational conventions of standard Z and several object-oriented variants [Step92]. In addition, software design with FunZ is similar to that with Z, except that each step of the methodology 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. Particularly, 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 straightforward 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 sooner 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.

3.1.1 Why Z and Haskell?

Throughout their respective histories, specification languages and functional programming languages have been closely associated. Hope [Burs80a], an early functional language, includes parameterized modules that are based on those of CLEAR [Burs77b], the first algebraic specification language. At least two prototyping languages, me too [Hend86] and SAMPÆ [Henh91], 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.

As previously discussed, Z is a model-oriented specification language based on typed set theory and predicate logic. Since its initiation in the 1980's, Z has become increasingly popular. This popularity is corroborated not only by the numerous conference and journal articles devoted to software development in Z, but also by the fact that several textbooks [Dill90; Pott91; Ince92; Word92] are now available on the subject. Z was selected as the initial specification language, partly due to its widespread acceptance but more importantly, because of its strong mathematical foundation as substantiated in a formal semantics [Spiv88]. In addition, Z possesses a unique calculus that facilitates the incremental development of specifications.

Haskell [Huda92b] is a purely functional programming language that closely resembles Miranda, yet 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, it was chosen as the implementation language because of its endorsement by the functional programming community as
a standard for non-strict, purely functional languages. Furthermore, there are several features of Haskell that distinguish it as a good target language for Z specifications.

First, a Haskell implementor need not specify flow of control because there are no side effects in a purely functional language. This characteristic corresponds nicely with Z since a specification contains no information about the evaluation order of its operations. Second, a Haskell programmer can define infinite data structures due to lazy evaluation. Since many Z specifications contain infinite sets, the transition from Z to code is easier in a lazy language. An additional benefit of lazy evaluation is that a programmer is unencumbered by efficiency concerns because Haskell expressions will only be calculated when their values are required.

Third, the overall translation process is tractable because Z and Haskell are both strongly typed. Algebraic datatypes can be used to represent objects from the Z specification, which keeps the program at a very high level, or built-in types such as lists, tuples, and arrays can replace the abstract objects. A Z type checker will discover typing errors in the Z specification, and if typing errors occur in the translation from Z to Haskell, a Haskell compiler should uncover these errors at compile time.

Fourth, since the major Haskell implementations from Yale, Chalmers, and Glasgow support literate programming*, a Z document can be included with its corresponding Haskell program to provide a complete history of a software project. Moreover, because Haskell does not require a particular sequencing for its function definitions, the definitions may appear in the order most conducive to readability with the Z text.

In summary, the declarative style of Haskell renders it a logical choice as a target language for Z specifications. Furthermore, the translation from specification to code is easier than with an imperative language due to the inherent properties of Haskell.

* To use the literate programming style in Haskell, one simply types the symbol > as the first character in each line of program code. The effect is all lines without the designated symbol are treated as comments.
3.1.2. 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: 1) proving the concrete or lower level specification is consistent with the abstract version and 2) verifying the code against the concrete specification. Furthermore, by following refinement guidelines, the developer documents his or her design decisions.

A popular method [Word92], 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 [Morg90], and then apply the laws of the calculus to derive guarded command programs. In [King90b], the translation occurs after data reification in Z, while [King90a] initiates the change in notation one step sooner by converting the abstract specification schemas to the refinement calculus.

Meanwhile, prior to the development of FunZ, this research included an investigation to study the effects of changing from Z to Haskell at different stages in the software life cycle. The earliest experiments applied 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, whereas 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. Note that Chapter 4 of this manuscript includes a description of one of these designs, while [Sher93; Sher94] recount supplementary designs and their corresponding translations.
To formalize this early work, two problems were proposed: 1) adapt the traditional refinement approach [Spiv89] in order to give it a more functional flavor and 2) define a general set of transformation rules for converting Z designs to Haskell. While attempting to solve these problems, a collection of mappings of the following form was developed.

\[
\text{Z set operators } \rightarrow \text{Z sequence operators } \rightarrow \text{Haskell list functions}
\]

As an example, if one considers lists without duplicate elements, set difference \(\backslash\) maps to range subtraction \(\triangleright\), which in turn maps to list difference \(\\backslash\\). Due to 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.

The methodology encompassing FunZ is the first formal method to employ an intermediate language when translating Z specifications to functional implementations. However, [Wood91] describe a formal approach in which Z specifications are initially transformed to ANNotated Ada (Anna) [Luck87] and then to Ada code. Although the transformation rules of their approach are language independent, the advantage in targeting Ada programs is twofold: the existing specification language Anna serves as the intermediate language and the Anna 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 an algebraic datatype in Hope, whereas \(|\) is the correct Haskell syntax. As another example, -- is the list difference operator in Miranda, while \(\\backslash\\) is used in Haskell.
3.2 Definition of FunZ

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

- The language should be a straightforward extension of Haskell.
- The language should be conducive to specifying the characteristics of a purely functional programming language.
- The language should preserve the notational conventions and structuring facilities of Z.

Furthermore, an associated methodology, patterned after the Hursley method of IBM [Word92], would be developed simultaneously.

To satisfy the design constraints of FunZ, features from both Haskell and Z were integrated. The result is a specification language that syntactically resembles Haskell, but semantically matches Z. In addition, fashioned for those designers and implementors who prefer functional programming languages, FunZ allows software developers to model state operations more 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 a 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 favor of a syntax patterned after Haskell modules.

Much the same as Z, each FunZ design consists of state and operation spans that refine the corresponding schemas of the Z specification. Unlike Z, the generic 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. First, note that Figure 1, which immediately follows, contains a template for a typical state span.

```
span <Span_Name> where
<declaration part>
inv is
<invariant part>
rel is
<relation part>
init is
<initialization part>
end span <Span_Name>
```

Figure 1. Template for a FunZ 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.

The initial phrase of a span was designed to complement that of the Haskell module. The keyword `span` introduces the concrete specification or design unit, while the keyword `where`
precedes the declarations. Each occurrence of \texttt{is} initiates the layout rules of Haskell. In other words, semicolons are not required to separate declarations or predicates. Rather, the items of a particular component may appear one per line, as long as they are aligned in the same column, and this column appears to the right of the keyword introducing the associated component. To distinguish the boundary of one specification unit from another, each span contains the terminal phrase \texttt{end span} with the appropriate span name.

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 a FunZ operation span and its Z counterpart. The most important change is the \texttt{modifies} declaration, as it represents not only an alteration in syntax but also one in semantics. Meanwhile, the predicate part is analogous to that of Z.

\begin{verbatim}
span <Span_Name> where
  modifies <Schema_Name> ( <state_vars> )
  <input variables>
  <output variables>
  pred is
    <predicate part>
end span <Span_Name>
\end{verbatim}

Figure 2. Template for a FunZ Operation Span

The \texttt{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 \texttt{modifies} tuple. As an example, consider the following modifies clause that forms a part of span \texttt{Testok}, which is developed in Chapter 4.

\texttt{modifies Class (ns, ts)}

The declaration denotes that all the variables and predicates of spans \texttt{Class} and \texttt{Class'} are visible, yet only the values of variables \texttt{ns} and \texttt{ts} may vary. If no state values should change
as a result of an operation, then the \texttt{modifies} tuple is simply written as ( ). In other words, the expression \texttt{modifies <Schema_Name> ( )} replaces the \texttt{Xi (E)} convention of \texttt{Z}. Note that \texttt{span Class} is not related to the \texttt{class} construct of Haskell. Although \texttt{class} is a reserved identifier, there is no naming conflict since variable names are case sensitive in Haskell.

Another important point concerning the \texttt{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 \texttt{modifies} tuple is meant to reflect that the variables will need to be actual parameters in a Haskell implementation.

### 3.3 FunZ Syntax

The intermediate language FunZ is a straightforward extension of Haskell. The primary additions include the \texttt{span} construct and predicate logic operators that allow for the incremental construction of FunZ specifications.

The syntax for FunZ is written with a BNF-like metalanguage, where each production has the form:

\[
\text{<nonterm>} \rightarrow \text{choice}_1 | \text{choice}_2 | \text{choice}_n
\]

To promote readability, several conventions are used. In particular, the notation \texttt{V\_1 V\_2 ... V\_n} represents \texttt{n} instances of a syntax class that are separated by commas. Similarly, the phrase \texttt{V\_1 NL V\_2 NL ... NL V\_n} stands for \texttt{n} instances of class \texttt{V}, one per line (NL is the symbol for newline).

All keywords for FunZ appear in typewriter font. Additional classes of terminal symbols, where \texttt{H} indicates Haskell code and \texttt{Z} denotes Z notation, include the following:
The formal syntax rules for FunZ follow:

```
<document> → <paragraph> NL ... NL <paragraph>
<paragraph> → <global_var> | <span>
<global_var> → <basic_set> | <constant> | <enum_type> | Htype_syn
<basic_set> → basic <c_ident>
<constant> → Hvar :: Htype where <constraint>
<constraint> → Hexp
<enum_type> → data Huser_type
<span> → span <spanid> where
   <spanbody>
   end span <spanid>
<spanid> → <c_ident>
<spanbody> → <statebody> | <opbody> | <combody>
<statebody> → <statevars> NL <invariant> NL <relpart> NL <initpart>
<statevars> → <svar> NL <svar> NL ... NL <svar> (n ≥ 1)
<svar> → Hvar :: <gen_type>
<gen_type> → Htype | Bset
```
<invariant> → inv is <preds>

<preds> → <pred₁> NL <pred₂> NL ... NL <predₙ> (n ≥ 1)

<pred> → True | False | (pred) | not <pred>
| <pred> & <pred> | <pred> || <pred>
| <pred> implies <pred>
| ∀ spantext • pred
| ∃ spantext • pred
| <pred> == <pred> | Hexp

<relpart> → rel is NL <retmaps>

<retmaps> → <map₁> NL <map₂> NL ... NL <mapₙ> (n ≥ 1)

<map> → <declmap> NL <pairs>

<declmap> → <mapid> :: <gen_type> -> Ztype

<mapid> → abmapᵢ (1 ≤ i ≤ n)
(• n = no. of retrieve functions * )

<pairs> → <pair₁> NL <pair₂> NL ... NL <pairₙ> (n ≥ 1)

<pair> → Zabvar₁ = Retfun₁ Hexp
Zabvar₂ = Retfun₁ Hexp
...
Zabvarᵢ = Retfunᵢ Hexp (1 ≤ j ≤ m)
(• m = no. of abstract state var. *)
(• n = no. of retrieve functions *)

<initpart> → init is
(varid₁, varid₂, ..., varidₙ)
(val₁, val₂, ..., valₙ)
varidᵢ ∈ Hvar
valᵢ ∈ Hval
(n ≥ 1)

<opbody> → <declpart> NL <predpart>

<declpart> → <modifies_cl> NL <input_vars> NL <output_vars>
3.4 Overview of the Methodology

When deriving purely functional programs from Z specifications by means of the intermediate specification language FunZ, there are two principal phases in the overall process. The initial phase encompasses the refinement of the abstract Z specification to a concrete FunZ specification, whereas the successive phase covers the transformation of the FunZ specification to a Haskell program. Figure 3 provides a graphic representation of the entire software process by highlighting these two phases. Note that the names of the phases, data refinement and algorithmic refinement, reflect the fact that the FunZ methodology is an adapted form of the Hursley method [Word92].
Traditionally, the initial phase of software development (from an abstract Z specification to a concrete Z specification) is further subdivided into two stages: data refinement and operation refinement [Pott91]. During data refinement, data structures of the target language are first selected to represent the objects of the abstract specification, and then, the correspondence between concrete objects and abstract objects is documented with a retrieve function. Upon the completion of data refinement, each abstract operation is converted into a concrete counterpart (operation refinement). Similarly, the second phase can be broken into two subparts: algorithmic refinement and implementation refinement. Algorithm refinement involves the usage of Dijkstra's guarded command language to bridge the gap between concrete specification and final program, whereas implementation refinement is the transliteration process from guarded command code to executable code of the target language.

The following outline presents a general overview of the corresponding process in FunZ. Note that the methodology assumes the existence of an initial abstract Z specification.

I. Translate the abstract Z specification into a concrete FunZ specification.

A. Translate the global variables of Z to FunZ equivalents.
B. Define a FunZ state span SS that is equivalent to the Z state schema.
   (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 function, which maps the concrete FunZ objects to the abstract
       Z objects, and place this in the relation part of SS. The mapping must be sur-
       jective 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 appropri-
       ate 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 the state variables whose values should change into the modifies tuple
       of OS.

D. Perform correctness proofs. (This step is optional).
   (1) Prove that each initial concrete state corresponds to an initial abstract state.
   (2) Prove that every operation span satisfies the safety condition.
   (3) Prove that every operation span satisfies the liveness condition.

II. Translate the FunZ specification to a Haskell program.
   A. Use Dijkstra's guarded command language (an adapted form) to express algorithms.
      (1) For each operation span whose body is a disjunction of spans, translate the dis-
          junction to an alternation expression.
(2) If it is necessary to perform operations in a particular order, use function composition.

(3) If an algorithm requires searching or iteration, use list comprehensions or recursion.

B. Transliterate the guarded command code to Haskell code.

In summary, the FunZ methodology parallels a development route entirely in Z, except that each stage has a more functional flavor in order to accommodate a final implementation in Haskell. Furthermore, at the end of phase one, much of the design process is complete because a FunZ specification forms the framework or skeleton of a Haskell program. This in turn effects phase two. In particular, the distance from concrete specification to code is less than with Z. The extra labels in Figure 3 (design and implementation) indicate this shift in emphasis when a software developer uses FunZ as opposed to Z.

3.5 Basic Features of the Methodology

This section summarizes the essential points of the FunZ methodology. In the course of the design phase, defining a state span requires the most creativity on the part of the designer. Haskell data types are chosen to represent the abstract objects of the Z specification and an invariant for the concrete state is contrived. Furthermore, the correspondence between concrete and abstract objects must be documented with a retrieve function. Since this abstraction mapping plays an integral role in the translation procedure for operation spans, the next subsection defines a specific retrieve function and some related theorems that are representative of those possible with the intermediate language FunZ. Meanwhile, section 3.5.2 describes the general procedure for deriving concrete predicates, and section 3.5.3 presents the necessary proof obligations for functional refinement in FunZ.

The second phase of the methodology utilizes Dijkstra's guarded command language as a means of recording the principal refinements from a concrete specification to an executable
program. This is similar to the Hursley method but, because FunZ targets the functional pro-
gramming paradigm, the traditional control structures of the guarded command language must
be adapted. In particular, function composition replaces sequencing, and recursion supplants
iteration. Due to the fact that the disjunction operator (∨) plays a major role in structuring
specifications, section 3.5.4 concentrates on the refinement of a general disjunction statement
to the alternation construct from the guarded command language. Finally, subsection 3.5.5
provides a review of the complete methodology.

3.5.1 A Retrieve Function

A retrieve function and collection of associated theorems are instrumental both in deriv-
ing FunZ specifications and in proving that the resulting designs refine their Z counterparts.
As a typical example, the remainder of this section defines function set and two theorems.
Theorem 1 relates the abstract operation of set union to the Haskell function for list concatenation ++, whereas Theorem 3 links set difference \ with list difference \ \.

Although this section only contains the theorems needed for the FunZ case study in
Chapter 4, function set and its auxiliary theorems are general in that they are applicable to an
entire class of FunZ specifications—namely, those designs that model sets by means of
Haskell lists. Since the list is the traditional data structure for functional languages, the mem-
bers of this class represent a significant portion of a universal set U containing all possible
FunZ designs.

Definition. The set function converts lists of type a to sets of type P a. Its definition
employs the same list notation as Haskell: [] for the empty list and (x:xs) for the nonempty
list, where x denotes the head of the list, xs its tail, and (::) is the predefined cons function.

\[
\begin{align*}
\text{set} & : \text{[a]} \rightarrow \text{P a} \\
\text{set} \; [] & = \; \{ \} \\
\text{set} \; (x:xs) & = \; \{x\} \cup \text{set} \; xs
\end{align*}
\]
The following proofs use the technique of structural induction over lists [Bird88]. To help clarify these proofs, as well as those in Chapter 4, Appendix A contains the definitions of all relevant Haskell functions. Note that each expression `fun.num`, appearing as justification for a proof step, designates pattern `num` in the definition of a function named `fun`. For example, `set.1` refers to the line `set [] = {}` in the above definition.

**Theorem 1.** If `xs` and `ys` are finite lists, then `set (xs ++ ys) = set xs ∪ set ys`.

**Proof:** The proof is by induction on `xs`.

**Case [ ].**

\[
\begin{align*}
set ([] ++ ys) &= set (foldr (: ) ys []) & (++) .1 \\
&= set ys & foldr .1 \\
&= {} \cup set ys & \{ \} \cup S = S \\
&= set [] \cup set ys & set .1
\end{align*}
\]

This establishes the case.

**Case (x:xs).**

\[
\begin{align*}
set ((x:xs) ++ ys) &= set (foldr (: ) ys (x:xs)) & (++) .1 \\
&= set ((:) x foldr (: ) ys xs) & foldr .2 \\
&= set (x : foldr (: ) ys xs) & (:) as an infix operator \\
&= set (x : (xs ++ ys)) & (++) .1 \\
&= (x) \cup set (xs ++ ys) & set .2 \\
&= (x) \cup (set xs \cup set ys) & Induction Hypothesis \\
&= ((x) \cup set xs) \cup set ys & Associativity of ∪ \\
&= set (x:xs) \cup set ys & set .2
\end{align*}
\]

This establishes the case. □

Although a correspondence between set union and list concatenation exists for all finite lists, an association between set difference and list difference requires that the respective lists satisfy an additional condition. To help express the constraint, a new definition along with a corresponding notation is introduced.

**Definition.** The *multiplicity* of an element `x` with respect to list `xs`, denoted by `[x]_{xs}`, is the number of times that `x` appears in `xs`. For example, `[5]_L = 2` when `L = [0,5,5,10,10,10]`. 
The following lemma is a special case of Theorem 3. In particular, it associates the abstract operator for set difference with the Haskell function `del` when a singleton set is the second set in a set difference.

**Lemma 2.** If $\text{xs}$ is a list and $|y|_{\text{xs}} \leq 1$, then $\text{set (xs 'del' y)} = \text{set xs \setminus \{y\}}$.

**Proof:** The proof is by induction on $\text{xs}$. Note that backquotes enclose function `del` to allow its use as an infix operator.

**Case $[]$.**

$$
\text{set ([]} '\text{del}' y) = \text{set []} = (\{\}) = (\{\}) \setminus \{y\} = \text{set []} \setminus \{y\} = \text{set.1}
$$

This establishes the case.

**Case $(x:xs), x = y$.**

$$
\text{set ((x:xs) 'del' y)} = \text{set xs} = (\{\}) \setminus \{y\} = \text{set xs \setminus \{y\}} \neq \text{set xs because } y = x \text{ and } |y|_{(x:xs)} \leq 1 \Rightarrow |y|_{\text{xs}} = 0
$$

This establishes the case.

**Case $(x:xs), x \neq y$.**

$$
\text{set ((x:xs) 'del' y)} = \text{set (x : (xs 'del' y))} = \text{set (x \cup (set xs \setminus \{y\}))} = (\{x\} \cup \text{set (xs 'del' y)}) = (\{x\} \cup \text{set xs \setminus \{y\}}) = (\{x\} \cup \text{set xs \setminus \{y\}}) = (\{x\} \cup \text{set xs \setminus \{y\}})
$$

This establishes the case. □

**Theorem 3.** Let $\text{xs}$ and $\text{ys}$ be lists. If $|z|_{\text{xs}} \leq |z|_{\text{ys}}$ for each element $z$ of $\text{ys}$, then $\text{set (xs \setminus ys)} = \text{set xs \setminus set ys}$.

**Proof:** The proof is by induction on $\text{ys}$. 
Case [].

\[
\text{set (xs } \setminus \text{ [ ] }) \\
= \text{ set (foldl del xs [ ])} \quad (\setminus).1 \\
= \text{ set xs } \quad \text{foldl}.1 \\
= \text{ set xs } \setminus \{ \} \\
= \text{ set xs } \setminus \text{ set [ ]} \quad \text{set}.1
\]

This establishes the case.

Case (y:ys).

\[
\text{set (xs } \setminus \text{ (y:ys))} \\
= \text{ set (foldl del xs (y:ys))} \quad (\setminus).1 \\
= \text{ set (foldl del (del xs y) ys)} \quad \text{foldl}.2 \\
= \text{ set ((del xs y) } \setminus \text{ ys)} \quad (\setminus).1
\]

There are three subcases to consider. Either \(y\) is not an element of \(xs\), \(y\) appears exactly once in \(xs\), or \(y\) appears multiple times. The subsequent discussion considers the case where \(y\) is not in \(xs\).

\[
\text{set ((del xs y) } \setminus \text{ ys)} \\
= \text{ set (xs } \setminus \text{ ys)} \quad y \text{ is not in } xs \\
= \text{ set xs } \setminus \text{ set ys} \quad \text{Induction Hypothesis} \\
= (\text{set xs } \setminus \{y\}) \setminus \text{ set ys} \quad y \notin \text{ set xs since } y \text{ is not in } xs \\
= \text{ set xs } \setminus ((y) \cup \text{ set ys}) \quad (S \setminus T \setminus V = S \setminus (T \cup V)) \\
= \text{ set xs } \setminus (y:ys) \quad \text{set}.2
\]

This establishes the first subcase. Now assume that \(y\) appears exactly once in list \(xs\). Continuing from above, the proof is as follows:

\[
\text{set ((del xs y) } \setminus \text{ ys)} \\
= \text{ set (del xs y) } \setminus \text{ set ys} \quad \text{Induction Hypothesis applies because } 0 = |y|_{\text{del xs y}} \leq |y|_{\text{ys}} \\
= (\text{set xs } \setminus \{y\}) \setminus \text{ set ys} \quad \text{Lemma 2} \\
= \text{ set xs } \setminus ((y) \cup \text{ set ys}) \quad (S \setminus T \setminus V = S \setminus (T \cup V)) \\
= \text{ set xs } \setminus \text{ set (y:ys)} \quad \text{set}.2
\]

This establishes the second subcase. The final subcase, when \(|y|_{xs} > 1\), follows.

\[
\text{set ((del xs y) } \setminus \text{ ys)} \\
= \text{ set (del xs y) } \setminus \text{ set ys} \quad \text{Induction Hypothesis applies because } \forall \ z \text{ in } xs \\
\quad |z|_{xs} \leq |z|_{y:ys} \Rightarrow |z|_{\text{del xs y}} \leq |z|_{ys} \\
= \text{ set xs } \setminus \text{ set ys} \quad |y|_{xs} > 1 \Rightarrow \text{set (del xs y) } = \text{ set xs} \\
= \text{ set xs } \setminus \text{ set (y:ys)} \quad y \text{ in } ys \Rightarrow \text{set ys } = \text{ set (y:ys)}
\]
In the preceding proof step, the justification assumes that \( y \) is an element of \( y_s \). Suppose, instead that \( |y|_{y_s} = 0 \). Then \( |y|_{y;y_s} = 1 \) and \( |y|_{x_s} > |y|_{y;y_s} \), which is a contradiction. This establishes the last subcase for \((y:y_s)\). □

3.5.2 A Procedure for Deriving Concrete Predicates

A fundamental requirement of the procedure for deriving concrete predicates in FunZ is that the retrieve relation must be a surjective function (i.e., each concrete state corresponds to exactly one abstract state and each abstract state has a concrete representation). This requirement is necessary because the retrieve function serves as a translator for the abstract objects in the Z specification. A prescription for the actual translation process, as it applies to the set function, follows.

First, the designer replaces each abstract object in a Z predicate with its corresponding concrete representation, as specified by the retrieve function. Next, he or she attempts to simplify the predicate by using the laws of set theory and theorems associated with the retrieve function. The strategy is to derive an intermediate predicate that matches one of two templates:

1. \(<\text{complex}\_\text{exp}> \ zop \ <\text{complex}\_\text{exp}>\)
   
   where \(<\text{complex}\_\text{exp}> ::= \text{simple}\_\text{id} \mid \text{set list}\_\text{exp}\)

2. \(\text{set list}\_\text{id}' = \text{set list}\_\text{exp}\)

After achieving a match, the designer should administer a corresponding guideline.

**Guideline 1:** If the intermediate predicate matches template (1), apply the appropriate transformation rules from Table 2, or one of the subsequent axioms, to obtain a feasible concrete predicate.

**Guideline 2:** If the intermediate predicate matches template (2) and Rule R6 applies, first simplify the predicate to \(\text{list}\_\text{id}' = \text{list}\_\text{exp}\). Then apply the necessary transformation rules from Table 2 to obtain a possible concrete predicate.
<table>
<thead>
<tr>
<th>Rules</th>
<th>$Z$</th>
<th>FunZ/Haskell</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td><code>varname?</code></td>
<td><code>varname</code></td>
</tr>
<tr>
<td>R2</td>
<td><code>varname!</code></td>
<td><code>varname</code></td>
</tr>
<tr>
<td>R3</td>
<td><code>enumid</code></td>
<td><code>Enumid Enumid is a type constructor</code></td>
</tr>
<tr>
<td>R4</td>
<td><code>∈</code></td>
<td><code>'elem'</code></td>
</tr>
<tr>
<td>R5</td>
<td><code>∉</code></td>
<td><code>'notElem'</code></td>
</tr>
<tr>
<td>R6</td>
<td><code>set xs</code></td>
<td>`xs xs is a list and ∀ z in list xs, (</td>
</tr>
</tbody>
</table>

To apply Rule R6, the multiplicity of each list element must be 1. However, the following axioms are more general in that they apply to all lists.

**Axiom 1.** If $xs$ is a list, then $x \in set xs \leftrightarrow x \ 'elem' \ xs$.

**Axiom 2.** If $xs$ is a list, then $x \notin set xs \leftrightarrow x \ 'notElem' \ xs$.

**Comments:** (a) The only difference between $xs$ and $set xs$ is that $xs$ may contain duplicate elements and the order of the elements is significant. (b) Axiom 1 incorporates Rules R4 and R6 from Table 2, while Axiom 2 combines Rules R5 and R6.

### 3.5.3 Functional Refinement

The proof obligations in FunZ are analogous to those of $Z$. For those readers less familiar with $Z$, the discussion below includes the standard set of conditions for functional refinement [Spiv89], as well as those adapted for FunZ.

A single proof obligation must be satisfied to show that every concrete initial state corresponds to an abstract initial state. The traditional proof obligation follows:

$$\forall Astate; Cstate \cdot Cinit \land Abs \Rightarrow Ainit$$

Note that each variable name corresponds to a $Z$ schema. In particular, $Astate$ and $Cstate$ respectively denote the abstract and concrete state spaces. Similarly, $Cinit$ and $Ainit$ specify concrete and abstract initial states. Finally, $Abs$ refers to the abstraction schema, another name for the schema containing the retrieve relation. Restating this in FunZ yields:

$$\forall Astate; StSpan \cdot StSpan: init' \Rightarrow Ainit'$$
The notation $\text{StSpan}\init'$ denotes 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 proof obligation does not explicitly refer to the retrieve function, its equations are visible due to the declaration of $\text{StSpan}$.

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 [Word92]. The safety condition ensures that whenever an operation on the abstract state $(Aop)$ terminates, its corresponding operation on the concrete state $(Cop)$ will also terminate. In $Z$, this is written as

$$\forall \text{Astate}; \text{Cstate}; x?: X \cdot \text{pre Aop} \land \text{Abs} \Rightarrow \text{pre Cop},$$

whereas in Fun$Z$ it becomes

$$\forall \text{Astate}; \text{StSpan}; x :: \text{inp } X \cdot \text{pre Aop} \land \text{StSpan}\init \Rightarrow \text{pre OpSpan}.$$  

Note that the expression ‘pre S’, where S is either a schema or span identifier, denotes the preconditions of $S$.

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 obligations are

$$\forall \text{Astate}; \text{Astate}'; \text{Cstate}; \text{Cstate}'; x?: X; y!: Y \cdot$$

$$\text{pre Aop} \land \text{Abs} \land \text{Cop} \land \text{Abs}' \Rightarrow \text{Aop}$$

in $Z$, and

$$\forall \Delta\text{Astate}; \Delta\text{StSpan}; x :: \text{inp } X; y :: \text{out } Y \cdot$$

$$\text{pre Aop} \land \Delta\text{StSpan}\init \land \text{OpSpan} \Rightarrow \text{Aop}$$

in Fun$Z$.

The safety and liveness conditions of Fun$Z$ reflect the fact that a state span contains two additional components when compared to a state schema. In particular, the subterm $\text{StSpan}\init$ 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 relation.
Note that the FunZ proof obligations use the $\Delta$ symbol to denote before and after state
schemas, as well as state spans.

3.5.4 Refinement of Disjunctions

A standard practice when writing a Z specification for a system operation is to specify the
normal circumstances first, to designate the error conditions afterwards, and lastly, to combine
the individual schemas by applying the disjunction operator. Therefore, a typical Z specifica-
tion will contain several definitions of the following form:

$$S \equiv A_1 \lor A_2 \lor \ldots \lor A_n$$

During the design phase, each of these definitions is converted to a FunZ span. As an illustra-
tion, the preceding definition translates as:

```
span S where
S \equiv A_1 \lor A_2 \lor \ldots \lor A_n
end span S
```

Then, during the implementation phase, each FunZ span matching this pattern is refined
(\(\sqsubseteq\) is the refinement operator) to guarded command code by applying the general refinement
rule for disjunctions.

$$A_1 \lor A_2 \lor \cdots \lor A_n \sqsubseteq$$

if pre $A_1 \rightarrow A_1$ $\square$ pre $A_2 \rightarrow A_2$ $\square$ $\cdots$ $\square$ pre $A_n \rightarrow A_n$ fi

Note that [Word92] contains a proof of this refinement rule when $n = 2$, and a straightforward
argument by induction establishes the rule for every positive integer $n$. Moreover, since each
operation span in FunZ is a refinement of a corresponding abstract schema expressed in Z, the
refinement of a disjunction statement to an alternation expression also holds for FunZ specifi-
cations.
3.5.5 Review of the Methodology

There are two primary translation phases when converting a Z specification to a Haskell program via the intermediate specification language FunZ and its associated methodology. The first phase (design) covers the translation of an abstract Z specification to a concrete FunZ specification. The second phase involves the transformation of a FunZ specification to a Haskell implementation.

The first three parts of this section have described fundamental features from the design phase. As a brief recap, an integral part of the design process is the use of a retrieve function and a collection of related theorems to guide the derivation of concrete predicates. After the predicates and corresponding spans are defined, the associated proof obligations (see section 3.5.3) can be fulfilled if the user wants to guarantee the correctness of the design with respect to its initial specification.

During the implementation stage, basic refinement rules are applied in order to translate a FunZ specification to guarded command code. The previous section presented one such rule, namely a prescription for translating a FunZ disjunction to a corresponding alternation construct. Finally, the last step of the FunZ methodology, the transliteration of guarded command code to a Haskell program, is best communicated by virtue of an example. Therefore, this part of the methodology will be explained in section 4.5 when the entire software process (from Z to FunZ to Haskell) is demonstrated with a classical case study.
Chapter 4
Case Studies

4.1 Introduction

It is standard practice in the area of formal methods to use a case study to illustrate the individual steps in applying a particular methodology. Some well-known examples from the literature include the birthday book database [Spiv89], the telephone database [Spiv88], and a computerized class roll [King90b].

To demonstrate how one can translate Z specifications to Haskell code, this chapter presents two case studies, both based on the the Z specification for the class manager's assistant as described in [Word92]. Standard Z is the design language in the first example, whereas FunZ is applied in the second study. Because the intermediate language FunZ and its associated methodology are the primary contributions to the research described herein, more emphasis is given to the second example. However, the first translation is of interest in its own right, because even though Z is an established specification language, it has been used infrequently during the latter stages of software development [King90b]. Furthermore, most of the published instances of transforming Z specifications to functional programming languages describe approaches devised primarily for animation or prototyping, whereas this research assumes that a purely functional language is the final implementation language.

In preparation for the subsequent design specifications, the next section states the original problem, and section 4.3 delineates the abstract Z specification. Meanwhile, the actual case studies appear in sections 4.4 and 4.5. Finally, section 4.6 compares the use of FunZ to that of Z when a purely functional programming language is the target language.
4.2 Requirements

The class manager’s assistant was first published in [Jones, 1980] and has since become a classic example in formal methods. As outlined in [Word92], the specification is comprised of four basic operations: Enrol, Test, Leave, and Enquire. Each of the case studies described in this chapter focuses on a single operation. In particular, the concrete Z specification corresponds to a design for Leave, whereas the FunZ specification depicts a blueprint for an implementation of Test. Informal descriptions of the class manager’s assistant and the respective operations follow.

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.

Leave operation

This operation removes a student from the class with an indication of whether the student is entitled to a completion certificate. Only students who have done the exercises are entitled to a certificate. If the student is not enrolled, a warning is given.

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.

4.3 The Initial Z Specification

As previously mentioned, a Z specification is written using formal Z notation and natural English. The informal statements help to explain the formal notation so that the specification is meaningful both to the customer and future users. Throughout this chapter, statements
preceding each schema describe schema declarations, whereas those afterwards explain its predicates. As is customary in a Z specification, all formal notation appears in italics.

4.3.1 The Abstract State

The specification begins with the declaration of a given set Student. Note that this given set has two functions: it introduces a type, and it postpones representation decisions concerning the type.

\[
\text{[Student]}
\]

The global variable size denotes the maximum size of the class.

\[
| size: N
\]

The enumerated type Response defines appropriate warnings or messages, which are delivered at the conclusion of an operation.

\[
\text{Response ::= success | notenrolled | nocert | cert | alreadytested ...}
\]

Schema Class, defined below, describes the abstract state for the class manager's assistant. The set enrolled represents the class roll, while tested designates the set of students who have completed the exercises.

\[
\begin{align*}
\text{Class} & \quad \text{enrolled, tested: } P\text{ Student} \\
# \text{enrolled} \leq size & \quad \text{tested } \subseteq \text{enrolled}
\end{align*}
\]

The class roll never contains more than size students. Only enrolled students will have done the exercises.

When the class manager's assistant is first activated, no students are enrolled. The schema representing this abstract initial state follows:

\* The schemas in this section are from [Word92].
\( \text{ClassInit} \triangleq [\text{Class} \cdot \text{enrolled} = \emptyset] \)

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

4.3.2 Specification of Leave

Schemas \( \text{Leaveok} \) and \( \text{NotEnrolled} \) specify the Leave operation. Each of these schemas includes a \( Z \) convention that makes its specification more concise. \( \text{Leaveok} \) contains an instance of the Delta convention, namely \( \Delta \text{Class} \), to indicate that the state changes as a result of the operation. Meanwhile, \( \text{NotEnrolled} \) employs the Xi convention with declaration \( \Xi \text{Class} \).

Schema \( \text{Leaveok} \)

When a student leaves the class, the class changes. Student \( s? \) should be furnished as input, and response \( r! \) should be produced as output.

\[
\text{Leaveok} \quad \begin{align*}
\Delta \text{Class} \\
 s? &: \text{Student} \\
 r! &: \text{Response} \\
\end{align*}
\]

\[
\begin{align*}
 s? &\in \text{enrolled} \\
\text{enrolled} = \text{enrolled} \setminus \{s?\} \\
((s? \in \text{tested} \land \text{tested} = \text{tested} \setminus \{s?\} \land r! = \text{cert}) \\
\lor (s? \notin \text{tested} \land \text{tested} = \text{tested} \land r! = \text{nocert}))
\end{align*}
\]

The input student should be a member of the class, i.e. set \( \text{enrolled} \). This student is removed from the class roll. If the student completed the exercises, he or she is also removed from the set \( \text{tested} \), and the output response is \( \text{cert} \). If the student did not do the exercises, then the response is \( \text{nocert} \).
Schema NotEnrolled

This schema records an error condition, namely that the user enters a student who is not a member of the class. In this situation, the class should not change. The input is student $s?$ and the output is response $r!$.

\[
\begin{align*}
\text{NotEnrolled} \\
\exists \text{Class} \\
\ s?: \text{Student} \\
\ r!: \text{Response} \\
\ s? \notin \text{enrolled} \\
\ r! = \text{notenrolled}
\end{align*}
\]

The input student is not enrolled. The response generated is notenrolled.

Schema Leave

A specification for operation Leave consists of the disjunction of schemas Leaveok and NotEnrolled.

\[\text{Leave} \equiv \text{Leaveok} \lor \text{NotEnrolled}\]

Note that this definition demonstrates how Z supports the incremental development of Z specifications through the usage of schema calculus operators.

4.3.3 Specification of Test

Three schemas are used to specify the Test operation: Testok, AlreadyTested, and NotEnrolled. Definitions for Testok and AlreadyTested follow. After reading the previous section, this text should be self-explanatory so only a brief informal explanation precedes each schema. Recall that schema NotEnrolled was defined in the previous section.

Schema Testok

This schema represents the case when the following two conditions hold: 1) a user enters an input student $s?$ who has been tested or, in other words, completed the exercises and 2) 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.

```
Testok ________________________________________
\Delta Class
s?: Student
r! : Response

s? $\in$ enrolled
s? $\notin$ tested
\text{tested}' = \text{tested} \cup \{s?\}
\text{enrolled}' = \text{enrolled}
r! = \text{success}
```

Schema AlreadyTested

This schema designates an error condition: the user enters a student $s?$ who has previously been recorded as completing the exercises. The appropriate output response is `alreadytested`.

```
AlreadyTested ______________________________________
\exists Class
s?: Student
r! : Response

s? $\in$ tested
r! = alreadytested
```

Schema Test

The three previous schemas are combined by using the disjunction operator $\lor$ of the schema calculus. The resulting schema comprises the definition of operation `Test`.

```
Test \triangleq Testok \lor AlreadyTested \lor NotEnrolled
```

4.4 $\mathbb{Z} \rightarrow \mathbb{Z} \rightarrow$ Haskell

Portions of this section originally appeared in "Experiences in Translating Z Designs to Haskell Implementations", by Linda B. Sherrell and Doris L. Carver, copyright (Software
This section traces the development of the Leave operation from its initial abstract specification expressed in Z to a final implementation in Haskell. The example was one of several problems that was translated from Z to Haskell as a part of a comprehensive study to determine the suitability of Z for specifying designs that target purely functional languages.

4.4.1 A Concrete Z Specification

The list has served as the primary data structure for functional languages throughout the history of functional programming (beginning with McCarthy's list-processing language Lisp). Likewise, higher-order functions have continued as a mainstay of the paradigm. Following in this tradition, Haskell's lists and its standard list functions form the basis of the following design.

Concrete state

Schema ConClass, which immediately follows, specifies a concrete state for the class manager's assistant. Two lists, a list of students who have completed the exercises and a list of students who have not, represent the actual class. Sequences testlist and nottested are used to model these lists, because many of the sequence operators from the Z Mathematical Toolkit [Spiv89] correspond to predefined list functions in Haskell.

\[
\begin{align*}
\text{ConClass} & \\
\text{testlist, nottested: seq Student} & \\
\# (\text{testlist} \cup \text{nottested}) \leq \text{size} & \\
\# \text{testlist} = \# \text{ran testlist} & \\
\# \text{nottested} = \# \text{ran nottested} & \\
\text{ran testlist} \cap \text{ran nottested} = \emptyset & \\
\end{align*}
\]
The size (\#) of the sequence resulting from the concatenation (\^) of testlist and nottested cannot exceed the maximum size (size) of the class. The values of testlist and nottested should not contain repetitions since the range (ran) of each sequence represents a group of students. A student cannot be a member of both testlist and nottested.

When Z is employed in data design, a schema is defined to formalize the relationship between the abstract and concrete states. This process is known as forward simulation [Word92]. The next schema, ForSim, describes how to obtain the abstract state Class given the concrete state ConClass.

\[
\begin{align*}
\text{ForSim} & \\
\text{Class} & \\
\text{ConClass} & \\
enrolled = \text{ran} \ (\text{testlist} \ ^\text{nottested}) \\
tested = \text{ran} \text{testlist}
\end{align*}
\]

Since a sequence is a partial function, each essential set of the class manager's assistant is derived from the range of an appropriate sequence. The values of the concatenated sequence (testlist \ ^\text{nottested}) comprise the set of enrolled students. Likewise, the range of sequence testlist corresponds to the set of tested students.

A concrete initial state, which is equivalent to the abstract initial state exhibited in schema ClassInit, follows next. Recall that there should be no students in the class when the class manager's assistant is first activated. To fulfill this requirement, schema ConClassInit sets the values of both tested' and nottested' to the empty sequence (<>).

\[
\begin{align*}
\text{ConClassInit} & \\
\text{ConClass}' & \\
testlist' = <> \\
nottested' = <>
\end{align*}
\]
Concrete Schemas for Leave

This section discusses how to translate the abstract state schemas for the Leave operation into schemas $C_{\text{Leaveok}}$ and $C_{\text{NotEnrolled}}$. For the successful component of the Leave operation, Table 3 illustrates each of its predicates on both the abstract state $\text{Class}$ and the concrete state $\text{ConClass}$. Schema $C_{\text{NotEnrolled}}$ appears below without explanation since its derivation is straightforward.

Table 3. Abstract and Concrete Predicates for Schema Leaveok

<table>
<thead>
<tr>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s? \in \text{enrolled}$</td>
<td>$s? \in \text{ran} (\text{testlist} \land \text{nottested})$</td>
</tr>
<tr>
<td>$\text{enrolled}' = \text{enrolled} \setminus {s?}$</td>
<td>$((\text{testlist}' = \text{testlist} \Rightarrow {s?} \land \text{nottested}' = \text{nottested}) \lor (\text{testlist}' = \text{testlist} \land \text{nottested}' = \text{nottested} \Rightarrow {s?}))$</td>
</tr>
<tr>
<td>$((s? \in \text{tested} \land \text{tested}' = \text{tested} \setminus {s?}) \land \text{r!} = \text{cert}) \lor (s? \not\in \text{tested} \land \text{tested}' = \text{tested} \land \text{r!} = \text{nocert}))$</td>
<td>$((s? \in \text{ran testlist} \land \text{testlist}' = \text{testlist} \Rightarrow {s?}) \land \text{r!} = \text{cert}) \lor (s? \not\in \text{ran testlist} \land \text{testlist}' = \text{testlist} \land \text{r!} = \text{nocert}))$</td>
</tr>
</tbody>
</table>

To obtain predicates on the concrete state $\text{ConClass}$, one can perform a natural translation using schema $\text{ForSim}$ from the previous section. This approach works well for the first and third predicates in the abstract column, but the second predicate translates as:

$$(\text{ran} (\text{testlist} \land \text{nottested}))' = (\text{ran} (\text{testlist} \land \text{nottested})) \setminus \{s?\}$$

The left-hand side of this predicate needs to be in terms of $\text{testlist}'$ and $\text{nottested}'$, and the right-hand side can be expressed more clearly.

Referring to schema $\text{ConClass}$, observe that a student is either a value of sequence $\text{testlist}$ or sequence $\text{nottested}$, but not both. Therefore, the disjunction of two predicates represents a concrete equivalent of predicate $\text{enrolled}' = \text{enrolled} \setminus \{s?\}$. If a student completed the exercises, then he or she must be removed from $\text{testlist}$; otherwise, the student is deleted from
nottested. The Z operator for range subtraction (\(\Rightarrow\)) specifies student removal from the appropriate set.

Schema CLeaveok

In the concrete column of Table 3, the second and third predicates are both a disjunction of conjuncts. Notice that the expression \(testlist' = testlist \Rightarrow \{s?\}\) forms a part of the first disjunct of each predicate and \(testlist' = testlist\) is included in each second disjunct. Due to these common subexpressions and the fact that each student in the class is an element of exactly one of the sequences \(testlist\) and \(nottested\), these two predicates are combined to form a single predicate. The resulting predicate along with the first concrete predicate from Table 3 constitute the predicate part of schema CLeaveok below.

\[
\begin{align*}
\text{CLeaveok} \quad & \quad \Delta \text{ConClass} \\
& \quad s?: \text{Student} \\
& \quad r!: \text{Response} \\
& \quad s? \in \text{ran} (testlist \land \text{nottested}) \\
& \quad ((s? \in \text{ran} testlist \land testlist' = testlist \Rightarrow \{s?\} \\
& \quad \land \text{nottested'} = \text{nottested} \land r! = \text{cert}) \\
& \quad \lor (s? \not\in \text{ran} testlist \land testlist' = testlist \\
& \quad \land \text{nottested'} = \text{nottested} \Rightarrow \{s?\} \land r! = \text{nocert}))
\end{align*}
\]

Schema CNotEnrolled

\[
\begin{align*}
\text{CNotEnrolled} \quad & \quad \exists \text{ConClass} \\
& \quad s?: \text{Student} \\
& \quad r!: \text{Response} \\
& \quad s? \not\in \text{ran} (testlist \land \text{nottested}) \\
& \quad r! = \text{notenrolled}
\end{align*}
\]
Schema CLeave

Obtaining the concrete schema \textit{CLeave} from \textit{Leave} is automatic. Each schema on the abstract state is replaced with its companion schema on \textit{ConClass}, and the disjunction is preserved.

\[ \textit{CLeave} \equiv \textit{CLeaveok} \lor \textit{CNotEnrolled} \]

As a final comment, the Z design schemas in this section look the same as those intended for an imperative language, but the nature of the functional programming paradigm necessitates that some notation has a modified meaning. In particular, because there are no assignment statements and the state must be passed around explicitly with parameters, Z variables have a different meaning.

For example, in the predicate part of schema \textit{ConClassInit}, testlist' is given the empty sequence as a value. In an imperative language, this predicate would correspond to an initialization procedure using an assignment statement. Here it specifies the initial value of parameter testlist when the first operation of the class manager's assistant is executed.

Likewise, a pair of variables such as testlist and testlist' command a modified meaning in the current design schemas. In an imperative design, testlist and testlist' designate the same state location: testlist represents the value stored in the location before an operation is applied, while testlist' represents the value afterwards. In these designs, testlist symbolizes a formal parameter of a Haskell function, whereas testlist' denotes a value returned by this same function. To keep the modified meaning of dashed/undashed variables in perspective, it is helpful to remember that the variables in a functional program behave as mathematical variables. In other words, they maintain their original values throughout the life of the program.

\subsection*{4.4.2 An Implementation for Leave}

This subsection describes the translation of the previous design to a functional program coded in Haskell. First, Table 4 illustrates the pre- and postconditions of schemas \textit{CLeaveok}
and $C_{NotEnrolled}$. The table is designed so that each postcondition depends on the precondition in its row. However, no postcondition appears with the first precondition of $C_{Leaveok}$.

For this portion of the table, each postcondition depends not only on the precondition in its row, but also on the first precondition.

Table 4. Preconditions and Postconditions for Concrete Leave Schemas

<table>
<thead>
<tr>
<th>Schema</th>
<th>Precondition</th>
<th>Postcondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_{Leaveok}$</td>
<td>$s? \in \text{ran} \ (\text{testlist} \land \lnot \text{notested})$</td>
<td>\begin{align*} \text{testlist}^* &amp;= \text{testlist} \uplus {s?} \land r! = \text{cert} \ \lnot \text{notested}^* &amp;= \lnot \text{notested} \end{align*}</td>
</tr>
<tr>
<td></td>
<td>$s? \in \text{ran} \ \text{testlist}$</td>
<td>$\text{testlist}^* = \text{testlist} \land r! = \lnot \text{nocert}$ \land \lnot \text{notested}^* = \lnot \text{notested} \uplus {s?}$</td>
</tr>
<tr>
<td></td>
<td>$s? \in \text{ran} \ \text{testlist}$</td>
<td>$r! = \lnot \text{enrolled}$</td>
</tr>
<tr>
<td>$C_{NotEnrolled}$</td>
<td>$s? \in \text{ran} \ (\text{testlist} \land \lnot \text{notested})$</td>
<td>$r! = \lnot \text{enrolled}$</td>
</tr>
</tbody>
</table>

Observe that the single precondition of $C_{NotEnrolled}$ is the negation of precondition one in $C_{Leaveok}$. This suggests that $C_{Leave}$ can be implemented in Haskell with a conditional expression as follows:

\[
\text{leave } s \ \text{ts} \ \text{ns} = \begin{cases} 
\text{if } s \ '\text{elem}' \ (\text{ts} \uplus \text{ns}) \ \text{then} \ \text{leaveok } s \ \text{ts} \ \text{ns} \\
\text{else } (\text{ts}, \ \text{ns}, \ \lnot \text{enrolled}*)
\end{cases}
\]

Now reexamine the preconditions of schema $C_{Leaveok}$. Precondition one has already been translated into code in function $\text{leave}$. The remaining preconditions are predicate complements that are placed in another conditional expression to form the body of function $\text{leaveok}$.

\[
\text{leaveok } s \ \text{ts} \ \text{ns} = \begin{cases} 
\text{if } s \ '\text{elem}' \ \text{ts} \ \text{then } (\text{ts} \ \backslash \ \{s\}, \ \text{ns}, \ \lnot \text{cert}*) \\
\text{else } (\text{ts}, \ \text{ns} \ \backslash \ \{s\}, \ \lnot \text{nocert}*)
\end{cases}
\]

Note that the functions above do not include type signatures. Since Haskell employs the Hindley-Milner type system, type signatures are not required, and the types are inferred automatically.
Next two supplementary tables appear. Table 5 displays relevant Z predicates from schema *CLeaveok*, as well as the corresponding Haskell code. A discussion of significant Haskell features immediately follows this table. Similarly, Table 6 highlights the associated Z operators and their Haskell counterparts.

**Table 5. Z Predicates and Haskell Expressions**

<table>
<thead>
<tr>
<th>Z</th>
<th>Haskell</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \sigma_? \in \text{ran (testlist} \uparrow \text{nottested)} )</td>
<td>( \sigma \ 'elem' (ts ++ ns) )</td>
</tr>
<tr>
<td>( \sigma_? \in \text{ran testlist} )</td>
<td>( \sigma \ 'elem' ts )</td>
</tr>
<tr>
<td>testlist ( \Rightarrow {\sigma_?} )</td>
<td>ts ( \setminus {s} )</td>
</tr>
<tr>
<td>nottested ( \Rightarrow {\sigma_?} )</td>
<td>ns ( \setminus {s} )</td>
</tr>
</tbody>
</table>

Identifier *elem* is the Haskell function that tests for list membership. Note that backquotes enclose the function name to allow its use as an infix operator.

Recall that ++ is the Haskell operator for list concatenation, while \( \setminus \) stands for list difference. Furthermore, the expression \( xs \ \setminus \ ys \) returns list \( xs \) with the first occurrence of each element of \( ys \) removed (in turn). Therefore, \( ts \ \setminus \{s\} \) has the desired effect of removing input student \( s \) from \( ts \), the group of tested students.

**Table 6. Z and Haskell Operators**

<table>
<thead>
<tr>
<th>Z</th>
<th>Haskell</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \in )</td>
<td>'elem'</td>
</tr>
<tr>
<td>( \uparrow )</td>
<td>++</td>
</tr>
<tr>
<td>( \Rightarrow )</td>
<td>( \setminus )</td>
</tr>
</tbody>
</table>

From Table 6, note that the last mapping does not hold in general. The expression \( X \Rightarrow Y \) maps to \( xs \ \setminus \ ys \) when \( X \) is an injection. In this example, both nottested and tested represent a group of students so this condition is satisfied.

### 4.4.3 Summary

The advantages of Z as a specification language carry over to its use as a design language. These include the precision of its notation and the detection of errors during the
development process. Furthermore, because Haskell is a declarative language, much of the
translation from Z notation to executable code is easier than with an imperative language.

As one example, a natural mapping exists between Z sequences and Haskell lists. Table 6 illustrates the Z operators and corresponding Haskell functions for the previous design, but this table only scratches the surface. Other sequence operations with analogous Haskell functions include the sequence composition operations, reversing a sequence, the filter operation, length of a sequence, and sequence indexing. Since most, if not all, of the Z sequence operators map to predefined Haskell functions, any design using these operators should be relatively straightforward to translate to Haskell code.

4.5 Z → FunZ → Haskell

Building on the specification for the Test operation (see section 4.3), this section illustrates the primary steps in applying the FunZ methodology. Recall that there are two major translation phases: design and implementation. The subsequent description of the design phase (Z to FunZ) includes data refinement, both global and state, followed by an in-depth treatment of operation refinement. Note that the formal correctness proofs appear in a separate subsection (4.5.2). Similarly, the explanation of the implementation phase is divided into two subsections. Section 4.5.3 describes how to use Dijkstra's guarded command code to express an algorithm for the Test operation, and section 4.5.4 demonstrates how a Haskell module can be used to represent an implementation of the entire class manager's assistant.

4.5.1 Data Design with FunZ

The process of constructing a concrete 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 document. For the convenience of the reader, this section reiterates the guidelines corresponding to the design phase of the methodology.
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 function, which maps the concrete FunZ objects to the abstract Z objects, and place this in the relation part of SS. The mapping must be surjective 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.

D. Perform correctness proofs. (This step is optional).
   (1) Prove that each initial concrete state corresponds to an initial abstract state.
   (2) Prove that every operation span satisfies the safety condition.
   (3) Prove that every operation span satisfies the liveness condition.

Translate global variables

The developer has the option of selecting a concrete representation for any basic set that was previously defined on the abstract state. However, in the following development, these
low-level design decisions are postponed so that the translation process merely consists of several syntax changes.

```haskell
basic Student
size :: Int where size ≥ 0
data Response = Success | Alreadytested | Notenrolled ...
```

Note that basic sets become types whose declarations begin with the keyword `basic`. Meanwhile, variable declarations permit constraints. As an example, the declaration of `size` uses a `where` clause to indicate that its value must be nonnegative. Finally, all variables of an enumerated type begin with a capital letter since they correspond to data constructors in Haskell.

**Define state span**

Span Class, which appears in Figure 4, specifies a concrete state for the class manager's assistant. Meanwhile, the subsequent commentary describes the four required components of the state span in order of their appearance.

```haskell
span Class where
ns :: [Student]
ts :: [Student]
inv is
  length (ns ++ ts) ≤ size
  (ns == nub ns) && (ts == nub ts)
  ∀ s :: Student • (s 'elem' ns implies s 'notElem' ts)
  ∀ s :: Student • (s 'elem' ts implies s 'notElem' ns)
rel is
  abmap :: [Student] -> P Student
  enrolled = set (ns ++ ts)
  tested = set ts
init is
  (ns, ts)
  ([], [])
end span Class
```

Figure 4  Formal Text for Span Class
Two lists, a list of students who have never been tested (ns) and a list of students who have (ts), represent the actual class. The declaration part asserts that ns and ts are lists of type Student.

As with a Z document, a FunZ specification consists of both formal text and informal explanations. An informal description for each predicate from the invariant part follows. Since the concatenation (++) of lists ns and ts corresponds to the set of enrolled students, an obvious design constraint is that the length of (ns ++ ts) must not exceed the maximum size of the class. Because each list represents a group of students, ns and ts should not contain repetitions. Note that the predefined Haskell function nub, which removes duplicate elements from a list, specifies this condition. Finally, in tandem, the last two predicates designate that lists ns and ts should be disjoint. This requirement stems from the fact that it is impossible for a student to have never been tested and tested, both at the same time.

Symbols ==, &&, 'elem', and 'notElem' are infix operators that correspond to the =, \&, \in, and \notin 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 constraints in Haskell helps the developer to better understand the chosen data structure. As a final note, the keyword implies is not part of the Haskell language. It has been added to FunZ to handle implication, because \Rightarrow 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 predicates shown above are also legal expressions in Haskell.
As previously mentioned, the relation part contains the retrieve relation, a surjective mapping from concrete to abstract states. In this case, because lists of students model sets of students, a logical choice for the retrieve relation is the \texttt{set} function, previously defined in section 3.5.1.

The initialization part consists of two tuples. The first tuple contains the variables to be initialized, while the second contains the actual values. The notation reflects the fact that the initial values for the concrete objects must be passed as actual parameters when the first operation of a system is executed. For the class manager's assistant, there should be no students in the class when the system is first activated. Therefore, in the span above, the first tuple holds formal parameters \texttt{ns} and \texttt{ts}, while the second specifies that both actual arguments should be the empty list \texttt{[]}.

Derive operation spans

A complete FunZ specification for the Test operation appears at the end of this section. However, to illustrate the actual translation from abstract schema to concrete span, the discussion below focuses on a single component, namely \texttt{Testok}.

As a means of reference, Figure 5 enumerates the predicates from schema \texttt{Testok}, while Figure 6 displays the retrieve function from span \texttt{Class}. Theorems 1 and 3 were previously stated and proved in Section 3.5.1, while the laws from set theory are from [Spiv89].

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{A1.} \quad s? \in enrolled \\
\textbf{A2.} \quad s? \notin tested \\
\textbf{A3.} \quad \texttt{tested'} = \texttt{tested} \cup \{s?\} \\
\textbf{A4.} \quad \texttt{enrolled'} = \texttt{enrolled} \\
\textbf{A5.} \quad r! = \texttt{success} \\
\hline
\end{tabular}
\end{center}

\textbf{Figure 5.} Predicates from Schema \texttt{Testok}
enrolled = set (ns ++ ts)
tested = set ts

Figure 6. Retrieve Function from Span Class

Furthermore, to distinguish the predicates in the subsequent derivation, the following labeling scheme is used. AN designates an original predicate on the abstract state, where N represents the number of the predicate (see Figure 5). Meanwhile, CN denotes a corresponding predicate on the concrete state.

The translation paths from A1 and A2 are similar in that each predicate matches template (1) from section 3.5.2 after a single application of the retrieve function. The resulting predicates are then converted to concrete predicates by employing transformation rule R1 and the appropriate axiom.

A1. \( s? \in \text{enrolled} \Rightarrow \)
    \( s? \in \text{set (ns ++ ts)} \Rightarrow \) Retrieve Function.1
    (*) \( s \ '\text{elem}' \ (ns ++ ts) \) Axiom 1, Rule R1

A2. \( s? \notin \text{tested} \Rightarrow \)
    \( s? \notin \text{set ts} \Rightarrow \) Retrieve Function.2
    C2. \( s \ '\text{notElem'} \ ts \) Axiom 2, Rule R1

By jointly considering the predicate labeled (*) and predicate C2, 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, \( s \ '\text{elem'} \ ns \). Since this new, simpler predicate implies (*), it is selected as C1.

The structure of predicates A3 and A4 suggest translation routes leading first to template (2) and then to Guideline 2. Surprisingly, the predicate that appears to be the simpler of the two expressions requires far more derivation steps in order to match template (2).
A3. \( \text{tested}^\prime = \text{tested} \cup \{s?\} \Rightarrow \) Retrieve Function.2 (twice)

S3. set \( ts' = \text{set ts} \cup \{s?\} \Rightarrow \) Commutative Law for \( \cup \)

set \( ts' = \{s?\} \cup \text{set ts} \Rightarrow \) set.2

C3. \( ts' = ts \) Guideline 2, Rule R1

A4. \( \text{enrolled}^\prime = \text{enrolled} \Rightarrow \) Retrieve Function.1 (twice)

set \( \text{ns}' ++ \text{ts}' = \text{set ns} ++ \text{ts} \Rightarrow \) Theorem 1 (twice)

set \( \text{ns}' \cup \text{set ts}' = \text{set ns} \cup \text{set ts} \Rightarrow \) C3

set \( \text{ns}' \cup \{s\} \cup \text{set ts} = \text{set ns} \cup \text{set ts} \Rightarrow \) set.2

set \( \text{ns}' \cup \{s\} = \text{set ns} \Rightarrow \) \( S \cup T = W \cup T \Rightarrow S = W \)

\( \{s\} \cup \{s\} = \text{set ns} \setminus \{s\} \Rightarrow \) \( S = W \Rightarrow S \setminus T = W \setminus T \)

\( \{s\} \setminus \{s\} = \text{set ns} \setminus \{s\} \Rightarrow \) \( S \setminus V = (S \setminus V) \cup (T \setminus V) \)

\( \text{set ns}' \setminus \{s\} = \text{set ns} \setminus \{s\} \Rightarrow \) \( S \cup \emptyset = S \)

\[ (**\) \text{set ns}' \setminus \{s\} = \text{set ns} \setminus \{s\} \]

The left-hand side of \( (**\) simplifies as follows. First, \( s \text{ elem} \ ts' \) from C3 (see above), which implies \( s \text{ notElem} \ ns' \) due to the invariant on \( \text{span class} \). Therefore, \( \text{set ns}' \setminus \{s\} = \text{set ns}' \). Meanwhile, by a property of the cons function and the definition of \( \text{set} \), the right hand side (RHS) of \( (**\) is equal to \( \text{set ns} \setminus \{s\} \). Hence, \( (**\) is equivalent to the following intermediate predicate:

\[ (***) \text{set ns}' = \text{set ns} \setminus \{s\} \]

A single change to \( (***) \) produces a predicate matching template (2). Since list \( ns \) contains no duplicates, by the invariant in \( \text{class} \), the multiplicity of each of its elements is one. In particular, \( |s|_{ns} \leq |s|_{[s]} = 1 \). Therefore, the RHS of predicate \( (***) \) satisfies the hypothesis of Theorem 3, and \( \text{set ns} \setminus \{s\} = \text{set} \text{ns} \setminus \{s\} \). With this change, \( (***) \) simplifies to \( \text{set ns}' = \text{set} \text{ns} \setminus \{s\} \), which matches template (2). The last step, applying Guideline 2, gives:

C4. \( ns' = ns \setminus \{s\} \).

Finally, by applying Rules R2 and R3, predicate A5 translates as follows:

C5. \( r = \text{Success} \)
This completes the derivation of the predicate part of span Testok. The derived predicates appear together in Figure 7, which immediately follows.

C1. s 'elem' ns  
C2. s 'notElem' ts  
C3. ts' = s:ts  
C4. ns' = ns \ [s]  
C5. r = Success  

Figure 7. Predicates for Span Testok

The next step is to construct the declaration part, which consists of the modifies clause and the variable declarations. To determine which variables should be placed in the modifies tuple, the designer checks all concrete predicates for decorated variables. In this case, the relevant predicates are as follows:

\[ ts' = s:ts \quad \text{and} \quad ns' = ns \setminus \{s\} \]

Therefore, the required modifies clause is:

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

It is worth noting that, in general, the derivation procedure for a predicate may produce an expression of the form \( \text{var}' = \text{var} \). Such a predicate should be omitted from the final design, since the semantics of the modifies clause negates its necessity.

With a few syntax changes, one can easily produce the remainder of the declaration part. The variable declarations of schema Testok are as follows:

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

In FunZ, keywords inp and out respectively replace the ? and ! suffixes. One reason for altering the output notation is because, in Haskell, the symbol ! is the operator for array indexing.
Spans for Test

The FunZ specification for the Test operation follows. The design includes spans Testok, AlreadyTested, and NotEnrolled. Similar to the abstract Z specification, the definition of span Test is the disjunction of the three previous spans.

Span Testok

span Testok where
  modifies Class (ns, ts)
  s :: inp Student
  r :: out Response
  pred is
    s 'elem' ns
    s 'notElem' ts
    ts' = s:ts
    ns' = ns \ {s}
    r = Success
  end span Testok

Span AlreadyTested

span AlreadyTested where
  modifies Class ( )
  s :: inp Student
  r :: out Response
  pred is
    s 'elem' ts
    r = Alreadytested
  end span AlreadyTested

Span NotEnrolled

span NotEnrolled where
  modifies Class ( )
  s :: inp Student
  r :: out Response
  pred is
    s 'notElem' (ns ++ ts)
    r = Notenrolled
  end span NotEnrolled
83

Span Test

span Test where
Test ≤ Testok ∨ AlreadyTested ∨ NotEnrolled
end span Test

4.5.2 Correctness Proofs

Recall that there were three guidelines for proving the correctness of the FunZ specification with respect to its initial Z specification.

- Prove that each initial concrete state corresponds to an initial abstract state.
- Prove that every operation span satisfies the safety condition.
- Prove that every operation span satisfies the liveness condition.

To prove that every initial concrete state of the class manager’s assistant corresponds to an initial abstract state, the required proof obligation is:

\( \text{Class:init'} \Rightarrow \text{ClassInit'} \)

The conclusion of this proof obligation yields the following predicates:

1. \( \# \text{enrolled'} \leq \text{size} \) from \( \text{Class' in ClassInit'} \)
2. \( \text{tested'} \subseteq \text{enrolled'} \) from \( \text{Class' in ClassInit'} \)
3. \( \text{enrolled'} = \emptyset \) from \( \text{ClassInit'} \)

Note that the first two predicates form part of the hypothesis of \( \text{Class:init'} \), so only the third predicate must be shown. The proof of predicate (3) is as follows:

\[
\begin{align*}
enrolled' &= \text{set(ns' ++ ts')} & \text{from rel is} \\
&= \text{set([ ] ++ [ ])} & \text{from init is} \\
&= \text{set [ ]} & \text{definition of ++} \\
&= \{ \} & \text{set.1} \\
&= \emptyset
\end{align*}
\]

This concludes the proof. Therefore, every concrete initial state is consistent or, in other words, every concrete state is well defined.

To prove the safety and liveness conditions for span Test, the following implications must be established:
\textbf{Proof of safety condition}

As previously stated, the proof obligation is as follows:

\begin{equation*}
\text{pre Test} \land \text{Class}\init \Rightarrow \text{pre TestSpan}
\end{equation*}

Expanding the conclusion generates the invariant predicates, (4) - (7), as well as the disjunction of the following predicates:

\begin{align*}
(12) & \quad s \ 'elem' \ ns & \text{from pre TestokSpan} \\
(13) & \quad s \ 'elem' \ ts & \text{from pre AlreadyTestedSpan} \\
(14) & \quad s \ 'notElem' \ (ns ++ ts) & \text{from pre NotEnrolledSpan}
\end{align*}

Note that the predicate $s \ 'notElem' \ ts$ from the precondition of span Testok is not listed. This is because it can be derived from predicates (12) and (6).

The proof is straightforward. First, predicates (4) - (7) are automatically satisfied since they form part of the hypothesis. Second, the disjunction of predicates (12) - (14) is true,
because one of these predicates must hold for each student \( s \). Therefore, span Test satisfies the safety condition because the consequent of the required proof obligation is true.

**Proof of liveness condition**

Since Test is a disjunction of spans Testok, AlreadyTested, and NotEnrolled, the proof can be broken into cases. Note that each of these spans shares predicates (4) - (11), from above, as well as the following predicates from Class'\textendash\textquoteleft init.

\[
\begin{align*}
(15) & \quad \text{length (ns' ++ ts')} \leq \text{size} \quad \text{from inv is} \\
(16) & \quad (\text{ns'} == \text{nub ns'}) \&\& (\text{ts'} = \text{nub ts'}) \quad \text{from inv is} \\
(17) & \quad \text{s 'elem' ns' implies s 'notElem' ts'} \quad \text{from inv is} \\
(18) & \quad s 'elem' ts' implies s 'notElem' ns' \quad \text{from inv is} \\
(19) & \quad \text{tested' = set ts'} \quad \text{from rel is} \\
(20) & \quad \text{enrolled' = set (ns' ++ ts')} \quad \text{from rel is} \\
(21) & \quad \# \text{enrolled'} \leq \text{size} \quad \text{from Class'} \\
(22) & \quad \text{tested' \subseteq enrolled'} \quad \text{from Class' }
\end{align*}
\]

The proof is by cases. Several of the proof steps refer to specific theorems, axioms, and transformation rules. Recall that Theorems 1 and 3 are from section 3.5.1, while the transformation rules and axioms appear in section 3.5.2.

**Case 1.** \( \text{pre Testok} \land \Delta \text{Class'init} \land \text{TestokSpan} \Rightarrow \text{Testok} \)

The hypothesis generates predicates (4) - (11), (15) - (22), and the following predicates:

\[
\begin{align*}
(23) & \quad s? \in \text{enrolled} \quad \text{from pre Testok} \\
(24) & \quad s? \notin \text{tested} \quad \text{from pre Testok} \\
(25) & \quad s '\text{elem' ns} \quad \text{from TestokSpan} \\
(26) & \quad s '\text{notElem' ts} \quad \text{from TestokSpan} \\
(27) & \quad \text{ts'} = s:ts \quad \text{from TestokSpan} \\
(28) & \quad \text{ns'} = \text{ns} \setminus \{s\} \quad \text{from TestokSpan} \\
(29) & \quad r = \text{Success} \quad \text{from TestokSpan}
\end{align*}
\]

The conclusion requires the establishment of the Testok predicates. Since the predicates from \textit{Class} and \textit{Class'} are part of the hypothesis, only the subsequent predicates need be shown.

\[
\begin{align*}
(30) & \quad s? \in \text{enrolled} \\
(31) & \quad s? \notin \text{tested} \\
(32) & \quad \text{tested'} = \text{tested} \cup \{s?\}
\end{align*}
\]
(33) \( \text{enrolled}' = \text{enrolled} \)
(34) \( r' = \text{success} \)

Predicates (30) and (31) are immediate because they form part of the precondition of schema Testok. The proofs for (32) and (33) follow.

Proof of (32):

\[
\text{tested}' = \text{set ts}' \quad \text{by (19)}
\]
\[
= \text{set (s:ts)} \quad \text{by (27)}
\]
\[
= \{s\} \cup \text{set ts} \quad \text{set.2}
\]
\[
= \text{set ts} \cup \{s\} \quad \text{Commutativity of } \cup
\]
\[
= \text{tested} \cup \{s'\} \quad \text{by (8) and Rule R1}
\]

Proof of (33):

\[
\text{enrolled}' = \text{set (ns' ++ ts')} \quad \text{by (20)}
\]
\[
= \text{set ns' \cup set ts'} \quad \text{by Theorem 1}
\]
\[
= \text{set (ns \setminus [s]) \cup set ts'} \quad \text{by (28)}
\]
\[
= \text{set (ns \setminus [s]) \cup set (s:ts)} \quad \text{by (27)}
\]
\[
= \{\text{set ns \setminus [s]}\} \cup \{s\} \cup \text{set ts} \quad \text{set.2}
\]
\[
= \text{set ns \cup set ts}
\]

From predicate (25) of the hypothesis, \( s \ '\text{elem}' \ ns \) or \( s \in \text{set ns} \) (Axiom 1). But, \( s \in \text{set ns} \) is equivalent to \( \{s\} \subseteq \text{set ns} \). Therefore, the law \( S \subseteq T \Rightarrow (T \setminus S) \cup S = T \) is germane in that it justifies the previous proof step. The remainder of the proof is as follows:

\[
\text{set ns} \cup \text{set ts} = \text{set (ns ++ ts)} \quad \text{by Theorem 1}
\]
\[
= \text{enrolled} \quad \text{by (9)}
\]

Finally, predicate (34) is established by applying Rules R2 and R3 to (29). This completes the case for span Testok.

Case 2. \( \preceq \text{AlreadyTested} \land \Delta \text{Class\init} \land \text{AlreadyTestedSpan} \Rightarrow \text{AlreadyTested} \)

The hypothesis consists of predicates (4) - (11), (15) - (22), and the following predicates:

(35) \( s' \in \text{tested} \) from \( \preceq \text{AlreadyTested} \)
(36) \( s \ '\text{elem}' \ ts \) from \( \text{AlreadyTestedSpan} \)
(37) \( r = \text{AlreadyTested} \) from \( \text{AlreadyTestedSpan} \)
Meanwhile, the predicates below from schema AlreadyTested, in addition to those of the invariant on the abstract state, form the conclusion.

(38) \( s? \in \text{tested} \)
(39) \( r! = \text{alreadytested} \)

Note that (38) is from the precondition of AlreadyTested, part of the hypothesis, so only (39) needs to be verified. By applying rules R2 and R3 to predicate (37), one obtains (39). Therefore, span AlreadyTested satisfies the liveness condition.

**Case 3.** \( \text{pre NotEnrolled} \land \Delta \text{Class\_init} \land \text{NotEnrolledSpan} \Rightarrow \text{NotEnrolled} \)

The hypothesis consists of predicates (4) - (11), (15) - (22), and the following predicates:

(40) \( s? \notin \text{enrolled} \) from pre NotEnrolled
(41) \( s \text{ 'notElem' (ns ++ ts)} \) from NotEnrolledSpan
(42) \( r = \text{Notenrolled} \) from NotEnrolledSpan

The predicates below from schema NotEnrolled must be established.

(43) \( s? \notin \text{enrolled} \)
(44) \( r! = \text{notenrolled} \)

Predicate (43) is a part of the hypothesis. To prove (44), rules R2 and R3 are applied to predicate (42). This concludes the proof of the liveness condition for span NotEnrolled, as well as for the Test operation. \( \Box \)

### 4.5.3 An Implementation for Test

As discussed in chapter 3, the implementation phase of the FunZ methodology involves the usage of an adapted form of Dijkstra's guarded command language. Recall the definition of the Test operation: \( \text{Test} = \text{Testok} \lor \text{AlreadyTested} \lor \text{NotEnrolled} \)

The disjunction of spans suggests that the alternative expression is an appropriate starting point for algorithm design. In particular, the following refinement rule is relevant:

\[
A_1 \lor A_2 \lor \ldots \lor A_n \quad \models \\
\text{if pre } A_1 \rightarrow A_1 \quad \Box \text{ pre } A_2 \rightarrow A_2 \quad \Box \ldots \quad \Box \text{ pre } A_n \rightarrow A_n \quad \Box
\]
Table 7 contains the pre- and postconditions for each span of Test. Note that the single precondition \( s \ 'notElem' \ (ns ++ ts) \) from NotEnroiled has been replaced by two equivalent preconditions. Furthermore, \( s \ 'notElem' \ ts \) from Testok does not appear in the table since it can be derived from the invariant of span Class.

**Table 7. 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 \ \setminus \ [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>NotEnroiled</td>
<td>( s \ 'notElem' \ ns ) &amp;( s \ 'notElem' \ ts )</td>
<td>( r = Notenrolled )</td>
</tr>
</tbody>
</table>

After making the appropriate substitutions in the preceding refinement rule, the result is:

\[
\text{Testok} \lor \text{AlreadyTested} \lor \text{NotEnroiled} \\Rightarrow \\
\begin{aligned}
\text{if } s \ 'elem' \ ns &\rightarrow \\
&| \text{Testok} \\
\text{□ } s \ 'elem' \ ts &\rightarrow \\
&| \text{AlreadyTested} \\
\text{□ } s \ 'notElem' \ ns \&\& s \ 'notElem' \ ts &\rightarrow \\
&| \text{NotEnroiled} \\
\end{aligned}
\]

Because \( A \subseteq \text{post } A \) [Morg90], where post \( A \) is the postcondition of \( A \), each span that occurs in an alternative command can be refined by its corresponding set of postconditions. Furthermore, refinement is a transitive relation. Therefore, the following alternative command is a refinement of the preceding if statement.

\[
\text{Testok} \lor \text{AlreadyTested} \lor \text{NotEnroiled} \\Rightarrow \\
\begin{aligned}
\text{if } s \ 'elem' \ ns &\rightarrow \\
&| (ts' = s:ts) \&\& (ns' = ns \ \setminus \ [s]) \&\& r = Success \\
\text{□ } s \ 'elem' \ ts &\rightarrow \\
&| r = Alreadytested \\
\text{□ } s \ 'notElem' \ ns \&\& s \ 'notElem' \ ts &\rightarrow \\
&| r = Notenrolled \\
\end{aligned}
\]
It is easy to see that the preconditions for each component of the Test operation are mutually exclusive by recalling the subsequent predicate from the state invariant.

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

Therefore, the following implementation of the test function employs a sequence of guards.

\[
\text{test} \ (ns, \ ts) \ s \ | \ s \ '\text{elem}' \ ns = ((ns \ \setminus \ [s], \ s:ts), \ \text{Success}) \\
| \ s \ '\text{elem}' \ ts = ((ns, \ ts), \ \text{Alreadytested}) \\
| \ \text{otherwise} = ((ns, \ ts), \ \text{Notenrolled})
\]

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 \( s \ '\text{notElem}' \ ns \) and \( s \ '\text{notElem}' \ ts \), the expression otherwise comprises the last guard. Note that otherwise is simply syntactic sugar for the Boolean value True.

In summary, the test function is a realization of its FunZ specification. First, the arguments match the modifies tuple and student input variable; second, the guards correspond to the preconditions of Test; and finally, the respective function values contain the Test postconditions. More importantly, the technique used to implement the Test operation is a general technique that can be applied to other FunZ designs.

4.5.4 An Implementation for the Class Manager’s Assistant

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 appears at the end of this section. A brief description follows.
Module ClassADT exports the type Classroll and its associated operations (each of which has the same type): enroll, test, leave, and enquire. As is customary for an ADT, its actual representation and the implementation of its operations are hidden from the user. Notice that the module does not contain definitions for the Student and Response types. The example assumes that the module BasicDef, which appears in the import declaration, contains these definitions. As a further note, the type Classroll could be defined as ([Student], [Student]), but the type synonyms Nottested and Tested were introduced for better readability.

```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 above)
leave :: Classroll -> Student -> (Classroll, Response)
-- Definition of leave would appear here
enquire :: Classroll -> Student -> (Classroll, Response)
-- Definition of enquire would appear here

4.6 FunZ Versus Z for Design

By definition, software development with FunZ is similar to that with Z. In particular, the language FunZ retains the structuring facilities of the schema calculus along with certain conventions, albeit modified to correspond better with the functional programming paradigm. Likewise, the methodology encompassing FunZ is an adapted form of the Hursley method [Word92], a software development approach for Z including both data design and algorithm development.
Although FunZ possesses a Z-like flavor, generally the syntax and structure of FunZ more closely resemble Haskell. Therefore, when a software designer uses FunZ to specify a concrete representation for the abstract state and each abstract operation, the resulting specification forms the skeleton of a Haskell program.

To highlight the differences between FunZ and Z specifications, the following tables compare the formal text of the corresponding components in specifications for the class manager’s assistant. In particular, Table 8 exhibits concrete representations for the objects that symbolize the class itself, while Table 9 displays concrete predicates for the successful part of the Test operation. Each table also includes the related Z notation on the abstract state.

Table 8. Objects of the State Space

<table>
<thead>
<tr>
<th>Z (Abstract)</th>
<th>FunZ/Haskell</th>
<th>Z (Concrete)</th>
</tr>
</thead>
<tbody>
<tr>
<td>enrolled, tested: P Student</td>
<td>ts, ns :: [Student]</td>
<td>testlist, nottested: seq Student</td>
</tr>
</tbody>
</table>

Table 9. Predicates for the Successful Component of the Test Operation

<table>
<thead>
<tr>
<th>Z (Abstract)</th>
<th>FunZ/Haskell</th>
<th>Z (Concrete)</th>
</tr>
</thead>
<tbody>
<tr>
<td>s? ∈ enrolled</td>
<td>s 'elem' ns</td>
<td>s? ∈ ran nottested</td>
</tr>
<tr>
<td>s? ∈ tested</td>
<td>s 'notElem' ts</td>
<td>s? ∈ ran testlist</td>
</tr>
<tr>
<td>tested' = {s?} ∪ tested</td>
<td>ts' = s:ts</td>
<td>testlist' = &lt;s?&gt; ∪ testlist</td>
</tr>
<tr>
<td>enrolled' = enrolled</td>
<td>ns' = ns \ [s]</td>
<td>nottested' = nottested \ {s?}</td>
</tr>
<tr>
<td>r! = success</td>
<td>r = Success</td>
<td>r! = success</td>
</tr>
</tbody>
</table>

As these tables demonstrate, when software designers use FunZ as opposed to Z, the distance from design to code is reduced. Stated in another way, the job of the implementor is simplified. An additional benefit due to the earlier derivation of code fragments is that the total cost for a software project should decrease.

A possible drawback to FunZ is that the formal expressions in design specifications are sometimes more complex than their counterparts. As an illustration, Table 10 presents the state invariants for the class manager’s assistant in both Z and FunZ.
Table 10. Invariants on the State Space

<table>
<thead>
<tr>
<th>Z (Abstract)</th>
<th>FunZ</th>
<th>Z (Concrete)</th>
</tr>
</thead>
<tbody>
<tr>
<td># enrolled \leq \text{size}</td>
<td>length (ns ++ ts) \leq \text{size}</td>
<td># (\text{nottested} \land \text{testlist}) \leq \text{size}</td>
</tr>
<tr>
<td>tested \subseteq enrolled</td>
<td>ns == nub ns</td>
<td># \text{nottested} = # \text{ran nottested}</td>
</tr>
<tr>
<td></td>
<td>ts == nub ts</td>
<td># \text{testlist} = # \text{ran testlist}</td>
</tr>
<tr>
<td></td>
<td>(s 'elem' ns) implies (s 'notElem' ts) &amp;&amp;</td>
<td>ran nottested \cap \text{ran testlist} = \emptyset</td>
</tr>
<tr>
<td></td>
<td>(s 'elem' ts) implies (s 'notElem' ns)</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 5

Conclusions

5.1 Summary

In recent years, both formal methods and software reuse have been increasingly advocated as a means of alleviating the ills of the software crisis. During this same time period, purely functional programming languages, which have a long history in the realm of rapid prototyping, have emerged as a viable medium for real-world applications. Since these trends are likely to continue, software developers will need improved methods to translate existing specifications into purely functional implementations.

Therefore, an intermediate specification language, FunZ, has been designed to facilitate the derivation of purely functional programs from Z specifications. FunZ combines features from both Z and Haskell, thus providing a bridge between Z specifications and functional implementations. 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.

Along with the FunZ specification language, an associated methodology has been defined. An essential part of the methodology is a procedure for translating abstract predicates composed in Z to concrete predicates expressed in FunZ. Additionally, for those software developers who wish to prove that a FunZ design correctly implements its initial Z specification, the methodology includes proof obligations for functional refinement.
The overall methodology encompassing FunZ has been demonstrated with a classic example, the class manager's assistant [Word92]. As a means of comparison, as well as motivation for the design of FunZ, a traditional Z design has also been presented.

In addition to the general methodology, a specific retrieve function (set) together with some auxiliary theorems has been characterized. Notwithstanding the important role that these theorems play in the current FunZ case study, their greater value lies in the fact that they constitute a representative sample of the type of theorems possible with FunZ. Furthermore, the set function and supplementary theorems are applicable to an entire class of FunZ specifications—namely, those designs that model sets via Haskell lists.

5.2 Significance of the Research

The incentive for much of the previous work interconnecting formal specifications and functional programming was primarily to simplify rapid prototyping. Animation was often applied in order to validate user requirements and to obtain a working prototype as quickly as possible from an existing specification. Meanwhile, Joosten's method [1989] advocates the use of mathematical notation, as opposed to a dedicated specification language, in order to compose an initial specification and subsequent (more concrete) versions. Recall that this approach supports evolutionary prototyping but, by the author's own admission, only applies formal methods to a minimal degree. A more formal approach to rapid prototyping is the me too methodology [Hend86], but me too is limited to the specification and design stages of the software life cycle.

A complete, formal methodology exists with the wide-spectrum language Extended ML [Sann85]. However, the approach targets Standard ML, which is not a purely functional language, and depends heavily on the sophisticated module system of SML. Johnson and Sander's work [1990] is the most relevant to the work presented in this dissertation since it describes how to translate an abstract Z specification to a functional implementation (written
in Lazy ML). Their method is a hybrid approach combining elements of rapid prototyping with traditional transformation rules. A drawback of the approach is that the initial translation from Z to code is not fully formalized.

As reviewed above, previous methods to target the functional programming paradigm were predominantly influenced by the demands of rapid prototyping. Now that purely functional languages are gaining more popularity as final implementation languages, more extensive software development approaches are required. The intermediate specification language FunZ and its corresponding methodology have been designed to satisfy this requirement when Z is the initial specification language and Haskell is the final programming language.

The following points convey the overall significance of this research:

• The FunZ methodology is currently the most comprehensive, formal approach for translating Z specifications to purely functional programs. The methodology spans the entire software life cycle, from specification through design to final implementation, thus providing a systematic means of recording all decisions throughout the development process. Since the intermediate specification language is particularly designed for functional languages both the design and implementation stages are simplified. In addition, the associated documentation not only supports the maintenance stage of software development, but also the reuse of FunZ design components.

• The FunZ methodology is based on established formal techniques, namely the Hursley method [Word92] and Dijkstra's guarded command language [1975]. It is only methodology to adapt these techniques to the functional programming paradigm.

• The FunZ methodology supports a wide variety of software development styles. In particular, intuitive arguments constitute one end of the spectrum, while formal refinement proofs comprise the other end. Furthermore, FunZ allows software developers to prove
properties about the system design within the realm of Z or Haskell. This means that proofs can be performed throughout software development and the designer is free to select the most appropriate notation.

- FunZ is the first intermediate specification language to target the functional programming paradigm. By combining features from both Z and Haskell, the language provides a natural link between Z specifications and Haskell programs. Moreover, FunZ is a straightforward extension of Haskell. Additions to the language include the span construct and predicate logic connectives that provide for the combination of spans (as in the Z schema calculus).

- The architecture of a FunZ specification is a mechanism that enables a smooth transition from design to code, because much of the syntax is patterned after Haskell. In addition, since Haskell code fragments comprise a part of each concrete specification, those software developers who would like to use testing as opposed to formal proofs are afforded this opportunity earlier in the software life cycle. A related benefit is that the total cost of a software project should decrease since the distance from design to code has been reduced.

- The FunZ methodology is a general software development approach. Furthermore, the intermediate specification language FunZ can be adapted so that its syntax conforms with other purely functional languages.

- FunZ encourages the use of formal methods. Since the FunZ notation is closer to a programming language than Z, FunZ may prove more palatable to those software developers who were previously reluctant to apply formal methods.

In summary, FunZ and its associated methodology advance the field of software engineering by providing a systematic means of translating existing Z specifications to Haskell.
implementations. When compared to traditional animation techniques, the FunZ methodology is much more comprehensive in that it includes formal proofs and is intended for the entire software life cycle.

5.3 Future Research

There are two main approaches for deriving imperative programs from Z specifications: the Hursley method [Word92] and the Refinement Calculus [Morg90]. The current methodology encompassing FunZ emulates the Hursley method except that all design specifications are composed in FunZ and each subsequent step in the development process has functional overtones to accommodate a final implementation in a purely functional language. A future extension of this research is the development of an alternative methodology incorporating FunZ and the refinement calculus. One necessary phase in the project is the modification of the laws of the refinement calculus to match the language constructs of functional programs.

To save time and expense, software developers are often encouraged to reuse existing software components that have already been tested and debugged. As a means of supporting reuse at the specification stage, several formal methods have specification libraries. These libraries typically include specifications for standard data types and associated properties about these types. A valuable addendum to FunZ will be to expand its present base of retrieve functions and auxiliary theorems into a more comprehensive library to aid software developers in translating Z specifications to Haskell implementations. This library will be similar in spirit to the Z Mathematical Toolkit [Spiv89] and the handbooks of the Larch Shared Languages [Gutt90].

Formal methods are much more likely to be employed for practical applications if the methods have computer-aided support tools. Suggested tools for FunZ include a parser/type checker and a proof assistant. In addition, the feasibility of automating the translation from a FunZ design specification to Haskell code should be investigated.
The current version of FunZ targets the conventional features of modern functional pro-
gramming languages. In other words, any idiosyncrasies specific to Haskell have been
excluded. There are two reasons for this design decision. First, the language is made much
simpler by concentrating on the universal characteristics of functional languages and is there-
fore easier for users to learn. But, more importantly, unique features of Haskell, such as its
type classes, are not firmly established, and proposals for improvements to these language
constructs routinely appear on the Haskell mailing list. As the Haskell language becomes
more stabilized, specifications targeting these new features will be examined.

As previously mentioned, functional programming languages are often referred to as exe-
cutable specification languages. Because FunZ is higher-level than Haskell, the practicability
of FunZ as an initial specification language, without regard to Z, is worthy of further investi-
gation. In particular, type classes may prove useful in specifying designs that target object-
oriented languages.
Bibliography


mental applicative language. In The 1980 LISP Conference, Stanford Univer­
sity, University of Santa Clara, The USP Co., pp. 136-143.


Chen, K., P. Hudak, and M. Odersky (1992). Parametric type classes (extended


Dijkstra, E. W. (1975). Guarded commands, nondeterminacy and formal deriva­


directions. In Workshop on Reusability in Programming, Newport, RI. ITT Pro­gramming, Stratford, CN, pp. 2-16.


Appendix A

Haskell Functions

The following function definitions, along with their comments, are adapted from the Standard Prelude as published in the Haskell Report [Huda92b]. Note that the symbol \( \cdot \), which appears in several definitions, is the Haskell infix operator for function composition. Also, \( /= \), which occurs in the definitions of \( \text{null} \) and \( \text{notElem} \), is the symbol for inequality.

-- list concatenation (right-associative)

\[
\text{xs} \cdot \text{ys} = \text{foldr}(\cdot) \text{ys} \text{xs}
\]

-- 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:

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

\[
\text{foldr} :: (a \to b \to b) \to b \to [a] \to b
\]

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

\[
\text{foldr } f \ z \ (x:xs) = f \ x \ (\text{foldr } f \ z \ xs)
\]

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

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

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

\[
\text{del } (x:xs) \ y = \begin{cases} x \equiv y & \text{=} \text{xs} \\ \text{otherwise} & = x : \text{xs} \cdot \text{del} \cdot y \end{cases}
\]

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

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

\[
(\setminus) = \text{foldl del}
\]
-- foldl is the left-to-right dual of foldr.

foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f z []    =  z
foldl f z (x:xs) =  foldl f (f z x) xs

-- 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.

genericLength :: (Num a) => [b] -> a
genericLength =  foldl (\n _ -> n+1) 0
length :: [a] -> Int
length =  genericLength

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

nub :: (Eq a) => [a] -> [a]
nub []    =  []
nub (x:xs) =  x : nub (filter (/= x) xs)

-- filter, applied to a predicate and a list, returns the list of
-- those elements that satisfy the predicate; i.e.;
-- filter p xs == [x | x <- xs, p x].

filter :: (a -> Bool) -> [a] -> [a]
filter p =  foldr (\x xs -> if p x then x:xs else xs) []

-- Boolean function for conjunction

(&&) :: Bool -> Bool -> Bool
True && x  =  x
False && _ =  False

-- Boolean function for disjunction

(||) :: Bool -> Bool -> Bool
True || _ =  True
False || x =  x

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

elem :: (Eq a) => a -> [a] -> Bool
elem =  any . (==)
-- notElem is the negation of predicate elem.

```haskell
notElem :: (Eq a) => a -> [a] -> Bool
notElem = all . (/=)
```

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

```haskell
any :: (a -> Bool) -> [a] -> Bool
any p = or . map p
```

-- Applied to a predicate and a list, all determines if all elements of the list satisfy the predicate.

```haskell
all :: (a -> Bool) -> [a] -> Bool
all p = and . map p
```

-- map f xs applies f to each element of xs; i.e.,
-- map f xs == [f x | x <- xs].

```haskell
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

-- or returns the disjunction of a Boolean list. For the result to be False, the list must be finite; True, however, results from a True value at a finite index of a finite or infinite list.

```haskell
or :: [Bool] -> Bool
or = foldr (||) False
```

-- and is the conjunctive dual of or.

```haskell
and :: [Bool] -> Bool
and = foldr (&&) True
```
Appendix B

Copyright Permission

5454 Foxridge Drive
Baton Rouge, LA 70817
January 31, 1995

John Wiley & Sons Limited
Baffins Lane
Chichester
West Sussex PO19 1UD, UK

To Whom It May Concern:

I am writing to request permission to include material from my article "Experiences in Translating Z Designs to Haskell Implementations" as part of my published dissertation. The article was published in the December 1994 issue of *Software—Practice & Experience*, pages 1159-1178.

I am currently a doctoral candidate in computer science at Louisiana State University and plan to defend my dissertation at the end of March. I need to include a copy of this request and the corresponding permission as an appendix to the dissertation. I would appreciate a prompt reply to this letter as I would prefer to have permission before the final draft of my dissertation goes to the members of my examining committee. At any rate, I need a reply before April 13, as that is the final date for spring semester graduates to turn in their completed dissertations. Thank you for your help.

Sincerely,

[Signature]

Linda B. Sherrell
7th April 1995

Linda B Sherrel
5454 Foxridge Drive
Baton Rouge
Louisiana 70817
USA

Dear Ms Sherrel

This letter represents formal grant of permission for you to include material previously published in the December 1994 issue of Software Practice and Experience, pages 1159-1178, in part of your published dissertation.

Credit should include the following components:

Title, author(s) and/or editors, Copyright ( ) (date and owner).

Reprinted by permission of John Wiley & Sons, Ltd.

Yours sincerely,

Erica Pearce-Howard (Miss)
Editorial Assistant
Vita

Linda Bostwick Sherrell was born in Knoxville, Tennessee, on February 7, 1950. She attended the public schools of Palm Beach County, Florida, and Huntsville, Alabama. After graduation from Huntsville High School in 1968, she entered Auburn University, where she received the B.S. degree with highest honors in mathematics in 1972. The Department of Mathematics at Auburn University then awarded her a graduate teaching assistantship, and she completed the M.S. degree in mathematics in 1974. Before returning to graduate school in computer science, she taught mathematics at the college level for a number of years. Ms. Sherrell is the recipient of both a Louisiana Board of Regents Fellowship and a National Upsilon Pi Epsilon Scholarship for graduate students. Her current research interests are formal methods and functional programming. She and her husband, Daniel Lawson Sherrell, are celebrating their silver wedding anniversary this year. They have two children, Robert Lawson and David Jeffrey.
DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: Linda Bostwick Sherrell

Major Field: Computer Science

Title of Dissertation: A Formal Methodology for Deriving Purely Functional Programs from Z Specifications via the Intermediate Specification Language FunZ

Approved:

[Signatures of Major Professor and Chairman, Dean of the Graduate School]

EXAMINING COMMITTEE:

[Signatures of committee members]

Date of Examination:

March 28, 1995