UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

THE GOZER WORKFLOW SYSTEM

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

By

JASON MADDEN
Norman, Oklahoma
2010

THE GOZER WORKFLOW SYSTEM

A THESIS APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY

Acknowledgements

I wish to thank my friends and colleagues at RiskMetrics Group, including Nicolas Grounds, Matthew Martin, Jay Sachs, and Joshua Zuech, for their valuable additions to the Gozer Workflow System. It wouldn't be this complete without them. Thanks also go to the programmers, testers, deployers and operators of Gozer workflows for their patience in dealing with an evolving system, and their feedback and suggestions for improvements.

Special thanks go to my manager at RiskMetrics, Jeff Muehring. Without his initial support (following a discussion consuming most of the duration of a late-night flight to New York) and ongoing encouragement, the development and deployment of Gozer could never have happened.

Finally, I wish to express my appreciation for my thesis advisor, Dr. John Antonio, for keeping me an the right track and guiding me through the graduate process, and for my committee members, Dr. McGovern and Dr. Page, for their support and for serving on the committee.

Contents

1 Introduction and Background **1**
1.1 Before Gozer 2
1.2 From XML to Lisp 4
1.3 Gozer Design Philosophy 5
1.4 Gozer Development 6
1.5 Related Work 7

2 The Gozer Language **10**
2.1 Syntax 10
2.1.1 The Reader 11
2.1.2 Standard Reader Macros 12
2.2 Evaluation 15
2.2.1 Control Flow 17
2.2.2 Macros 20
2.3 Object Orientation and Data Types 22
2.3.1 Functions 23
2.3.2 Java Interfaces 25
2.3.3 Generic Functions 25
2.3.4 Java Function Façades 29
2.3.5 Dynamic Java Access 30
2.3.6 Common Data Types 34
2.4 Condition System 40
2.4.1 Programmer Interface 41
2.4.2 Implementation 43
2.5 Standard Library 45
2.6 Development Environment and Tools 46

3 The Gozer Virtual Machine and Compiler **50**
3.1 GVM Architecture 50
3.1.1 Image 51
3.2 Function Calling Convention 53
3.3 Bytecode Design 57
3.3.1 Instruction Format 57
3.4 Java Exception Handling 60

3.4.1 Nested Interpreters 61
3.5 The Compiler 63
3.5.1 Compilation Phases 64

4 Workflows **70**
4.1 Overview 71
4.1.1 Tasks and Fibers 72
4.1.2 BlueBox 73
4.2 Distribution 75
4.2.1 Workflow Services 75
4.2.2 Non-Blocking Service Requests 77
4.2.3 Deflink 79
4.2.4 Forking Fibers 80
4.2.5 Error Handling 88
4.2.6 Implementation 90
4.3 Workflow Patterns 93

5 Conclusion **96**
5.1 Contributions 96
5.2 Future Work 97

Appendices

A GVM Bytecodes **104**

List of Listings

2.1 Simplified EBNF-Like Syntax Productions 11
2.2 Basic Gozer Syntax 13
2.3 Standard Reader Macros 14
2.4 The Map Reader Macro 15
2.5 Substitution Rule 15
2.6 Block/Return-From 19
2.7 Catch/Throw 20
2.8 Manual Macro Creation 21
2.9 Template Macro Creation 21
2.10 Closures for CONS 24
2.11 Generic Functions Example 28
2.12 Package-based Java Integration 30
2.13 Clojure-style Java Integration 31
2.14 Type Conversion 33
2.15 Sums-of-Squares Variants 39
2.16 Condition Example 43
2.17 Restart-Case Expansion 45
3.1 Required Argument Assembly 54
3.2 Optional Argument Assembly 55
3.3 Keyword Argument Assembly 56
3.4 Complex Keyword Argument Assembly 56
3.5 Simple Nested Interpreter 61
3.6 Nested Interpreter With Restort 63
3.7 Mapcar Simplification Compiler Macro 67
3.8 Type Inference Examples 68
4.1 Sums-of-Squares Variants 71
4.2 Deflink 81
4.3 Distribution Related Vinz Forms 83
4.4 Vinz Spawn Limit 84
4.5 Using A Task Variable 87
4.6 Task Variable Reader Macro 88
4.7 Vinz Error Handling 89

List of Figures

1.1 Gozer Workflow System 7

2.1 Condition Example Call Tree 44
2.2 Conceptual Type Hierarchy 46
2.3 Gozer Debugger and Inspector 48

3.1 GVM Concepts 52
3.2 Gozer Instruction Format 57

4.1 Process Lifetime 73
4.2 Sample Workflow Lifetime 78

Chapter 1

Introduction and Background

The Gozer Workflow System (GWS) is a production workflow authoring and execution platform that was developed primarily by the author at RiskMetrics Group. It provides a high-level language and supporting libraries for implementing local and distributed parallel processes. The GWS was developed with an emphasis on distributed processing environments in which workflows (complex business processes) may execute for hours or even days. Key features of the GWS include: implicit parallelization that exploits both local and distributed parallel resources; survivability of system faults/shutdowns without losing state; automatic distributed process migration; and implicit resource management and control. The Gozer language is a highly dynamic Lisp dialect designed for the rapid development of complex scripts that can easily exploit a distributed environment as well as local parallelism. Although fundamentally object-oriented, it supports multiple programming paradigms, including functional, imperative, object-oriented and generic. The GWS runs on the Java virtual machine and incorporates ideas from languages such as Common Lisp, Scheme, and Java, among others.

RiskMetrics Group has more than 150 workflows written in Gozer, many of which execute every day. The Gozer Workflow System has had several major releases and is constantly incorporating new ideas with the intent of making the language easier and more productive for workflow developers. The entire GWS implementation currently consists of approximately 15,000 lines of Java code, and about 10,000 lines in Gozer itself. Work is ongoing, particularly in the areas of improved tool support.

1.1 Before Gozer

In late 2006, after determining that no existing commercial workflow systems would meet the needs of RiskMetrics, implementation was begun on a workflow system that was to be tightly integrated with the "BlueBox" distributed computing environment (briefly described in Section 4.1.2). The initial version of this workflow system was heavily influenced by the early BPEL[1] specifications, and expressed workflows as XML documents. At this stage, the workflow system was able to interact with BlueBox services, and express distributed parallel computations and distributed loops of identical computations with different data. Running workflows were able to migrate from machine to machine within the BlueBox environment using basic native Java serialization. It supported the map step of the map-reduce paradigm [8]; the reduce step wasn't directly supported as a loop iteration could produce no values, only side-effects. Service interaction was written into the XML document as an XML template used to produce the service's input.

This XML-based workflow system had no support for variables, only the named values of previous service interactions. Furthermore, there was no support for subroutines or scoping of the named return values. Workflow documents were intended to express a simple coordination of steps, essentially shuffling data from one service to another. An explicit design choice was to require all control-flow decisions to be

[1]The Business Process Execution Languge is a standard designed for specifying interactions with Web Services.

carried out inside a service; thus, the workflow document had no way to express conditionals.

This workflow document, while rigid, met the needs of the first generation of workflow processes, with initial delivery in April of 2007, and production use by August of 2007. One benefit of the rigid structure was that it worked well with off-the-shelf XML schema-driven editors which could assist workflow authors in creating well-formed, valid documents. It also lent itself to a variety of static analyses such as finding incorrect named value references. Running workflow documents was a relatively simply a matter of traversing the document linearly from top to bottom, executing the appropriate well-defined action for each node. With no variables, variable scope, conditionals or subroutines, process migration required only a pointer to the "current" XML document node and the list of previous service results to continue the linear traversal.

Over the next few months, as the workflow system showed benefits in process management and rapid development, it became desired to deliver more and more complicated workflows on the platform. Four major iterations of the workflow system were delivered, each one adding new syntax to the document.

However, as workflow documents became longer and more complex, the limitations of the XML format started to become apparent. Without subroutines, workflow authors often resorted to a "copy-and-paste" style of development, resulting in documents that were difficult to maintain and debug. Because all service interaction was written out as XML templates, there was a lot to copy, and discrepancies could easily creep in. One revision of the workflow system added basic support for the XInclude standard [26] to promote code reuse through placing duplicated XML snippets in separate files and including them multiple times. This effort failed, however, because there was no way to parameterize anything in the included file.

Finally, by November, it was obvious that workflow documents were going to need true conditionals and subroutines. The top-to-bottom, rigid XML format had been stretched to its breaking point. Users now needed to be able to write workflow *programs*, not just workflow *documents*.

1.2 From XML to Lisp

The decision to migrate from XML to Lisp was supported by several factors and observations. A prime observation was that XML's tree structure and Lisp's S-expressions are essentially isomorphic—whatever can be represented in one can be represented in the other by choosing an appropriate transformation. This was an important fact, as, by this time, tens of workflow documents were in production use and would need to be supported, unchanged, by the next version of the workflow system. A single unified execution engine would reduce overall complexity, and, in fact, it was apparent that the existing XML execution engine could be generalized into a primitive tree-walking Lisp interpreter, with its primitive process migration strategy generalized into serializable continuations.

Another factor supporting the migration to Lisp was the realization that Lisp's S-expressions are easy to manipulate and generate programmatically, a feature exploited through Lisp macros. Lisp macros allow creating very high-level syntax to concisely express complicated ideas; creating a custom syntax for describing distributed workflows (much like the custom XML format) would help users write their workflow programs faster. Moreover, if the first version of the custom syntax was found to be insufficient, it would be relatively easy to modify it and extend it, while still maintaining the old syntax (because all these syntax extensions would be developed as high-level macros and not low-level grammar productions).

Also weighing heavily in favor of pursuing a Lisp-based solution was the existence of vast numbers of tools and quantities of documentation that had been developed by the Lisp community over its 50-year life and which would be would be immediately applicable to the new language.

1.3 Gozer Design Philosophy

Thus Gozer was born. Its immediate goal was to be able to express everything the prior workflow system could, and to ultimately grow to be a nearly general-purpose language, tailored for writing workflows on the BlueBox platform. A workflow is mostly a "script" that coordinates efforts among BlueBox services, so this meant that Gozer would need to support features commonly associated with "scripting languages," such as rapid prototyping, interactive development, and quick turnaround times. The BlueBox platform is based on Java, so Gozer would need to run on the Java Virtual Machine (JVM), and it would need to provide access to all the existing Java libraries in use. The Groovy language builds on top of Java and the JVM to provide scripting language-like features, and was already in use in the BlueBox platform, so it was natural for Gozer both to make use of Groovy in its implementation and to expose familiar Groovy features to Gozer programmers. Finally, the workflow system was already in production with new demands on the horizon, so the implementation time had to be short and the implementation effort focused on achieving the most benefit in the limited time available, leading to a pragmatic decision making process in the style of Richard Gabriel's "New Jersey approach" [11].

There are a number of Lisp dialects in broad use today, each with its own semantics. Two dialects that have national or international standards are Scheme and Common Lisp. Both of these dialects were considered as a base for Gozer's semantics. Ultimately, Common Lisp would provide the major influence, though Scheme would contribute several important concepts such as continuations and futures.

Common Lisp's influence runs deep through the design and implementation of the Gozer platform, which will be seen throughout the rest of this thesis (for example, Gozer is a "Lisp$_2$" [12]). However, Gozer is not intended to be a full Common Lisp implementation, simply to be "close enough" to be able to make use of existing Common Lisp documents, tutorials and tools. Existing Common Lisp implementations, both on and off the JVM have served as benchmarks and comparisons, and when the "right" semantics for an operation have not been clear, the semantics used by a Common Lisp implementation, typically SBCL[2], have been followed.

1.4 Gozer Development

The first implementation of Gozer was delivered in December of 2007. By January of 2008, it was mature enough to replace the previous workflow system implementation with the help of an XML-to-S-expression translator developed by Nic Grounds (which, much enhanced, is still in use today). It was that time that the name "Gozer" was introduced for the workflow system as a whole, with various layers having more specific names as the responsibilities became more clearly defined; the developing language would generally be referred to as just Gozer, and the workflow-specific functionality that integrated with the distributed system would be called "Vinz," both in homage to a famous film, and continuing the usage of ghoulish names in BlueBox (other components have been named Casper and Spectre). Figure 1.1 shows how the Gozer and Vinz layers relate to each other, to BlueBox, and to the Java and Groovy platforms.

The Gozer language and implementation continued to evolve rapidly over the next year. By May 2008, some sophisticated off-the-shelf Common Lisp programs could be run in Gozer. The Emacs SLIME IDE was supported in July, and by October of that year, a version of the current bytecode interpreter was in use.

[2]Steel Bank Common Lisp, an open source derivative of Carnegie Mellon's CMU CL, can trace its development back through the 1980s. It is available from `http://www.sbcl.org/`.

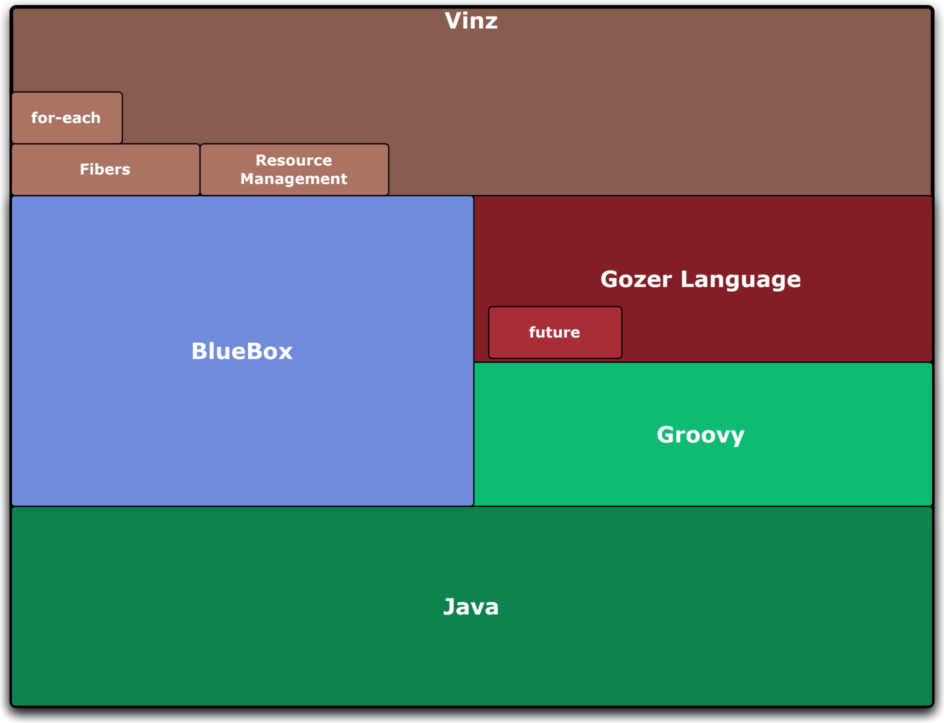


Figure 1.1: Gozer Workflow System

The Vinz workflow layer underwent concurrent modification and evolution. Its process management features became more general and restrictions such as the lack of return values from distributed loops were removed. The efficiency of the distributed operations was always a target for improvements, and support for additional workflow patterns was important too. By September 2010, there had been 20 Vinz workflow feature releases and a dozen Gozer language feature releases put in production.

1.5 Related Work

There are a variety of Lisp derived languages for the JVM, many of which are well known or based on existing national or international standards. These include an implementation of Scheme R4 (JScheme[3]), Scheme R5 (Kawa Scheme[4].), two

[3]Available from `http://jscheme.sourceforge.net/jscheme`.
[4]Available from `http://www.gnu.org/software/kawa`

implementations of Common Lisp (Armed Bear Common Lisp[5] and CLforJava [3]) and a new dialect focused on shared-memory concurrency (Clojure [17]). In contrast to Gozer, which focuses on distributed programming and serializable continuations, these languages all compile to Java bytecode and are executed directly by the JVM, accepting both the benefits (speed and interoperability) and limitations that brings (e.g., Kawa's continuations are "upward-only" and cannot be re-entered).

Likewise, there are a number of dynamic languages for the JVM, including the JRuby implementation of Ruby [9] and the Jython implementation of Python [20], as well as the Groovy language. These too compile to Java bytecode (though some also feature an interpreter), and none of them provide serializable continuations.

Using continuations to simplify the process and robustness of distributed programming has been studied often since the early 2000s in the context of Web programming, where the stateless nature of HTTP is abstracted away using continuations [14]. This can be seen as similar to the approach taken in this thesis: Vinz abstracts away asynchronous service interactions using continuations. Implementations of this idea are also available for languages such as Scheme [23], Smalltalk [31], and Common Lisp[6]. These vary in their design and implementation, depending on the capabilities of the underlying platform. Some use whole-program continuation-passing transformations, some are built on top of the native, first class continuations of the platform, and some require the programmer to delimit the continuations in code. Some produce serializable continuations while some do not; serializable continuations are generally considered to be more scalable than the non-serializable versions [27].

Gozer's intended use for developing and executing distributed workflows rules

[5]Available from `http://common-lisp.net/project/armedbear/`.

[6]The UnCommon Web framework is available from `http://common-lisp.net/project/ucw/`, and WebLocks is available from `http://common-lisp.net/project/cl-weblocks/`.

out straightforward compilation to JVM bytecode due to the need for portable, serializable, re-enterable continuations. Although there are exception-based techniques that can allow for the creation of (at least some degree of) serializable continuations in compiled JVM bytecode, these either require compile-time program transformation [29] or the ability to modify the bytecode as it is loaded (and for best effect, the bytecode of the entire system) [39]. Therefore, for simplicity and the expedient reuse of the existing interpreter code, it was decided to use a non-native call stack, similar to the approach taken by Stackless Python [40] and SISC Scheme [28]. This in turn provides opportunities to extend the execution model for increased dynamism, with the goal of making it faster to write and simpler to evolve distributed workflows.

Chapter 2

The Gozer Language

As a Lisp dialect, the core of the Gozer language is relatively simple and straightforward. This core is extended with features from multiple other Lisp dialects in order to produce a convenient, functional object-oriented Lisp that runs well on the Java Virtual Machine. This chapter begins by describing the textual and conceptual syntax of the Gozer language, including its support for various kinds of macros. Following this is a discussion of the evaluation of Gozer programs and a description of the object-oriented features of the language. The condition handling system (akin to the exceptions of other languages) is described, with particular focus on its implementation in nearly pure Gozer code. Finally, the standard library is briefly reviewed.

2.1 Syntax

The basic unit of Gozer syntax is the S-expression (for symbolic expression, commonly abbreviated to sexp). An sexp may be a single object (an *atom*) or a *list* of other sexps. Gozer syntax is not defined in terms of its textual representation but the S-expression tree that may result from parsing a textual representation or may be programattically constructed.

The ability to programatically construct S-expressions enables the Gozer pro-

Listing 2.1: Simplified EBNF-Like Syntax Productions

```
sign = "+" | "-" ;
exponent-marker = "e" | "E" ;
decimal-digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
all-characters = ? all visible characters ? ;
whitespace = ? white space characters ? ;

integer =  sign , {decimal-digit} ;
exponent =  exponent-marker , integer ;
ratio =  integer , "/", {decimal-digit} ;
float =  integer , ".", {decimal-digit}, [exponent]
       | integer , [".", {decimal-digit}], exponent ;

numeric-token =  integer | ratio | float ;
string = '"' , {all-characters - '"' } , '"' ;

atom = numeric-token | string | {all-characters - whitespace} ;

s-expression = atom | "(", [whitespace], {atom}, [whitespace], ")" ;
```

grammer to extend the textual representation of the language to be more convenient. This can be accomplished with the reader macros discussed in the next section; the ordinary macros of Section 2.2.2 operate at a higher level by modifying the evaluation rule and transforming one S-expression into another.

2.1.1 The Reader

Like Common Lisp, Gozer provides a component with a programmer-accessible interface called the *reader*. It is the job of the reader to take textual data (always encoded in UTF-8) and produce S-expressions. The reader may be extended in arbitrary ways through the *readtable* and *reader macros*; the rest of this section describes the standard syntax understood by an uncustomized reader. Listing 2.2 provides examples.

The basic reader understands only a few simple rules, and it operates by transforming a character stream into a sequence of textual tokens, and thence into the atoms and lists that make up S-expressions. Lists are marked with matching open and close parenthesis. Inside a list, tokens are delimited with whitespace.

Numbers may be integer quantities written in base 10, or rational fractions expressed in base 10, or they may be floating-point quantities written in base 10. (Only floating point, rational fractions, and base 10 decimal integers are a fixed part of the reader. Standard reader macros provide syntax for base 16, base 8 and base 2 integer expressions.)

Strings begin with a double-quote character and proceed until a terminating double-quote is encountered. Within the string, occurrences of the double-quote that are not intended to terminate it must be prefixed by escaping them with a forward slash; this is the only character that requires or allows this treatment. In this way, literal newlines may be embedded to produce a multi-line string, but there is no support for writing escaped newlines or other control characters such as tab. (Individual characters may be expressed with a standard reader macro.)

Any lexical token that cannot be parsed as a string or number is taken to be a symbol. Surrounding a symbol with vertical bars allows a symbol to include spaces or other characters that would ordinarily be interpreted specially by the reader. Within (or at the beginning of) a symbol, the colon character may be used to provide a package qualifier (one colon for exported symbols, two colons for internal symbols)[1]

Comments begin with a semi-colon character and continue to the end of the line. A reader macro provides for multi-line comments.

2.1.2 Standard Reader Macros

The standard reader includes a set of reader macros to make writing certain common productions easier. A reader macro is a function that defines special syntax for the reader, consuming characters and producing S-expressions. Earlier, the reader

[1]Packages, described in Section 2.3.6 are a namespace feature and are represented as objects accessible to the Gozer programmer. This demonstrates the deep integration of all levels of the language.

Listing 2.2: Basic Gozer Syntax

```
"This is a string"
;; This is a comment
symbol                     ;; symbol in current package
package:external-symbol   ;; symbol exported from package
package::internal-symbol;; symbol internal to package
:keyword-symbol            ;; symbol in keyword package
|symbol with spaces|

1234      ;; base-ten integer
1.23      ;; base-ten float
1.23e4    ;; base-ten float: 12300.0
-1.23e-4 ;; base-ten float: -0.000123
3/5       ;; exact rational

(a list (nested list)
   "containing a multi-line string
    and a number:"
   1234)
```

macros for numbers were introduced, and, in fact, the double-quote and parenthesis can be considered reader macros as well. This section describes several other reader macros, with examples in Listing 2.3.

There are two types of reader macros, single character reader macros and dispatching reader macros. The double-quote is an example of the former, and the octal, hex, and binary number reader macros are examples of the latter. Single character reader macros are triggered by the occurrence of the character itself, while dispatching reader macros are triggered by a prefix character which then reads the next character and dispatches to a function associated with that character.

Other single character reader macros include the single quote, back quote and comma. The single quote placed in front of a symbol quotes that symbol (see Section 2.2 on quoting). The back quote and comma are forms of quoting and unquoting entire S-expressions for producing templates in macros. The Vinz module provides access to single character reader macros for task variables (see Section 4.2.4).

Dispatching reader macros are defined to read literal character values, literal array values, literal URL and URI values (corresponding to Java's URL and URI

Listing 2.3: Standard Reader Macros

```
;; Numeric bases
#o11 ;; octal integer 9
#x11 ;; hex integer 17
#b11 ;; binary integer 3

;; Conditional reading based an features
#+feature "Only read this if feature is present"
#-feature "Only read this if feature is absent"

#|
Multiline comments
with vertical bars
|#

;; Unique, uninterned symbols
#:uninterned-symbol

;; Quoting
'quoted-symbol
'(a quoted list)
`(a backqouted ,template ,@list)

;; Unreadable objects
#<triggers an error>

;; Characters, by name or self-describing
#\Space
#\Newline
#\a

;; Vectors (arrays)
#(1 2 3)
#(a vector of symbols)

;; A URL and a URI. Their syntax is validated by the reader macro function
#Uhttp://example.org/file.txt
#uurn:example.org:identifier

;; Readable structures
#S(struct-name :slot "value" :slot2 1234)

;; Reference to a Java class, validated by the reader macro function
#Ljava.lang.Object

;; Java method call, property access
#{object.property.method()}

;; A Map
#["key" => "value" "key2" => 1234]
```

Listing 2.4: The Map Reader Macro

```
(defun %read-hash-table (s c n)
"Read maps like 'key => value'. "
  (let ((data (read-delimited-list #\] s))
        (setfs (list))
        (i 0))
    (while (< i (length data))
      (assert (eq '=> (elt data (+ i 1))))
      (let ((key (elt data i)))
       (push `(setf (gethash ,key table) ,(elt data (+ i 2)))
             setfs))
      (incf i 3))
    `(let ((table (make-hash-table)))
       ,@setfs
       table)))

(set-dispatch-macro-character #\# #\[ #'%read-hash-table)
```

Listing 2.5: Substitution Rule

```
(+ 1 2 3 (* 2 2) (if (= 3 (+ 1 2)) 5 -5))
(+ 1 2 3 4       (if (= 3 (+ 1 2)) 5 -5))
(+ 1 2 3 4       (if (= 3 3)       5 -5))
(+ 1 2 3 4       (if T             5 -5))
(+ 1 2 3 4                         5)
15
```

classes), literal hashtables (Listing 2.4), references to Java classes and user-defined structures. In principle, almost any character could be used as a prefix character for dispatching macros; in practice, only the octothorp, #, is used in this way.

2.2 Evaluation

The basic evaluation rule for Gozer programs can be stated simply: left-to-right, inside-out. In general, Gozer evaluation follows the “substitution rule,” where every S-expression form is replaced with its value in left-to-right, inside-out order. Every form has a value. That is, there is no distinction between “expressions” that have a value and “statements” that do not as in languages such as Java or C. (See Listing 2.5.)

Unlike Common Lisp and now Scheme [10], the Gozer language does not distinguish between compile time, load time, and evaluation time. Loading a file always results in its compilation, and as each top level expression is encountered and compiled, it is evaluated. This makes it easy to implement interactive development, as the effect of loading from files is the same as typing a REPL (Read-Eval-Print-Loop). Gozer does follow Common Lisp in distinguishing between *extent* (duration) and *scope* (visibility), and *lexical* (textual) and *dynamic* (runtime) environments.

An individual symbol is treated as a variable and evaluates to its most recently assigned value. Variables may be either lexical (definite extent and lexical scope) or dynamic (dynamic extent and indefinite scope); all variables are assumed to be dynamic variables unless otherwise declared lexical. Dynamic variables (also referred to as global or special variables) are interesting by acting as if they have a stack of values. Only the top of the stack can be read or assigned. Exiting a scope in which a dynamic variable was bound restores its value to what it had on entering that scope, even if the variable was assigned within the scope. A literal such as a string or number is its own value.

Lists such as `(+ 2 3)` denote function application. The first element of such a list is typically a symbol, and the function named by that symbol is applied to the results of evaluating the other elements in the list. Functions are described in more detail in Section 2.3.1.

The evaluation rule can be modified in two ways. The first way is through *special forms.* A special form is written like a function application, but it is interpreted differently. Each special form defines its own evaluation rule. An example of a special form is the `if` form: `(if predicate then else)`. The *then* form is evaluated only in the case that the *predicate* form's value is true; otherwise the *else* form (if present) is evaluated. This is an obvious exception to the substitution rule and requires special support.

Gozer defines a number of special forms that implement various control structures (Section 2.2.1) (block, catch, go, if, return-from, tagbody, throw, unwind-protect, while) modify symbol bindings (setq), establish lexical environments (flet, labels, let, macrolet, symbol-macrolet), create and invoke functions (apply, funcall, lambda), interact with the compiler and runtime (declaim, declare, the), and support parallel (Section 2.3.6) and distributed (Section 4.2) computing (future, yield). The special forms are fixed and cannot be added or changed by Gozer programmers, but macros (Section 2.2.2) allow programmers to define their own evaluation rules.

The second way the evaluation rule can be modified is to bypass it entirely through *quoting*. Any form can be quoted by placing it inside the special form quote. Quoting a symbol results in the symbol itself, not the value bound to it, and quoting a function application results in the S-expression that defines the function application (a list of S-expressions). Although this is just an example of a special form, it is a heavily used example, so heavily used that the standard reader allows abbreviating it to a single leading ' character. The concept of quoting allows Lisp code to be treated as data; any expression that could be evaluated could also be quoted and dealt with as an S-expression. This, in turn, allows for very high-level and symbolic programming through the use of macros or other types of S-expression walking programs; since these are generally written in Lisp itself, quoting may be an example of self-reference.

2.2.1 Control Flow

Common Lisp and Gozer offer very powerful control flow capabilities to the programmer. These capabilities allow the programmer to develop new abstractions that would otherwise have to be provided directly by the language implementation. For example, Java's `try`/`catch`/`finally` language-level keywords can be implemented using Gozer's control flow operators (see Section 2.4).

Operator	Local	Lexical
if	Local	Lexical
while	Local	Lexical
unwind-protect	Local	Lexical
block	Non-Local	Lexical
return-from	Non-Local	Lexical
tagbody	Non-Local	Lexical
go	Non-Local	Lexical
catch	Non-Local	Dynamic
throw	Non-Local	Dynamic

Table 2.1: Control Flow Operator Classification

Gozer's control flow operators can be classified in two ways. One way is to consider whether their change is purely local or may have a non-local impact. The second way is to notice if their actions can be determined based on the lexical environment, or if the dynamic environment must be considered. Table 2.1 shows this classification.

The simplest way to modify the control flow of a program is with the conditional if, which, as in other languages, evaluates a predicate and then directs evaluation to one of two branches. The while operator provides the most basic looping support, repeating its body so long as its predicate evaluates to true.

The block/return-from pair allow evaluation of a named lexically-enclosing block to be prematurely aborted. Instead of resulting in the last value lexically evaluated in the block, the value given to return-from is the result. These operators are used to implement early return from functions as well as loops. Although at first they may seem to be local operators, in fact they are non-local operators because a closure that's created when a block is in scope may be used to later return from that same block. Listing 2.6 shows an example of this. The first invocation of recursive-block creates a block named B and passes a closure to a second invocation of itself. This

Listing 2.6: Block/Return-From

```
(defun recursive-block (fun)
  (block B
    (if fun
        (funcall fun)
        (recursive-block (lambda () (return-from B :blk))))
    :fun))
(recursive-block nil) ⇒ :blk
```

invocation also creates a block of that same name, and then calls the closure passed to it. The return-from in this closure returns from the block established by the *first* invocation (it is said to perform a *non-local exit* from the second invocation).

Somewhat similar to goto in C-derived languages, the tagbody/go pair in Gozer provide a way to transfer control to arbitrary named locations. The primary difference is that, like block/return-from, these transfers may be non-local when closures are involved. In Common Lisp, the standard iteration construct is the do macro, which is implemented in terms of the tagbody and go special forms. Early Gozer implementations lacked these two very general special forms, so the while special form was provided for iteration. Later, tagbody and go were added, allowing the addition of an implementation of Common Lisp's flexible English-like looping facility and the condition system (Section 2.4); while is retained for execution efficiency.

The last pair of operators is catch and throw. This pair is the dynamic counterpart of the block/return-from pair. Whereas a return-from lexically (statically) determines its target, the target of a throw is determined based on the dynamic call stack. Contrast the results of Listing 2.7 with those of Listing 2.6 to see the difference in dynamic versus lexical targeting; the throw in the invoked closure in Listing 2.7 is caught by the dynamically enclosing catch in the recursive execution of recursive-catch, and the first invocation of that function exits normally with the value :**fun**.

Finally, the unwind-protect form evaluates its first form and ensures that the

Listing 2.7: Catch/Throw

```
(defun recursive-catch (fun)
  (catch 'B
    (if fun
        (funcall fun)
        (recursive-catch (lambda () (throw 'B :catch))))
    :fun))
(recursive-catch nil) ⇒ :fun
```

remainder of the forms are evaluated even if the first form attempted a non-local exit.

2.2.2 Macros

Macros allow the Gozer programmer to extend the syntax of the Gozer language by transforming one (implicitly quoted) S-expression into another. Although Gozer, like Common Lisp, has four types of macros—reader macros, compiler macros, symbol macros, and standard macros—when the word “macro” is used alone it always refers to standard macros. Reader macros operate on text; standard macros operate on S-expressions. All types of macros are function objects (as in Section 2.3.1) that have particular signatures.

A macro application is written in the same way as a function application, as a list whose first element is a symbol, but the similarities end there. Instead of being evaluated at runtime and returning an arbitrary value, a macro application is evaluated at compile-time and must return an S-expression which is then evaluated instead of the original S-expression (possibly another macro application). None of the arguments to a macro are evaluated as they are to a function; instead, the macro receives the actual S-expressions. While a function provides functional abstraction, a macro provides syntactical abstraction. Often this abstraction takes the form of producing boilerplate code for common actions such as iteration or resource acquisition and release.

Although macros can be written to operate directly on the S-expression parame-

Listing 2.8: Manual Macro Creation

```
(defmacro my-dolist (&body body)
  (let ((result (list 'dolist (list (first (first body))
                                    (second (first body))))))
    (dolist (exp (cdr body))
      (append! result exp))
    result))
```

Listing 2.9: Template Macro Creation

```
(defmacro my-dolist ((var items) &body body)
  `(dolist ((,var ,items)) ,@body))
```

ter and to produce sexps by manually constructing lists in the appropriate patterns, as in Listing 2.8, this is very tedious and error prone. (However it does offer supreme flexibility and demonstrates the correspondence between Lisp code and Lisp data.)

Gozer addresses both of these problems by following Common Lisp. First, Gozer provides automatic *destructuring* of the argument passed to the macro. For macros that take regularly structured arguments, this allows the programmer to pass the tedium of walking through an S-expression off to Gozer. On the output side, Gozer provides the backquote reader-macro, `, for template-based creation of S-expressions. Using these features allows Listing 2.8 to be simplified to Listing 2.9.

Gozer's macros, like Common Lisp's and unlike Scheme's, are not hygienic. This means that it's possible for a macro to inadvertently refer to an identifier bound in its surrounding scope, hijacking it and producing incorrect results [22]. Gozer's package system and separate namespace for functions and variables (Gozer is a Lisp$_2$ [12]) reduce the chance for unintended name collisions, and there are some simple techniques (e.g., the use of gensym and facilities built on top of it) that can reduce the chances even further. Though not perfect, this has proven to be a reasonable balance between implementation effort and safety.

2.3 Object Orientation and Data Types

This section describes the use of objects in the Gozer language including information on achieving polymorphism, "duck-typing," the integration of the Java and Gozer object models, and how the Groovy runtime is leveraged for dynamic capabilities. It will also describe in more detail some of the data types used in Gozer and their behaviours. Centrally, every value in Gozer is an object, from the lists and atoms that make up S-expressions, to functions, to symbols, to conditions, to numbers, to classes, to user-defined types. In order to work well on the JVM, these objects are JVM-level objects; the behaviours that the JVM does not directly support are introduced with the Gozer (and Groovy) runtimes. Gozer does not provide syntax for creating and manipulating Java's primitive types (`boolean`, `char`, `short`, `int`, `long`, `float`, `double`); instead, all such values are always manipulated in their boxed form. Gozer auto-unboxes values passed to Java methods and auto-boxes results received from Java methods. This is consistent with most modern scripting languages and user expectations, and simplifies implementation.

What behaviours do JVM objects support? The object model of the JVM, which directly reflects the object model of the Java programming language, is class-oriented featuring single inheritance of implementation, but multiple inheritance of interface. Where inheritance or interfaces are involved, the selection of which code to invoke only considers the dynamic (runtime) type of the receiver (plus the static (compile-time) type of the arguments); Java is a single-dispatch language. There is a single root class called Object from which all other classes descend. Classes themselves are manifested as Object instances (of class Class), though they are not true first-class objects (i.e., it is not directly possible to create or manipulate new Class instances from within the Java language, and Class instances have no methods or behaviour of their own). Once defined, a class is immutable. Both the Java language and the

JVM permit *primitive* types that are not objects and not instances of a class.

Gozer maintains the single-inheritance, multiple-interface structure of JVM objects. From Groovy it gets the notion of first-class classes that can be modified at runtime (through Groovy's metaclass system [21], to which Gozer provides access). Ad-hoc polymorphism and multiple dispatch are also added; protocols for objects to cooperate with the system-provided generic functions (Section 2.3.3) and type conversion (Section 2.3.5) are developed from this basis.

As a functional language, the primary mechanism for interfacing with objects in Gozer is, naturally, functions. Following a general description of functions, the next section describes CLOS-derived generic functions and the sections that follow introduce functional façades over Java objects. After that, some key Gozer data types are discussed.

2.3.1 Functions

In Gozer, functions are of course first-class objects. They may be passed as arguments and invoked at runtime, and they may be either anonymous or bound to a symbol that provides their name (lexically or dynamically). Functions are always created with the lambda special form; the defun macro expands into code to assign a name with dynamic extent to a lambda expression. Section 2.3.4 shows how Java static and instance methods can be treated as functions while Section 2.3.3 shows how functions can be used for object-oriented programming. Functions created in Gozer directly implement the Java Function interface and may be passed as objects to, and invoked by, Java classes; this will call back into the GVM to interpret the Gozer bytecode of the function. Functions may also indirectly be converted to implement other Java interfaces containing one or more methods, as described in the next section.

If a function is created inside a lexical environment, it may *close over* existing

Listing 2.10: Closures for CONS

```
(defun cons (x y)
  (lambda (d &optional n)
    (case d
          (car    x)
          (cdr    y)
          (rplaca (setq x n)))))
(defun car (c)
  (funcall c 'car))
(defun cdr (c)
  (funcall c 'cdr))
(defun rplaca (c o)
  (funcall c 'rplaca o))

(setq c (cons 1 2)) ⇒ #<FUNCTION LambdaByteWrappedFunction(689292896)>#
(car c)             ⇒ 1
(cdr c)             ⇒ 2
(rplaca c 3)        ⇒ 3
(car c)             ⇒ 3
```

lexical variables. Abelson's classic example [1] of creating a traditional Lisp CONS cell using a closure can be replicated in Gozer (see Listing 2.10). Such closures could also be used in the implementation of Java interfaces (Listing 2.14 demonstrates a closure being used to implement a Java UnaryPredicate interface).

Functions fully support the complex lambda lists (arguments) of Common Lisp, including required positional arguments, optional positional arguments, keyword (named) arguments, and rest (variardic) arguments. Although support for complex lambda lists complicates the implementation, it supports graceful evolution of functions and can result in function calls that are self-documenting.

When invoked, functions always return a value. Sometimes it is desirable for a function to return multiple values. This can be done explicitly by returning a composite object like a structure (Section 2.3.6) or a sequence, or it can be done implicitly through the use of the values special form. A function returning values is said to return a primary value and any number of secondary values. Taken from Common Lisp, this notion of multiple return values is another way to support the graceful evolution of functions.

2.3.2 Java Interfaces

Gozer is not compiled to Java bytecode, so it is not possible to subclass existing Java classes from Gozer code. However, Java allows for the implementation of interfaces without generating Java bytecode through dynamic proxies. Gozer uses this mechanism to allow Gozer code to implement arbitrary interfaces. In this way Java code can invoke arbitrary Gozer code as a callback.

To implement an interface of one method (common in Java), Gozer programmers may simply use any function (often an anonymous lambda) that accepts the requisite number of arguments. More complicated scenarios, such as implementing multiple interfaces or multiple methods from a single interface, require the use of the new-proxy macro. As an example of where this is used, the bulk of the Gozer reader is implemented in Java. The reader defines a Java interface for reader macros. Gozer code that wants to implement a reader macro (as in Listing 2.4) is converted into an implementation of this interface through a call to new-proxy made in set-macro-dispatch-character.

2.3.3 Generic Functions

A *generic function* can be thought of as an abstract operation that defines a set of formal parameters and a set of outputs, but provides no implementation. *Methods* are added to a generic function to provide implementations of the operation for particular input data types or even particular instances; this is referred to as *specializing* the function. At runtime, when the generic function is invoked, it will choose and execute the most specific method based on the actual types of all arguments.

A common example is the system-provided arithmetic functions such as +. These are generic functions, choosing different implementations (e.g., integer vs floating point) based on their arguments. It is an obvious step to generalize from system-provided data types and functions into user-provided data types and functions. If types may be arranged in an inheritance graph, then in this way code may be reused (a sub-type may provide specialized methods for some generic functions, but leave the rest of the methods as implemented by the parent type; the specialized methods may themselves invoke less-specialized methods to perform part of the operation).

Common Lisp provides that generalization in a system called the Common Lisp Object System (or CLOS). CLOS is derived from experience with prior Lisp object systems, notably Xerox's Loops and MIT/Symbolic's Flavors. In contrast to message passing object systems such as Smalltalk and C++ where the class is the primary mechanism of object and code organization and the message send is the primary way of object interaction, in CLOS the (generic) function provides these capabilities. Because CLOS is function-based, Lisp style can be maintained and even code that wasn't written with objects in mind but with functions (through funcall and apply) can transparently interact with objects [13].

Gozer provides a subset of the CLOS generic function capabilities. All the ways of specializing a method are supported. Notably missing are support for multiple method combinations and programmer-defined method combinations, though these could be added if desired[2]. CLOS generic functions can also be used to implement what today would be called aspect-oriented programming by providing methods that run before, after, or around other methods, and Gozer implements this as well[3].

[2]A method combination determines what happens when there are multiple methods that could be applicable to a set of arguments. The standard method combination that Gozer provides invokes the most specific method and allows it to invoke less specific methods if desired. CLOS provides other combinations, but these are rarely used.

[3]As with the flexibility in function arguments and return values, it is expected that the flexibility provided by aspect-oriented programming will be of great help in evolving and extending Gozer-based workflows.

Types and Structures

Instead of defining its own type system for the purposes of generic functions, Gozer simply reuses the Java type system and hierarchy. Thus, a method can be specialized for any Java class or interface and is applicable for instances of that class or any of its subclasses, or any class that directly or indirectly implements the interface (see Listing 2.11).

Attaching methods to existing Java classes and interfaces satisfies many use cases for generic functions, but there are times when it may be desirable and convenient to group data at the Gozer level without statically creating a Java class. To allow for this, Gozer fully supports the notion of Common Lisp user-defined structures (or record types, see Section 2.3.6). Structures may be instances of a Structure class (by default), or may be based on lists or arrays (for speed or for evolution of older code where lists were used for prototyping). When the Structure class is used, methods can be specialized on them (see Listing 2.11 for an example). Structures support a limited form of single inheritance, which matches the single-inheritance model of Java classes, making it easy to evolve an implementation from a Gozer structure into a Java class or vice versa.

Performance

CLOS was designed to allow for efficient implementations [13], and some implementations were very highly optimized [7]. Despite being a highly dynamic system, some CLOS implementations are able to meet or exceed the performance of a static language such as C++ [42].

Gozer currently has none of these optimizations. Each invocation of a generic method results in a fresh calculation of the applicable method set and the generation of objects to represent the effective method combination. Invoking generic functions

Listing 2.11: Generic Functions Example

```
;; Create a generic function
(defgeneric foo (x y))
;; Add a default method
(defmethod foo (x y) 'bare)

;; Specialize on a number
(defmethod foo ((x number) y) 'number)

;; Specialize on a string
(defmethod foo ((x string) y) 'string)

;; Specialize on a string and a symbol
(defmethod foo ((x string) (y symbol)) 'string-symbol)

;; Specialize on a particular instance
(defconstant object (new 'java.lang.Object))
(defmethod foo ((x (eql object)) y) 'object)

;; Specialize on a keyword, number, or other constant
(defmethod foo ((x (eql :kw)) y) 'kw)

(assert
  ;; The default method
  (eq 'bare (foo 'blah 'blah)) )

(assert
  (eq 'number (foo 1 2)) )
(assert
  (eq 'string (foo "bar" 123)) )
(assert
  (eq 'string-symbol (foo "bor" 'sym)))
(assert
  (eq 'object (foo object t)) )
(assert
  (eq 'kw (foo :kw t)))


;; Specialize on a Java interface
(defmethod foo ((x java.util.List) y) 'a-list)
(defmethod foo ((x java.util.Collection) y) 'a-collection)
(assert
  (eq 'a-list (foo (list) nil)))
(assert
 (eq 'a-collection (foo (new 'java.util.HashSet) 'hash)))
```

is thus several times slower than invoking ordinary functions. In Gozer's niche this has not yet posed a problem, but with wider adoption of generic functions it is likely that some performance optimization will be required.

2.3.4 Java Function Façades

Gozer is a scripting language hosted on a platform defining its own language and providing access to a large number of existing libraries. Programmers want to be able to use those libraries. Unfortunately, those libraries are structured in terms of classes, interfaces and methods where Gozer is structured in terms of functions. To ease the impedance mismatch, Gozer recasts Java classes and methods as functions. This approach was previously taken by Charleston College in their Common Lisp implementation for the JVM [3].

This begins by taking advantage of one structural element that Java and Gozer do have in common: packages. A package in both languages is an organizational technique for system construction (programming in the large). Packages act as namespaces to prevent the symbol clashes that would otherwise inevitably occur. For example, the Java platform ships with two classes named Date, one in the java.util package and another in the java.sql package. Although there are differences—a Gozer package is a first-class object intimately tied to the Gozer reader, runtime system and compiler, a Java package is just a namespace—this common ground is enough to build upon.

The Gozer runtime is made aware of a Java package using the defjavapackage macro (analogous to the defpackage macro for standard Gozer packages) or the import-from-java macro. When the compiler encounters a symbol from a Java package, rather than looking for its definition in the Gozer runtime, the JVM is consulted instead (actually, Groovy's dynamic runtime information). If the symbol resolves to a Java class, a constructor function is returned; a method returns a function that

Listing 2.12: Package-based Java Integration

```
;; Make the runtime aware of the package and a specific class
(import-from-java "java.lang" "String")

;; Calling instance, static, and constructor methods
(String.charAt "string" 1)        ⇒ \s
(String.valueOf 1)                ⇒ "1"
(String "copy")                   ⇒ "copy"

;; Using in MAPCAR, etc
(mapcar #'String.toUpperCase
        '("string1" "string2"))   ⇒ ("STRING1" "STRING2")
(mapcar #'String
        '("copyme" "copymetoo"))  ⇒ ("copyme" "copymetoo")
```

will invoke either a static or instance method as appropriate; a static (class-level) field returns the current static value. In this way, a Java method is accessed in the same way as a Gozer function, by writing an S-expression with the name of the Java method in the first location and the arguments in the successive locations. New instances are constructed by using the name of the class as a function (similar to the Python language). Because this resolution is done at compile-time, the compiler can perform some checks to catch spelling and other simple errors.

These method and constructors can also be used as literals and passed to mapcar and other Gozer functions that use apply or funcall. Listing 2.12 gives an example of this usage.

2.3.5 Dynamic Java Access

The Charleston College style provides a clean functional interface between Gozer and Java, but it has some limitations. The long names it uses may seem cumbersome, and although they serve as documentation on the *expected* type of the first argument, Gozer's dynamic typing may cause this to be misleading. Because of this, Gozer provides another method to access Java classes, methods and fields, modeled after

Listing 2.13: Clojure-style Java Integration

```
;; Calling instance, static, and constructor methods
(. "string" (charAt 1))           ⇒ \s
(. 'java.lang.String (valueOf 1)) ⇒ "1"
(new #Ljava.lang.String "copy")   ⇒ "copy"

;; Accessing a property
(. "abcd" bytes)                  ⇒ #(97 98 99 100)

;; Chaining with regular and reader macros
(.. object property secondProperty (methodOnSecondProperty))
#{object.property.secondProperty.methodOnSecondProperty()}

;; Null-safe chaining
(.? object maybeNull (method))
```

the approach taken by Clojure, another JVM-based Lisp implementation [17][4].

Clojure-style integration (Listing 2.13) uses the **.** special form (presumably the **.** was chosen for its resemblance to Java source code). This first argument to this form is either an instance or a *class designator* (a Class, or a symbol whose name is a fully-qualified name of a Class). The second argument is either a symbol naming a property (in the JavaBeans sense [19]), or a list consisting of the method name to call and the arguments to the method[5]. This form returns the value of the property or the return value of the method (a method that returns `void` results in NIL). New instances are constructed with the new special form, whose first argument is a class designator (a Class or the name of a class) and the arguments to be passed to the Java constructor. A reader macro and a number of regular macros make this way convenient for the programmer.

[4]In reality, the Clojure-style came first, predating the implementation of Gozer's package system. Both styles enjoy approximately equal popularity in existing Gozer programs.

[5]The use of symbols to represent Java class, property and method names is the primary reason that Gozer symbols are case-sensitive, instead of the more common case-insensitivity seen in other Lisps.

Method Selection and Type Conversion

With generic functions, Gozer provides true multiple-dispatch. All the dynamic types of all the arguments are considered when making a method selection. When invoking methods on Java classes, however, one argument, the receiver, has a special place in determining the method to use. In Java, the static type of the arguments, together with the dynamic type of the receiver would determine the method (i.e., methods can be overloaded based on argument types). To help bridge this difference, Gozer performs multiple-dispatch on all the actual dynamic types of all the arguments and will heuristically perform type conversions if no matching method can be found.

Regardless of whether the Charleston College style or Clojure style is used, final method selection is deferred until runtime (Charleston College style performs some basic sanity checks at compile time). In a form of ad-hoc polymorphism, a method of the given name and matching the actual argument types is looked for on the runtime object or class actually provided; this also permits "duck-typing" or generic algorithms (for example, both Java's String and Collection types define an isEmpty method, but share no common superclass or interface).

If there are multiple method choices due to Java overloading, the method that requires the fewest, "least-expensive" type conversions from actual arguments to formal (static) parameter types is used (if such a method can be found). Converting parameter types when calling into Java works best if at least one side of the conversion is immutable (or at least treated in an immutable fashion) because the converted object is generally temporary and not available to the calling Gozer code (converted parameters are pass-by-value). Java methods that accept primitive parameters are prepared for this (primitives are pass-by-value) but those that accept objects may not be (objects are pass-by-reference) For this reason, conversions have

Listing 2.14: Type Conversion

```
;;; Algorithms, from Apache Commons-Functor, is a Java functional library
(let ((to-find "D"))
  ;; Java signature: Algorithms.detect( Iterator<X>, UnaryPredicate<X> )
  (Algorithms.detect
    ;; list to Iterator<String>
    '("A" "B" "C" :D)
    ;; closure to UnaryPredicate, symbol to String
    (lambda (s) (string-equal to-find s)))) ⇒ :D
```

different expenses. Converting to a known immutable class such as String is considered less expensive than an arbitrary conversion. Likewise, converting from a smaller integral type to a larger integral type is considered less expensive than performing an arbitrary object conversion.

This type conversion is convenient and allows for idiomatic style in both Gozer and Java. For example, Gozer's list literals can automatically become a specific type of Java array, and Gozer functions can automatically implement Java callback interfaces (see Listing 2.14). Many of the libraries that Gozer is used with are designed with immutable types in mind, so the pass-by-value semantics are rarely surprising.

In Gozer code, built-in operations are generally polymorphic on as many types as possible. For example, the indexed sequence accessor, elt, operates on at least seven disjoint types (and all of their subtypes). In addition, the Java programmer can make their classes work with elt through a well-defined protocol (add an implementation of Groovy's getAt method [21]). This allows the Gozer programmer to write generic algorithms that work with many different types (arrays and Lists, for example). These types do not all have to exist at compile time.

Sometimes the programmer wants to limit the dynamic possibilities of the Gozer language, requiring particular types and bypassing duck-typing. The Gozer declare special form can be used to require objects of specific types be bound to specific lexical variables. The runtime enforces this, and the compiler has some limited ability to emit errors and warnings when type declaration can be proved to be, or are suspected to be, violated.

2.3.6 Common Data Types

As established before, Gozer generally shares the same data types and objects as the Java platform. There are some differences and extensions, however, and the following sections describe some of the most important.

Numerics

Arithmetic in Gozer operates on signed quantities, and may either be exact or inexact. Arithmetic with Java's integral types (`byte`, `short`, `int`, `long`) is exact in all situations and not subject to overflow or underflow; over or underflows are detected and automatically produce results in the next larger type. The largest supported integral type is Java's BigInteger class, which offers infinite precision and is usable wherever any other number is. Literal values may be input in any of these types (either automatically or explicitly).

Java's floating point types, `float` and `double`, do not get this treatment; however, Java's BigDecimal class can be used for arbitrary precision decimal arithmetic. As with the integral types, literal values may be input in any of these types.

Generally the output of numeric functions is widened to match the widest parameter type; the exception is mixing exact and inexact (binary) floating point values. Rather than present the illusion of exactness when such is not the case, the BigDecimal value is shortened to its inexact approximation.

Literal syntax is provided for Gozer's rational type. Arithmetic with rational values is exact, and rational results are automatically simplified to integer values when possible. Rational values may be freely mixed with integral and exact values. As with BigDecimal, mixing ratios with inexact floating point values yields an inexact result.

Symbols

As in all Lisp dialects, Gozer provides a symbolic data type. Symbols are tokens that may be directly written in source code and that have a guaranteed identity (the same symbol written multiple times has a single identity).

In Common Lisp, a symbol conceptually consists of a name, a package, a value, a function, and a property list. The name of a symbol is its textual representation, and the package provides a qualifier or namespace to prevent collisions (see Section 2.3.6). In contrast to the name and package name of a symbol, which are immutable, the property list allows the programmer to associate additional mutable data with a particular symbol.

The value stores the current dynamic variable binding, and the function stores the current function binding. The language automatically obtains the value of a symbol when a symbol is used in a variable location, and it obtains the function associated with a symbol when a symbol is used to name a function application. It is these two aspects of a symbol (historically referred to as the value cell and function cell of the symbol), each of which is used in a different context, that makes Common Lisp a Lisp$_2$.

Gozer's symbols have three of these four attributes; the property list is currently omitted because it is not clear how useful it would be, nor what semantics it should have with respect to multiple threads.

Boolean

The notion of true and false values is intended to yield semantics that are both intuitive and expressive. To begin, the Java boolean values `true` and `false` and their object wrappers maintain their truth values in Gozer.

In Common Lisp, only the empty list (NIL) is considered to be false. All other values are considered to be true, with the constant T representing the canonical truth value.[6] In keeping with its unification of collection types (see Section 2.5), Gozer extends this concept, and any empty sequence (a Java Collection, Iterator, Enumeration, CharSequence or array) is considered to be false. The Java `null` value is mapped to the empty list, and thus is false.

From Groovy comes the idea of having the numeric value 0 be false. While this matches the behaviour of the C family of languages, it differs from both Common Lisp and Java. All remaining values are considered to be true as in Common Lisp, but the programmer can augment this rule for specific classes of his own.

Overall, these rules lead to very expressive code. Predicate functions may return values that are useful beyond their truth value, and often loops may be written without worrying about boundary conditions (e.g., an empty collection or a null reference). Only the treatment of the value 0 can be confusing. When functions return numeric values (such as the position function), users must remember to check the numeric value as they would in Java (that is, when checking for the existence of an item, it is incorrect to write `(if (position ...) ...)` because 0 is a meaningful return value; instead one would write `(if (>=0 (position ...) ...)))`.

Structures

Gozer fully supports the notion of Common Lisp user-defined structures (or record types).[7] Structures may be instances of a Structure class (by default), or may be based on lists or arrays (for speed or for evolution of older code where lists were used for prototyping). When the Structure class is used, structure instances can

[6]In fact, the Java boolean values are mapped to NIL and T in Gozer for printing.

[7]Structures were originally added to allow Gozer to support an open source implementation of the Common Lisp LOOP macro, and are also used in the Gozer port of the SWANK package, part of the Emacs SLIME IDE.

participate in Gozer's generic functions, and can be used in a duck-typing fashion with other Gozer and Groovy code. Structures support a limited form of single inheritance.

Packages

One of the most important considerations in the building and evolution of large systems is name management. Gozer provides a mechanism in the package system for this. Similar to C++ namespaces and Java packages, Gozer packages provide a scope for name resolution. Unlike these systems but like Common Lisp, Gozer packages are fully accessible to the programmer at runtime. While name management is usually handled by the reader (see Section 2.1), the programmer can interact with this process.

Every symbol belongs to exactly one package. The symbol is said to be interned in this package. Symbols can be exported from a package and imported into other packages, or entire packages can be added to the symbol resolution of other packages. Naming conflicts are immediately detected by the runtime and result in errors.

One of the ways that Gozer integrates with Java is by making Java packages available as Gozer packages. See Section 2.3.4 for details.

Gozer's package system is closely integrated with its module system. Common Lisp deprecated its under-specified module system for portability reasons, but Gozer, running on only one platform (the JVM) has no such concerns. A Gozer module is a high-level way of providing a particular named feature, and a corresponding high-level way for declaring dependence on that feature. Conventionally, a module contains a package of the same name.

Futures

Parallel operations in Gozer are based upon Java threads and a Java thread pool (Java's ExecutorService), the native parallel primitives provided by the underlying Java platform. Although the Gozer programmer is free to use these primitives, higher-level operators based upon those from Multilisp [18] are provided. These operators are all declarative in nature and are designed to help the programmer to focus on opportunities for achieving concurrency.

The core abstraction is the *future*. A future is a datatype that represents a computation that may not have completed yet, and represents a promise to deliver the value of that computation when required at some future point in time. Until a future's computation completes the future is said to be *undetermined*, after which the future is *determined*. Any value that is not a future is always said to be determined. Futures are used to exploit opportunities for concurrency that exist between the computation of a value and its ultimate use. For example, when transforming a set by applying a function to all members of a set, the earliest time that the transformed value for the *first* member of that set could be needed is after the transformation has completed on the *last* member of the set. An opportunity for concurrency exists that can easily be expressed with a future, which in Gozer is declared with the future macro. The function par-sum-squares from Listing 2.15 is an example of this; concurrency was added to the function seq-sum-squares with the use of a judiciously placed future macro.

When a computation involves futures, the Gozer programmer generally does not need to take special precautions. Futures can freely be mixed with other values, passed to and returned from functions, stored in data structures, and so on. The GVM is responsible for managing the execution and determination of futures.

The GVM does not provide a guarantee about the sequence in which futures

Listing 2.15: Sums-of-Squares Variants

```
(defun seq-sum-squares (numbers)
  (apply #'+
         (loop for number in numbers
           collect (* number number))))

(defun par-sum-squares (numbers)
  (apply #'+
         (loop for number in numbers
           collect (future (* number number)))))
```

are determined. In the case of IO or other side effects, order of operations can be important. Gozer's touch and pcall operators allow the programmer to control sequencing in these cases. The touch operator causes the calling thread to await the determination of a particular value before proceeding, while pcall applies a function, but only after all its arguments are determined.

Another way in which sequencing can be controlled is through the use of *agents* (based on those found in Clojure). An agent has a state (some user supplied data) and atomically updates that state by applying a user supplied function to it. State updates are guaranteed to be carried out in the order in which they are submitted (via the send function), and they are also guaranteed to be carried out in only one thread at a time. Agents make use of futures in their implementation (so they exploit parallelism), but unlike futures, which are tightly integrated into the GVM and automatically determine their values, the programmer is responsible for requesting the state of an agent when needed (using the deref function). The programmer can choose futures or agents based upon the needs of the problem: plain futures work well for transparently adding parallelism to existing functional, side-effect free code but become more cumbersome when sequence is important; agents easily support mutating state and sequential ordering, but are never transparent to the programmer.

2.4 Condition System

Many languages, including C++ and Java, support the notion of *exceptions*. Exceptions are a form of non-local control flow in which one portion of the program can *throw* an exception which is *caught* by one of its dynamic callers. Intermediate call frames are unceremoniously unwound and exited. Java includes a mechanism (the `finally` block) that allows frames being unwound to perform some action (often resource cleanup) as they are being exited. In object-oriented languages, exceptions are invariably arranged in a class hierarchy that allows callers to selectively catch particular types of exceptions.

As the name suggests, exceptions are generally reserved for exceptional conditions, most often error handling. Although exceptions are undoubtedly an improvement over the C-style of error handling (through function return values that are often ignored), they are relatively inflexible due to their unconditional stack-unwinding and two-part division into something that throws an exception and something that catches it. Common Lisp offers a superset of this functionality in its condition system. Flexibility is increased by dividing the responsibilities into three parts: signaling a condition, handling it, and restarting [37]. Gozer provides an implementation of the Common Lisp condition system that incorporates Java exceptions along with named conditions and is continuation-safe for the distributed programming case.

A *condition* is any circumstance that a called portion of the program would like to communicate to its dynamic callers. It does not necessarily represent an error, it could be a warning or merely informative. It is said that the condition is *signaled* to the callers. The act of signaling itself does not cause the stack to unwind. The condition is passed to each applicable *handler* that has been established by the dynamic callers in turn (from most recent to least recent caller) until one handler chooses to *handle* the condition. (If no handler handles the condition, the results depend upon the type of signal. What it means to handle a condition is explained in Section 2.4.2.) The signaler may provide options to its handlers by defining *restarts*, and a handler may choose to handle a condition by invoking a restart.

2.4.1 Programmer Interface

The main programmer interface to the condition system consists of just a few functions and macros, plus the definition of a condition itself. In Gozer, conditions are represented as objects. They may either be Java Throwable objects, in which case the Java class hierarchy is used to determine handler applicability, or they may be XML namespace-qualified names, QNames such as {urn:BlueBox}InvalidSession [4]. The use of QNames allows the Gozer programmer to define, signal, and handle his own conditions without dropping into Java to write new Throwable subclasses.[8] This is similar to the use of strings as exceptions in earlier versions of the Python programming language.

The basic function for signaling a condition is called signal. Given a condition, this function simply determines all the applicable active handlers and invokes each one in turn. If no handler handles the condition, this function returns normally with no additional action. Built on top of signal are the more semantic functions warn and error. The former function is like signal, but if no handler handles the condition, it is printed as a warning. The latter function is again like signal, but differs in that if the condition is unhandled, the debugger is invoked (if the program is running interactively) or the program is terminated (if the program is running non-interactively); error can never return normally.

Handlers are simply functions. Handlers are established using the handler-bind macro, which allows the programmer to specify an ordered list of condition types or names that will be active during the dynamic execution of the body and the

[8]It also has uses in distributed workflows as discussed in Section 4.2.5.

function (often an anonymous function) in invoke when the matching condition is signaled. For the common case of handling a signal (an error) by unwinding the stack and executing some code defined at the point the handler was established, the handler-case macro is provided.

The real power of the condition system comes from its separation of condition handling from error recovery. As mentioned, the handler-case macro ties these two steps back together to allow for Java-style `try`/`catch` error recovery, but sometimes that's not the best way to handle errors. By unwinding the stack, all the intermediate work-in-progress is lost, work that might be painful or infeasible to recreate. The notion of a restart allows high-level code to direct the error recovery process using options provided by lower level code without unwinding the stack. The code that wishes to provide alternatives does so by establishing named restarts using the macro restart-bind (or its higher-level cousin restart-case). Like handlers, restarts are simply functions. During the dynamic execution of the body of these forms, these restarts may be invoked by name using the invoke-restart function. Invoking a restart is a way to handle a condition and control is returned from the restart-case form with the value returned by the restart function.

Listing 2.16 shows a function that does computationally expensive things (the function do-expensive-stuff), but may sometimes signal an error. The highest level of code (the function high-level) needs to have this expensive computation done with different parameters that are calculated by an intermediate function. The intermediate function cannot know the best way to handle the failure of the expensive computation—sometimes it might have been critical to have every correct answer, sometimes a faster approximation or other value might do—so it establishes a restart called use-value. The highest level of code can then determine the policy of error handling. In the example, the high level code chooses to ignore the error and re-

Listing 2.16: Condition Example

```
(defun do-expensive-stuff (i)
  "Does expensive work. May signal an error."
  (if (= 3 i)
      (error "Failed on 3")
      i))

(defun expensive-loop (max)
  "Does expensive stuff `max' times with different parameters."
  (loop for x from 0 upto max
        collect (restart-case (do-expensive-stuff x)
                  (use-value (value) value))))

(defun high-level ()
  "When an error is signaled, uses a restart"
  (handler-bind ((error (lambda (e) (invoke-restart 'use-value
                                                     'from-high-level))))
    (expensive-loop 5)))

(high-level) ⇒ (0 1 2 'from-high-level 4 5)
```

place the missing computation with a distinguished value. In this way, all of the computations that do complete successfully are collected and no work is wasted. The function call tree for Listing 2.16 is visualized in Figure 2.1.

2.4.2 Implementation

Gozer conditions are implemented almost entirely in ordinary Gozer code. The only special support required from the Gozer Virtual Machine is in bridging exceptions thrown by Java code into signals that Gozer can then handle, and correct operation in the face of nested GVMs (see Section 3.4). Ordinary functions, closures, the control-flow operators tagbody/go and block/return-from, and dynamic variables are enough to implement the entire condition system.[9] Knowing that, we can answer the two unanswered questions of condition system implementation.

First, if handlers are just functions, and conditions are signaled by calling a function such as error which never returns, what does it mean to handle a condition?

[9]Obviously, such an approach doesn't admit as many optimization opportunities as an approach integrated with the Virtual Machine, but signal handling is believed to never be a bottleneck.

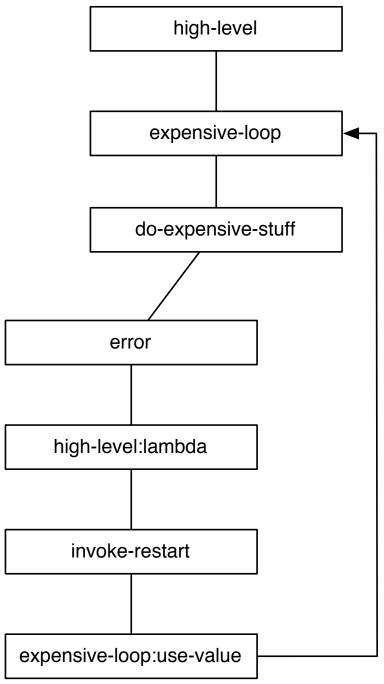


Figure 2.1: Condition Example Call Tree

Handling a condition simply means making a non-local exit out of the dynamic scope of the signaling function. A macro such as restart-case uses a tagbody to create targets to which a non-local exit can jump using a go command. It further creates closures within that tagbody that jump to those targets. When invoke-restart comes along later and invokes that closure, the closed-over target (presumably somewhere up the call tree) is found and the non-local exit arrives there. A simplified version of the macroexpansion of the restart-case contained in Listing 2.16 is presented in Listing 2.17.

Second, what does it mean to establish a restart or handler, and how are the active restarts and handlers tracked? This is actually the simpler question as the stack-like behaviour of Gozer's dynamic variables completely handles this. A linked list of the active restart and handler functions (together with some metadata about them such as a handlers applicability) are each contained in a dynamic variable. Establishing a handler or restart means creating a new binding for the respective

Listing 2.17: Restart-Case Expansion

```
(block #:block-G1371
  (tagbody (restart-bind ((use-value
                            (lambda (&rest rest)
                              (setq *rest-temp* rest)
                              (go #:tag-G1372))))
             (return-from #:block-G1371
               (do-expensive-stuff x)))
     #:tag-G1372
     (return-from #:block-G1371
       (apply (lambda (value) value) *rest-temp*))))
```

variable with let and pushing the new handler or restart on the front of this binding. At any point in time, examining these variables will yield their current bindings and thus the active handlers or restarts. As with all dynamic variables, exiting the binding scope automatically restores the previous value.

2.5 Standard Library

A Gozer programmer immediately has access to all the classes and functions provided by the immense Java standard library, as well as the additions provided by the Groovy runtime. The goal of the Gozer standard library, then, is to provide convenient ways to exploit Gozer's capabilities. For example, Java provides the overloaded static method Collections.sort to destructively sort a List, and nine overloaded versions of the static method Arrays.sort (one for each primitive type plus references) to destructively sort arrays. Groovy allows these to be used as instance methods on any Collection or array to non-destructively sort the instance. Gozer's single unified sort function applies Gozer's type conversion rules to be able to sort not just Collections and arrays but also Maps, Iterators and anything else that can be treated an iterable sequence. Figure 2.2 shows how many different Java interfaces (ovals) and classes (rectangles) are able to be treated as iterable; the dashed lines indicate conceptual relationships not present in the Java inheritance hierarchy.

Currently, the standard library consists of about 400 functions, macros and

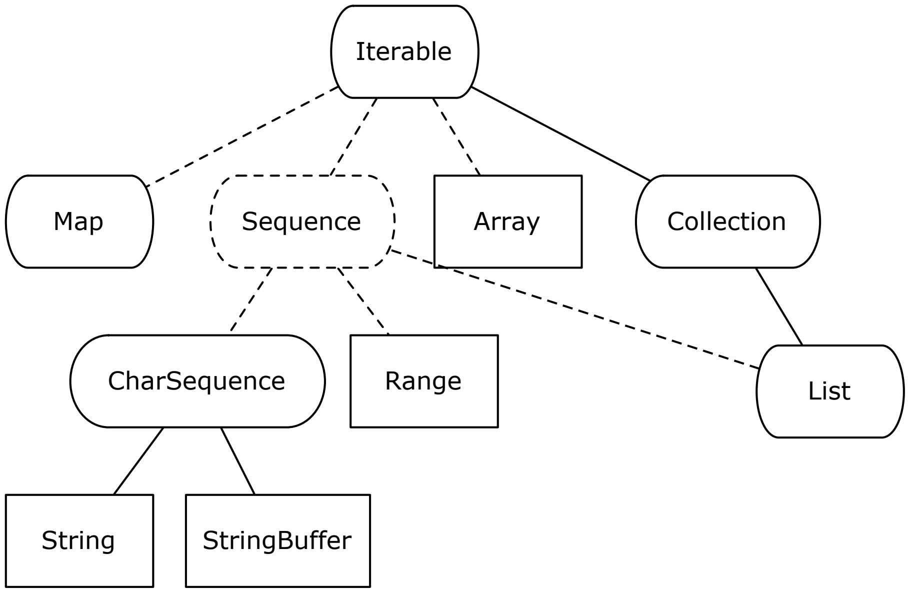


Figure 2.2: Conceptual Type Hierarchy

special forms (in comparison, the Common Lisp standard provides 752 functions, macros and special forms). Approximately 60 of those are implemented in Java, with the rest being implemented in about 4,000 lines of Gozer code. Much of the standard library is based on that of Common Lisp, but implemented in a more general way (for instance, the Common Lisp sort function is only specified to work on vectors and proper lists). Functions are provided for working with sequences, inspecting the runtime environment, type handling, string manipulation and regular expression matching, reading and printing Gozer data and code, etc. Most of the macros are for defining entities (like defun), provide control constructs (like iteration), or capture common patterns of resource usage. The standard library has grown organically, with Common Lisp or Gozer-specific functionality being added on demand and as time permits.

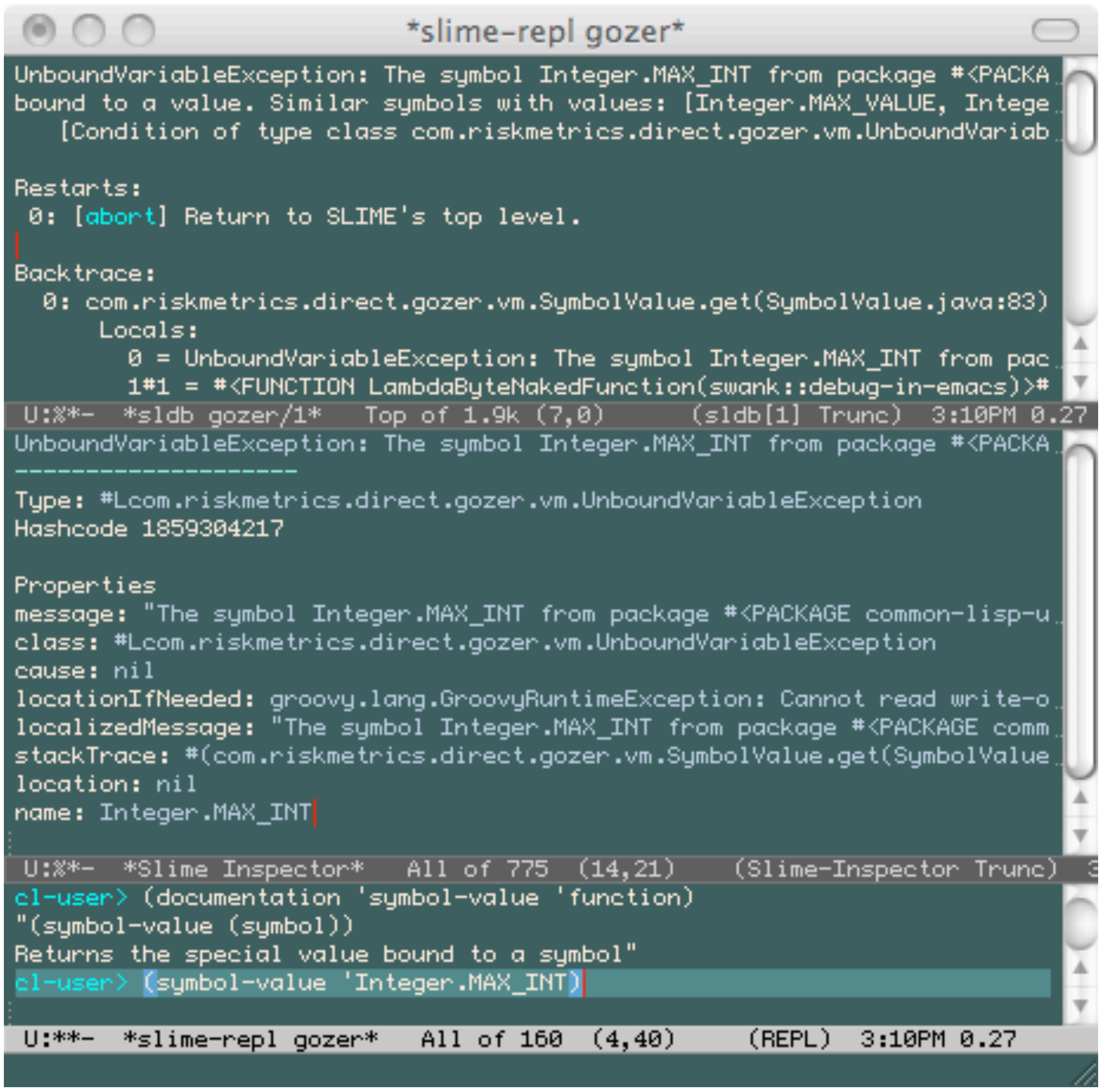
2.6 Development Environment and Tools

An important factor in the usability of a language is the suite of tools that are available to work with it. In particular, when working in a Lisp-derived language, a competent programmer's editor is a necessity. Fortunately, the open source and widely available GNU Emacs[10] editor has a long history as a Lisp editor [5] and offers many generic packages for editing Lisp code of any dialect; one powerful example is Paredit[11], a package that provides semantic S-expression editing (as opposed to simple text editing). Gozer programmers can take advantage of these editing tools in much the same way they were able to use XML editors in the previous workflow system. However, the expanded capabilities of a programming language are often better served with an interactive development environment (IDE).

Here again, the capabilities of GNU Emacs proved valuable. A mature open source package named Slime[12] exists to provide GNU Emacs with a Lisp IDE possessing features such as configurable, context-sensitive completion, a powerful REPL, runtime inspection capabilities, documentation and hyperlinked source lookup, and more. Slime is designed to be portable across different Lisp implementations, and to this end it is cleanly divided into two layers: one that runs in Emacs and manages user interaction and one (known as Swank) that executes in the target Lisp implementation and provides access to the Lisp's capabilities. The two layers communicate by exchanging simple S-expressions (operating as Remote Procedure Calls or RPC), and, because Slime is designed with portability in mind, only a small core of features is mandatory in order to begin using Slime; it is possible to incrementally support more and more features.

Developing a Swank implementation for Gozer seemed to be a natural way to quickly provide Gozer programmers with IDE support. The Gozer Swank implementation consists of approximately 2,500 lines of code, all written in Gozer, that implement some of the most important Swank functionality. This includes full sup-

[10]Available from `http://www.gnu.org/software/emacs/`.
[11]Available from `http://mumble.net/~campbell/emacs/paredit.el`.
[12]Available from `http://common-lisp.net/project/slime/`.



```
*slime-repl gozer*
UnboundVariableException: The symbol Integer.MAX_INT from package #<PACKA
bound to a value. Similar symbols with values: [Integer.MAX_VALUE, Intege
   [Condition of type class com.riskmetrics.direct.gozer.vm.UnboundVariab

Restarts:
 0: [abort] Return to SLIME's top level.

Backtrace:
  0: com.riskmetrics.direct.gozer.vm.SymbolValue.get(SymbolValue.java:83)
      Locals:
        0 = UnboundVariableException: The symbol Integer.MAX_INT from pac
        1#1 = #<FUNCTION LambdaByteNakedFunction(swank::debug-in-emacs)>#
 U:%*-  *sldb gozer/1*   Top of 1.9k (7,0)      (sldb[1] Trunc)  3:10PM 0.27
UnboundVariableException: The symbol Integer.MAX_INT from package #<PACKA
--------------------
Type: #Lcom.riskmetrics.direct.gozer.vm.UnboundVariableException
Hashcode 1859304217

Properties
message: "The symbol Integer.MAX_INT from package #<PACKAGE common-lisp-u
class: #Lcom.riskmetrics.direct.gozer.vm.UnboundVariableException
cause: nil
locationIfNeeded: groovy.lang.GroovyRuntimeException: Cannot read write-o
localizedMessage: "The symbol Integer.MAX_INT from package #<PACKAGE comm
stackTrace: #(com.riskmetrics.direct.gozer.vm.SymbolValue.get(SymbolValue
location: nil
name: Integer.MAX_INT

 U:%*-  *Slime Inspector*   All of 775  (14,21)    (Slime-Inspector Trunc)  3
cl-user> (documentation 'symbol-value 'function)
"(symbol-value (symbol))
Returns the special value bound to a symbol"
cl-user> (symbol-value 'Integer.MAX_INT)

 U:**-  *slime-repl gozer*   All of 160  (4,40)     (REPL)  3:10PM 0.27
```

Figure 2.3: Gozer Debugger and Inspector

port for the REPL, “compiling” source files and annotating them with errors and warnings reported by the compiler, a rudimentary debugger and object inspector (see Figure 2.3), cross-referencing functions, and source and documentation browsing—both for Gozer code and Java classes and methods it interacts with, even when packaged inside a standard Java Archive (JAR) file. Slime's open source development proceeds rapidly, which often leads to changes in the required RPC functions. This, unfortunately, means that Gozer programmers are frequently required to use an older version of Slime until such time as Gozer's Swank implementation adapts to the changes.

Partially to compensate for this characteristic lag and partly to demonstrate Gozer programming, a separate Web-based documentation searching and browsing application was developed by Matthew Martin. This entire application requires only about 400 lines of code. Additionally, an application for generating static HTML documentation for an entire Gozer system, in the spirit of Java's Javadoc, was developed by Matthew Martin and Joshua Zuech and occupies about 200 lines of code.

The standard library, Gozer tooling, the Vinz distributed workflow functionality, and all user-written Gozer code, requires a runtime environment. This environment is provided by the Gozer Virtual Machine, described in the next chapter.

Chapter 3

The Gozer Virtual Machine and Compiler

The Gozer Virtual Machine (GVM) implements the runtime semantics of the Gozer programming language using a bytecode interpreter running on top of the Java Virtual Machine. The bytecode is produced by a compilation process implemented jointly in Java and Gozer. The GVM itself is implemented in Java code, and it makes heavy use of support functions provided by the Groovy runtime. No tree-walking interpreter is supplied, and code is always compiled for execution (even expressions passed to the `eval` function are first compiled). This chapter will first describe the virtual machine before moving on to the compiler.

3.1 GVM Architecture

The GVM implements a stack-oriented architecture, in many ways similar to the JVM's architecture. The bytecode provides instructions for pushing values onto the stack, and function calls consume arguments from, and leave results on, the stack. The need for transparent process migration across machines is a key factor guiding many design decisions.

Lexical variables may be accessed by indexing into an array associated with the runtime stack frame, with the compiler mapping from programmer symbols to bytecode indexes. To support lexical closures, a garbage-collected "heap" may also be used for variables[1]; since the heap is much slower to access than function-local storage, the compiler only uses it when primitive escape analysis shows that a variable *may* be closed over.[2] Global (dynamic) variables are always allocated in their own heap. The Gozer language semantics for variables allow the same variable, local or global, to be assigned different values within nested lexical scopes, and the GVM supports this directly through deep binding of heap-allocated variables. The compiler uses different indexes to simulate shallow binding when achieving this for local variables [2].[3]

Compiled code is divided into segments of up to 64K bytecode instructions. Each segment is associated with a constant pool of up to 64K objects referenced from the code segment.[4] There is usually a one-to-one correspondence between a compilation unit such as a file and a code segment. Once created, segments are immutable, which allows them to be effectively cached in memory on multiple machines during process migration.

3.1.1 Image

Most Lisp implementations, like Smalltalk, are image-based, and Gozer is no exception. An image is simply all the programmer-visible state at any given time, including current global variable bindings, current function bindings, the set of loaded modules and packages, etc. Images can usually be saved to persistent storage and

[1]Some Lisp systems are able to use indexes for closed-over variables as well, by capturing the array associated with the stack frame in the closure. The Gozer runtime is not yet this sophisticated.

[2]If the compiler throws away symbol names and translates them to index numbers, this makes debugging more difficult and also interferes with the ability for low-level functions to access variables by name, so, with `declare` or `declaim`, the compiler may be instructed to use the heap for all variables.

[3]As a side effect of this, a particular value may have an active reference long after the programmer considers it dead. This interferes with garbage collection and potentially process migration, so the compiler inserts specific instructions to remove this extra reference. These instructions may be omitted in speed optimized code.

[4]The 64K limit comes from the use of 16-bit quantities to represent bytecode offsets and constant pool indexes as detailed in Section 3.3.1.

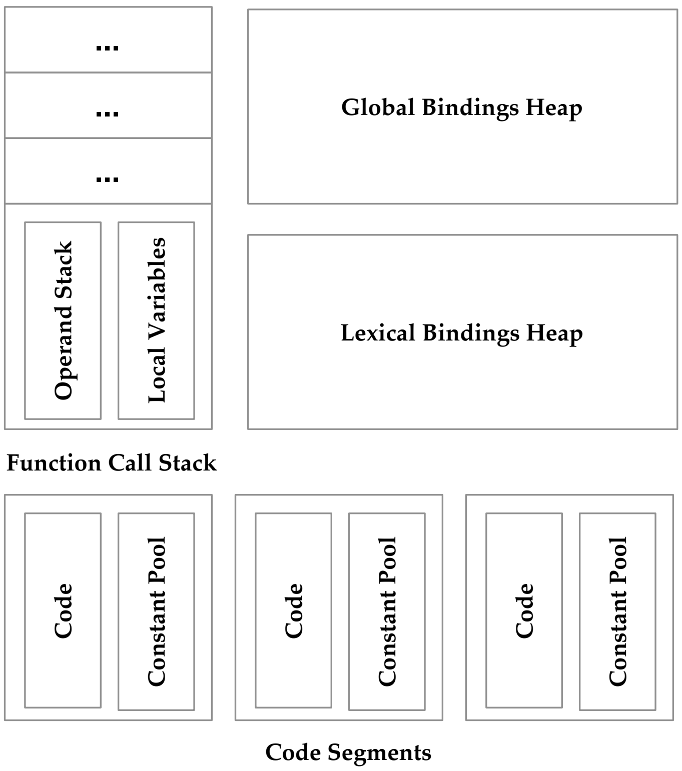


Figure 3.1: GVM Concepts

later be loaded again to continue from the saved state. A Gozer image can be saved together with a snapshot of the active function call stack and migrated to another machine, whereupon execution can continue.

A Gozer image is initially created through a bootstrapping process. Java code creates a set of empty data structures and populates them with a small set of functions and variable bindings, just enough to start reading, compiling, and executing the most basic Gozer code. The infant GVM is then handed Gozer expressions and told to begin executing them. After each expression is compiled and evaluated, the image is a little more complete and able to evaluate more complicated expressions.

As described previously in the section on evaluation (Section 2.2), the Gozer language does not distinguish between compile time, load time, and evaluation time. Loading a source file always results in its compilation, and as each top level expression is encountered and compiled, it is evaluated. Thus, there are no special contortions (Common Lisp's eval-when) necessary to arrange for defun to properly register a function when a file is loaded—the evaluation of the top level defun accomplishes that as a side effect, altering the state of the image. This model simplifies both the current implementation and a programmer's intuition (a file is simply a list of executable statements, processed from top to bottom), but it does complicate the idea of individually compiling a file into a unit that can then be loaded into an arbitrary other Gozer image (e.g., Common Lisp's "FASL" files). Fortunately, Gozer's use cases are compatible with always distributing and loading the source code.

3.2 Function Calling Convention

The Gozer calling convention allows for required positional arguments, optional positional arguments (which may have a default value that is an evaluated expression, and may bind a second local variable to indicate if the value was given to the function call or computed by default), optional named arguments (*keyword* arguments, which may also have default values and may bind an indicator variable), and variable-arity argument lists (*rest* arguments, which collect all remaining arguments into a list). The argument types may be freely mixed.

Conceptually, to call a function, the caller first pushes the arguments to the function onto the operand stack in a left-to-right order. When the function call instruction is encountered, the GVM pops the indicated number of arguments off the stack, pushing them into another stack (a different Java object) which it then places in the argument register before passing control to the called function.

The called function begins with a sequence of instructions (its prologue) that operate on the stack in the argument register, binding arguments to local lexical variables.

In principle, an argument is popped from the argument register onto the operand

Listing 3.1: Required Argument Assembly

```
;; Required Argument (optimized for speed)
(lambda (x) (declare (optimize (speed 3) (safety 0)) x)
;; Intermediate
(begin
      (allocate-local 1)
      (%pop-and-bind x 0)
      (no-op)
      (push-local x 0))

;; Required Argument (optimized for safety)
(lambda (x) (declare (optimize (speed 0) (safety 3)) x)
;; Intermediate
(begin
      (%pop-arg)
      (bind-var x nil)
      (%pop-no-args)
      (push-var x))
```

stack, and then any of the normal binding instructions can be used to assign it to a lexical variable. In practice, because argument passing and binding is such a common operation, the GVM provides one macro instruction for the most common case of binding a required argument to a local register index. The GVM provides an instruction that tests for the existence of a remaining argument, and this is combined with standard jump and binding instructions to implement optional positional arguments. Likewise, standard jump and binding instructions together with an instruction to test for and extract a named parameter are employed to implement keyword arguments. An instruction transforms the remaining contents of the argument register into a list and is combined with a binding instruction for rest arguments.

The instructions dealing with required arguments raise an error if there are not enough arguments provided (i.e., if the argument stack is prematurely empty). The final instruction in the prologue makes sure that all arguments have been consumed, raising an error if too many arguments were provided.[5] In cases where the target

[5]In optimized code, this instruction may be omitted.

Listing 3.2: Optional Argument Assembly

```
;; Optional Argument
(lambda (x &optional y) y)
;; Intermediate
(begin
      (allocate-local 2)
      (%pop-and-bind x 0)
      (%pop-if-next)
      (jump-if-true #<Label@2089263258># forward)
      (push-constant nil)
      (goto #<Label@742465109># forward)
      (label #<Label@2089263258># nil)
      (no-op)
      (label #<Label@742465109># nil)
      (set-local y nil 1)
      (%pop-no-args)
      (push-local y 1))
```

function is known at compile-time, the compiler can also perform these checks, but they can only be advisory because late-binding means a different function may actually be called at runtime.

Function calling is, perhaps, the most frequent operation in a Lisp-like language, so it makes sense to design the virtual machine and instruction set to make function calling as fast as possible. Early iterations of the GVM did not provide an argument register nor the instructions that operate on it, instead passing all arguments via a single Object array placed on the operand stack; sequences of ordinary instructions were then used to inspect this array and extract values from it. The introduction of the argument register and related instructions reduced bytecode size and had a significant runtime performance impact.

Function arguments are pushed onto the operand stack in a left-to-right manner, but the called function needs to operate on them in a FIFO manner. That is, the called function needs to find the first passed argument on the top of its argument register.[6] This can be accomplished by popping from one stack and pushing on

[6]In early versions of the GVM, arguments were evaluated and pushed right-to-left and so this property arose naturally. This violated programmer expectations, however, and the Gozer language specification was changed to use the more common left-to-right order

Listing 3.3: Keyword Argument Assembly

```
;; Keyword argument
(lambda (x &key y) y)
;; Intermediate
(begin
      (allocate-local 2)
      (%pop-and-bind x 0)
      (%pop-kw :y)
      (jump-if-true #<Label@439695831># forward)
      (push-constant nil)
      (goto #<Label@476602290># forward)
      (label #<Label@439695831># nil)
      (no-op)
      (label #<Label@476602290># nil)
      (set-local y nil 1)
      (%pop-no-args)
      (push-local y 1))
```

Listing 3.4: Complex Keyword Argument Assembly

```
;; Keyword with default value and supplied-p param
(lambda (x &key (y "default" y-sup-p)) y)
;; Intermediate
(begin
      (allocate-local 3)
      (%pop-and-bind x 0)
      (%pop-kw2 :y)
      (jump-if-true #<Label@1593598058># forward)
      (push-constant \"default\")
      (goto #<Label@186515422># forward)
      (label #<Label@1593598058># nil)
      (no-op)
      (label #<Label@186515422># nil)
      (set-local y nil 1)
      (no-op)
      (set-local y-sup-p nil 2)
      (%pop-no-args)
      (push-local y 1))
```

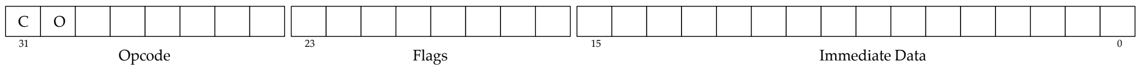


Figure 3.2: Gozer Instruction Format

another (as in the conceptual description), but to make this process more efficient—and particularly to avoid allocating temporary memory—the GVM simply reverses the apparent direction of a portion of the operand stack. In this way, the argument register stack becomes simply a reversed window onto the main operand stack, and function calling overhead is minimized.

3.3 Bytecode Design

In designing the bytecode for a virtual machine, as in designing the instruction set for a physical machine, tradeoffs must be made. The Gozer bytecode emphasizes ease of interpretation and compilation through a very regular instruction format. This comes at the expense of some degree of compactness, and, for rare instruction sequences, some potential performance degradation.

The design of the Gozer bytecode reflects some ideas found in the GNU CLisp bytecode [16], as well as some ideas found in the Scheme interpreter developed in [30].

Notably absent from the Gozer bytecode are arithmetic and logic instructions. In Lisp, arithmetic is written as a function application like any other, and, in fact, Gozer implements arithmetic using functions. This is not as slow as it might seem, for reasons which will be explained shortly.

3.3.1 Instruction Format

A Gozer bytecode is a 32-bit quantity. The leftmost 8 bits specify the *opcode* or unit of dispatch, the next 8 bits are used for opcode dependent *flags* which modify the behaviour of the operation, and the remaining 16 bits are always taken as an unsigned integer.

Within the opcode, the leftmost bit is used as a flag to indicate that the low 16 bits should be interpreted as the index of a constant in the constant pool. The second bit is used as a flag that the low 16 bits are an offset in the bytecode stream. These two flags are a convenience in the instruction decoder, but are primarily used to make disassembly of the bytecode simpler—the disassembler uses them to display constant values and textual labels in a generic fashion, without having to understand the format of each individual instruction. These two bits worth of flags mean that the number of possible distinct opcodes is reduced from 256 (2^8) to 64 (2^6). Half of these are currently defined. These opcodes are described in Appendix A.

Function Call Opcodes

Almost half of the defined bytecodes are related to function calling. Four of them are different ways to call functions, and the remainder are the argument parsing opcodes described in Section 3.2. Each of the function calling opcodes uses its flag byte to encode the number of arguments the function is being called with. For implementation simplicity, Java's primitive `byte` type, which is signed, is used in the instruction decoding phase, thus limiting the maximum number of arguments to a function to be 127.

The `JSR` (Jump SubRoutine) opcode is used when one Gozer function calls another Gozer function. Gozer is late-bound, so functions are usually called by name, and the constant index refers to the symbol that names the function. Function lookup proceeds using this name and control is passed to the located function. For system-provided functions, however, the constant index may actually refer directly to the function object, and the name lookup step is skipped for improved performance.

The `JPRIMITIVE` (Jump Primitive) opcode is used to implement certain low-level functions (including arithmetic) and special forms. The constant is a reference to a system-provided function object that implements the required functionality. JPRIMITIVE is used in Gozer where other systems might typically add additional special-purpose bytecodes. Because primitive functions have full access to the Gozer environment, and effectively serve as extensions to the interpreter, they raise the level of Gozer's *virtual machine interface* [32].

Integration with arbitrary Java or Groovy methods (see Section 2.3.4) is provided through the JJAVA opcode. This opcode directly corresponds to the Gozer language · special form, and is also emitted when symbols from imported Java packages are used in the function place of a form. The constant entry for this opcode is only a reference to the name of the method to call. The actual method is found at runtime using the type of the provided 'this' object and the types of the provided parameters. This makes interaction with Java extremely dynamic and facilitates code evolution and "duck-typing," but it does extract a performance penalty. In the future, the compiler might support declarations that tell it to resolve Java references statically,and the constant index would then refer to an actual Java Method object.

The final function call opcode defined is also the most special purpose. JRECUR was initially added to support recursive local functions defined with the labels special form. Local functions created this way must shadow any global function of the same name, so they cannot be found by name. Furthermore, local functions are closures created on-demand, so they also cannot be placed in the constant pool and called with that form of JSR. Instead, JRECUR looks on the runtime function call stack to locate the actual function object currently executing, and passes control back to (a new invocation of) it. This proved to be a worthwhile optimization to make for general recursive functions as well.[7]

[7]This means that recursive functions are not late bound to themselves, making it impossible to trace them.

3.4 Java Exception Handling

In Gozer, errors are signaled and handled through the condition system of Section 2.4. Java, on the other hand, uses the `try`/`catch` system of exception handling. The responsibility of integrating the two disparate systems falls to the GVM. The integration must function in both directions, allowing Gozer code that invokes Java code to handle Java exceptions, and allowing Java code that invokes Gozer code to catch Gozer signals.

Arbitrary Java code is invoked when interpreting an `JPRIMITIVE` instruction, and especially when interpreting an `JJAVA` instruction, so arbitrary Java exceptions (subclasses of Java's Throwable) may be thrown at those times. The GVM interpreter loop catches all these exceptions by catching Throwable. In the `catch` block, the GVM constructs a new stack frame containing a call to Gozer's error function[8], with the caught exception as the top argument on the argument stack. This frame is pushed onto the evaluation stack, and the loop continues its next iteration. In effect, it is as if the Java code itself had called error. In this way, a Java exception is seamlessly transformed into a Gozer signal. Of course, the Java stack has already unwound and so Java code cannot supply Gozer restarts; however Gozer code that called Java can.

When the error function is invoked, either directly by Gozer code or indirectly by the GVM in response to an exception, it does not return normally. It must either transfer control to a handler function, invoke a debugger, or, when no debugger is available, terminate Gozer execution (unwinding the stack as appropriate) and propagate the exception (throw it) to the calling Java code. In order to cause execution to terminate, the running Gozer error function must be able to communicate with

[8]If the error function is not bound, as may be the case if an exception is thrown early in the bootstrap process, the exception is simply allowed to propagate.

Listing 3.5: Simple Nested Interpreter

```
;;; pseudo-code demonstrating how nested interpreters
;;; arise and how control-transfer must pass from a nested
;;; interpreter to the outer interpreter
(handler-case
    ;; gozer code establishes handler,
    ;; then calls Java
    try {
      //Java calls back into Gozer
      //within a try/catch block
      (call-into-gozer)
    }
    finally {
      importantCleanup();
    }
  (error (e) ...))
```

the GVM interpreter in which it is running. Although an opcode could be provided for this purpose, the fixed limit on potential opcodes made it desirable to use a more general mechanism. Also, there are times when Java primitive functions would like to communicate with the running GVM, and they cannot use an opcode (or the return value, which was already used). Java exceptions are used to implement this more general mechanism, together with the Java primitive function %throw. The GVM defines a subclass of Throwable, VMOnlyException, which is caught and handled specially by the GVM, before the general error-handling mechanism. To cause execution to cease, the error function throws (using %throw) a VMOnlyException which the GVM catches and then re-throws to Java. In this way, Java code is able to catch and handle Gozer signals that are not handled within Gozer.

3.4.1 Nested Interpreters

A particularly subtle case arises when there are nested interpreters running and the nested interpreter catches a Java exception.[9] That is, when Gozer code calls Java code that in turn calls back into Gozer code which ultimately throws a exception.

[9]Actually, this case arises for any signal raised by a nested interpreter but the treatment is the same and involves the support of the GVM.

This happens, for example, when Gozer code supplies an implementation of a Java callback interface through new-proxy or automatic function coercion. The nested interpreter is running the same image as the enclosing interpreter, so any Gozer signal handlers established will be visible, including those that would transfer control "above" the nesting. Yet if the nested interpreter itself allowed this control transfer, the interpreter would still be nested and any cleanup established through `finally` blocks by the Java code would not execute and any Java `catch` blocks that should catch the exception before propagating it to the caller would be bypassed. When the nested interpreter ultimately did complete execution, the Java code would likely be in an inconsistent state, as would the outer interpreter.

Listing 3.5 shows a simple pseudocode example of this situation. For simplicity, the example illustrates only one level of nesting; however, nesting of any depth is possible. Further complications arise if the nested Gozer interpreters introduced unwind-protect forms that must be honored, and if the enclosing interpreter wants to handle the exception not by unwinding the stack but by invoking a restart established in the nested interpreter (Listing 3.6). The GVM properly handles nesting to arbitrary depths and correctly unwinds the Gozer stack when a transfer of control occurs across nesting levels, but it does not yet correctly handle the case of a restart (or other non-local control flow forms encountered while unwinding the stack), assuming instead that the stack should always be unwound to the handler. In practice, this has been adequate as complex signal handling is usually relatively localized and unlikely to cross interpreter boundaries.

Handling nested Gozer/Java calls is the single most complicated area of the Gozer condition system. There are four parts involved in handling nested interpreters. The first is knowing whether or not the current interpreter is nested, and thus whether or not an unhandled exception should be propagated or passed to the

Listing 3.6: Nested Interpreter With Restort

```
(handler-bind ((error (lambda (e) (invoke-restart 'continue))))
  ;; gozer code establishes handler,
  ;; then calls Java
  try {
    //Java calls back into Gozer
    //within a try/catch block.
    //Gozer establishes a restart
    (restart-case (raise-error)
      (continue () nil))
  }
  finally {
    importantCleanup();
  })
```

enclosing interpreter. The GVM does this by tracking its nesting depth using a Java thread-local counter. Each entry into the GVM increments the depth, and each exit decrements it. The second part is handling the Gozer stack unwinding at the correct time and in the correct interpreter. This is also accomplished through the use of a thread-local variable, with each nested interpreter contributing any necessary cleanup items to be executed by the handler. Third, determining whether an exception is handled by code that has executed within the nested interpreter requires not just the simple search of handler bindings performed in the non-nested case, but a search of handler bindings only back to the depth established upon entry to the nested interpreter, which requires knowledge of how handler bindings are established in the environment. Finally, when a transfer of control from one interpreter to another is required, a GVM-only exception (OneLevelUp) is thrown and Java unwinds the stack from one interpreter to the container.

3.5 The Compiler

The compiler is an important component of the Gozer system, occupying as much or more code than the GVM's bytecode interpreter. Like the reader, the compiler is a very programmable piece of the system, potentially interacting with programmer-provided code during the compilation process. Beyond simply transforming Gozer S-expressions into GVM bytecode, the compiler has a few other responsibilities. The remainder of this chapter describes the compilation process; the term compiler is used generically to refer to all the components involved in this process (e.g, the assembler and the optimizer).

3.5.1 Compilation Phases

Compilation begins when the reader passes an S-expression it has produced and expanded any macros present in it (which themselves were previously compiled and were then executed by the GVM to produce the expansion)[10] to the compiler. The intermediate output of the compiler is an S-expression representing the primitive operations supported by the GVM, a form of assembly language. Each primitive operation is represented as a list, usually corresponds to one GVM opcode, and is analogous to an instruction mnemonic in machine assembly. This S-expression is finally assembled into bytecodes by resolving labels, encoding constants, and producing a finished instruction stream.

The initial S-expression is recursively traversed until the process reaches a literal constant value, a symbol, or a special form—anything that remains must be a function application. Following the evaluation rules, constant values in incoming S-expressions simply become constant values in the assembly.

Symbols are variable references (or constants imported from Java packages), and primitive assembly for either a lexical (heap or indexed) or dynamic variable access must be output. In order to produce the correct primitive, the compiler must keep track of the names and locations of known lexical variables and imported constants (anything else is assumed dynamic). The GVM keeps track of imported Java

[10]As a practical matter, this expansion is handled by some of the same code that handles the later phases so that logic about name resolution, in particular, lexical macros, can be shared.

constants (through side effects of the import-from-java macro), but lexical variable bookkeeping is handled by the let special form itself.

Special Forms and Functions

Special forms are found in the first position in a list, the same as function applications. This allows the compiler to use the same mechanism to resolve functions as it does special forms. A special form is represented by an implementation of a particular Java interface (and special forms are always implemented in Java). The compiler delegates the compilation of such a form to the special form Java object.

Sometimes, the special form object will modify the state of the current compiler (as with the declare form when variables are given types or functions are deprecated). More often, the special form will create a second, nested, compiler that contains new information and use this new compiler instance to recursively compile its body, before returning the result to the original compiler. In the case of let, for example, the special form will create a new compiler with knowledge of the newly introduced lexical variables and this compiler will be able to reference the correct lexical locations for the variables. Likewise, the lexical function and macro binding special forms (macrolet, flet and labels) will use a nested compiler that has altered name lookup rules. This recursive process is elegant and matches the structure of the code itself.

When all the arguments in a list have been evaluated and the first position contains a name that is not a special form, the compiler will emit a function call primitive. Depending on the name lookup rules in place, this may be a direct call to a function object, or a late-bound, name-based call as described in Section 3.3.1.

Compiler Macros and Symbol Macros

The description in the previous section assumes that the programmer has not installed any symbol or compiler macros, two concepts that Gozer includes from Common Lisp. Symbol macros can be installed by the programmer to change the meaning of a bare symbol (i.e., not in the function-application place in a list). Instead of being taken as a variable reference or a constant, the symbol instead expands into an arbitrary S-expression that is then compiled in place of the original symbol. This capability is typically used to provide shorthand notation for commonly repeated actions, is usually used with a normal macro, and is limited to lexical scope. An example is the CLOS-inspired macro with-slots for quick access to the properties of an object. Therefore, when a bare symbol occurs, the compiler may need to execute programmer provided code before continuing.

Compiler macros apply to any function or macro application and are associated with the name in the application place. Although they can have many uses [24], in Gozer they are most often used to provide optimization advice to the compiler, at the S-expression function or macro optimization level. Due to the nature of the Gozer language and libraries, it is often easier to implement such optimizations in Gozer itself rather than in Java. For example, Listing 3.7 shows the compiler macro for transforming the (relatively expensive) functional iteration method mapcar into a simple loop when the iteration function is a compile-time constant. Again, this means that at any function or macro application site the compiler may be required to execute programmer provided code before continuing. Compiler macros and symbol macros that cause the compiler to run programmer provided code can be likened to Java's annotation processors.

Listing 3.7: Mapcar Simplification Compiler Macro

```
;;; Iterations over a constant number of lists can be optimized
;;; This example shows only 1 list and only literal lambda expressions
(define-compiler-macro mapcar (&whole w fn &rest lists)
  (if (= 1 (length lists))
      ;; replace with the equivalent dolist form.
     (cond
       ;; literal lambda expressions can be inlined.
       ;; (mapcar (lambda (x) ...) list) ⇒
       ;;   (let ((%res (list))) (dolist (x list) (append! %res ...)))
       ((%lambdap fn)
         `(let ((%res (list)))
            (dolist (,(first (second fn)) ,(first lists))
              (append! %res (progn ,@(cdr (cdr fn)))))
            %res))
        ...)
      w))
```

Code Analysis for Warnings and Optimizations

The compiler has a few other responsibilities, including emitting warnings, tracking types, and optimizing the generated assembly. These tasks require the compiler to have a certain level of semantic understanding of the source code, as well as to interact closely with the GVM and the state of the Gozer image.

The programmer can ask the compiler to emit warnings with the declaim special form. Warnings can be emitted for suspected type violations, invocations of Java classes, properties or references that might not be satisfied, application of functions that are not known to be defined, and usage of deprecated features. Both Gozer-level deprecations and Java-level deprecations are respected. When a warning is emitted, it is done using the standard condition system, meaning that the programmer is able to capture specific warnings (or all warnings) and take action as desired.

The compiler performs a limited version of type inference and propagation. This information is used primarily to output diagnostics and warnings to the programmer and for generating documentation. The compiler can become aware of type information in three ways: a compile-time constant has a known type, a non-ambiguous

Listing 3.8: Type Inference Examples

```
;; Functions of defined type
(- 1)              ⇒ Number
(+ 1 2 (* 3 4)) ⇒ Number

;; Un-ambiguous Java methods
(java.lang::Thread.holdsLock 1) ⇒ Boolean

;; A variable whose value begins as a constant
(let ((x 1.2)) x) ⇒ Double
;; Assignments propagate the type
(let ((y 1.2) x)
  (setq x y)
  x)               ⇒ Double
;; Then unify to a general number following a function
;; that returns only a number
(let ((x 1.2))
  (setq x (1+ x))
  x)               ⇒ Number
;; Finally, after an assignment whose only common type
;; is Object, that becomes the inferred type
(let ((x 1.2))
  (setq x 'foo)
  x)               ⇒ Object
```

Java method is invoked on an object of a known type, or the programmer supplies type information (for a variable or the result of a function) using declare. If the programmer supplies variable type declarations, and the compiler can prove the declarations are violated (with rudimentary linear flow analysis), an error is emitted; otherwise, extra runtime checks are inserted using the `CHECK_VAR` instruction. Listing 3.8 shows some expressions and the resulting type the compiler is able to infer.

The Gozer compiler performs only the simplest of analyses and cannot be truly considered an optimizing compiler. Nonetheless, there are certain differences in the final generated assembly that depend on the optimization preferences in effect. Thees differences may arise as code is being compiled into assembly, or during a final peephole optimization pass at the end of compilation. Two of Common Lisp's four standard optimization axes are supported, execution speed and safety, each taking on values between 0 (low) and 3 (high). These axes may be influenced locally with declare or more globally with declaim.

When low safety levels are requested, the `CHECK_VAR` instruction is omitted, as is the instruction that checks that all arguments are consumed by a function call (as mentioned in Section 3.2). Higher safety levels result in the *addition* of instructions to explicitly clear references to local variables when exiting a lexical scope for improved garbage collection and process migration.

When higher execution speed is requested, the if special form (to which most control-flow macros ultimately expand) performs dead-code elimination and con omit jumps. The usage of an array for local lexical variables is a more important optimization enabled by higher safety levels, as is the combination of multiple argument binding instructions into the single `LL_POP_AND_BIND` instruction. The combination of two sequential accesses to lexical variables into a single `PUSH_LOCAL_VAR` instruction is also enabled at higher optimizations and can improve function call speed. The first two of these optimizations are performed during assembly generation, while the last two are performed during peephole optimization.

This concludes the discussion of the Gozer language and implementation. The next chapter covers the primary application of the Gozer language, its use in the Gozer Workflow System.

Chapter 4

Workflows

Although Gozer performs well as a general purpose scripting language, its intended use case is the implementation of complex business processes called workflows. The component of the Gozer Workflow System that provides workflow functionality is called the Vinz module, and it runs within the “BlueBox” distributed computing environment. It is the goal of Vinz to make it easy to implement these complex business processes, and to make it easy to monitor and manage them in operation while also being robust and capable of handling partial systems failures.

There is great diversity in the workflows that Vinz is required to support. Workflows may be either long running (hours or days) batch processes or quick interactive processes (seconds). They may be initiated by external events or repeated on a schedule. Workflows may accept thousands of input files and produce gigabytes of results or accept only one input and produce only one result. In all of these cases, it is the responsibility of the workflow system to make the best possible use of the computing cluster’s resources, keeping in mind that the workflows are not the only resource consumers.

Listing 4.1: Sums-of-Squares Variants

```
(defun loc-sum-squares (numbers)
  (apply #'+
         (loop for number in numbers
           collect (* number number))))

(defun par-sum-squares (numbers)
  (apply #'+
         (loop for number in numbers
           collect (future (* number number)))))

(defun dist-sum-squares (numbers)
  (apply #'+
         (for-each (number in numbers)
           (* number number))))
```

4.1 Overview

Gozer's distribution facilities, and in general its integration with the BlueBox platform, are provided in a module known as Vinz. Vinz offers a simplified set of abstractions to workflow authors intended to make writing fully distributed, concurrent workflows as similar to writing local, sequential programs as possible. As with local parallelism, opportunities for distribution are written in a declarative fashion, and the details of implementation are provided by the platform.

Listing 4.1 shows example functions for computing the sum of squares in map/reduce style. The function loc-sum-squares performs the computation entirely locally using Gozer's sequential loop construct for the map step (squaring the numbers) and a sequential application of addition for the reduce step.

The function par-sum-squares allows for a degree of local parallelism in the map step by using the GVM-supported future construct, as described in Section 2.3.6. However, shared-memory thread-based local parallelism has a number of disadvantages for the construction of large, complex, evolving processes. First, it doesn't scale well beyond a single physical machine. The potential for side-effects makes it challenging to evolve a local process over time or to integrate code from multiple authors. Any robustness in the case of machine failure such as the saving and resuming of intermediate states must be programmed explicitly for each process. Finally, especially in the case of long running processes, it may be desirable to "suspend" a process in order to allow a higher-priority process to use scarce resources such as memory; such suspension would similarly require explicit handling in the design of each process. Gozer's distributed workflows automatically overcome these difficulties by allowing a single process to span multiple machines, by using a fork/join paradigm that limits or prohibits side-effects, by automatically creating and maintaining persistent checkpoints, and by using non-blocking, zero-resource consuming, event-driven processing.

The function dist-sum-squares demonstrates the simplicity and advantages of distributed programming with Vinz. Choosing an appropriate level of concurrency, distributing work to available nodes, gathering results, and continuing the computation when all concurrent work is completed are all automatically handled by Vinz. Importantly, waiting for computations to complete is event driven and consumes no resources. Even though conceptually dist-sum-squares blocks awaiting the for-each results, no actual blocking occurs because the (distributed) process state is saved to persistent storage and is restored only after all necessary results are available. This type of distribution may be nested to an arbitrary depth and the results of each step may be arbitrarily complex.

4.1.1 Tasks and Fibers

A running instance of a particular Vinz workflow is called a *task*. Every task contains one or more *fibers*. A task is to an operating system process as a fiber is to a thread. A fiber represents a Gozer flow of control that may be advancing in only a single Java thread in the distributed environment at any given time. Fibers may be halted at any point permitted by the GVM. When halted, a fiber is saved to persistent

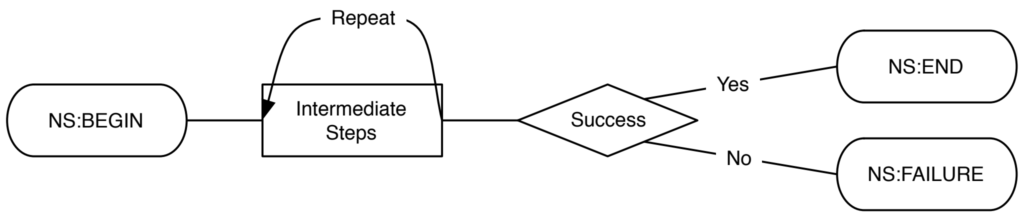


Figure 4.1: Process Lifetime

storage, and it can resumed at some later date, possibly on a different machine or within a different Java Virtual Machine.

Every fiber is uniquely identified by a ProcessID. A fiber may have one parent and zero or more children. A fiber is in some state starting with NS:BEGIN and ultimately terminating with either NS:END or NS:FAILURE; other possible states between these two are user defined (see Figure 4.1).

An external entity called the NotificationService is responsible for creating the unique ProcessIDs identifying fibers (and other *processes*, all of which start in the NS:BEGIN state and terminate in either the NS:END or NS:FAILURE state), and tracking their states. When a process terminates, the NotificationService may inform other entities of this fact.

The for-each and parallel macros described in Section 4.2.4 create and manage fibers automatically. The relatively recent fork-and-exec and join-process forms of Section 4.2.4 allow advanced users to manually create and wait for their own fibers.

4.1.2 BlueBox

Vinz was implemented within the existing BlueBox distributed system and service framework, and this system's properties influenced much of the design and implementation of Vinz. This service framework is an implementation of a service-oriented architecture, or SOA, intended to support both interactive usage and batch processing. The goal of an SOA is to achieve modularity and scalability in the system by dividing responsibilities among well-defined *services*. Each service publishes and adheres to an interface that defines the set of related *operations* the service provides. The service is invoked by *clients* who provide the input (request) to the operation and process its results (response). The implementation of a service is free to evolve as long as the interface remains compatible with existing clients. To allow for redundancy and load balancing, there may be many running instances of a service spread across some number of the *nodes* that make up the distributed system (cluster). These instances all provide the same interface; thus, service clients need not be concerned with which instance processes any given operation.

In the BlueBox SOA, clients and services are logically disconnected from each other. They interact by sending well-defined *messages* (even when physically running within the same JVM). These messages take the form of an XML document that adheres to the description in the service's interface definition, itself also an XML document known as a WSDL. The developer of the service provides an implementation of each operation defined in the interface and provides metadata with the service allowing the service framework to route incoming requests to the desired implementation for processing.

Conceptually in BlueBox, a client places a request message on the *enterprise service bus*, which is then responsible for delivering the message to some instance of the desired service, or even buffering the message if no instance is currently available to handle the request, due to high load or system downtime. For BlueBox, the service bus is an implementation of the Java Message Service (JMS) message queue; the two terms are used interchangeably. The service bus load balances among all instances by keeping track of the number of requests that are in progress within any given JVM and not allowing that number to exceed preset limits, and by giving precedence to higher-priority requests (e.g., interactive requests have a higher priority than batch requests). The service bus is also responsible for handling the failure of an instance when processing a request, transparently re-directing the request to another available instance. Once some instance has processed the request, the response is also placed on the service bus for delivery to the calling client. The BlueBox service framework handles the details of interacting with the service bus, including the creation and addressing of messages. The service is the primary unit of deployment, configuration, and operational management within the BlueBox service framework, but individual operations within a service may be monitored and configured for load balancing purposes as well.

4.2 Distribution

A distributed workflow is written as a Gozer program (or script) which may make requests of BlueBox services or execute Java or Gozer code and which may contain constructs intended to take advantage of distributed parallelism. It is the job of Vinz to take this program and make it possible to run it on the nodes of the BlueBox cluster, automatically handling all the details related to distribution and robustness. This section describes how this is accomplished, beginning with how Gozer programs can be deployed and run on the BlueBox service framework before moving on to specific aspects of workflows.

4.2.1 Workflow Services

The BlueBox service framework (and much of its supporting infrastructure) operate at the level of the service. It is the service that is deployed to the Bluebox nodes, and it is only through the operations provided in a service's interface that clients can interact with services, or services can interact with other services. It is natural, then, that the way in which a Gozer program is presented to the BlueBox service framework is as a service. Such a service is called a *workflow service*, and although the service author can provide his own operations (and their implementations) as

Operation	Description
Start	Asynchronously begin execution of a workflow, immediately returning its id. Used for batch processing.
Run	Synchronously execute a workflow, returning its id when complete. Used for interactive processing where the workflow's side-effects matter most.
Call	Synchronously execute a workflow, returning its last result. Used to treat an entire workflow as a value-returning service operation.
Terminate	Management operation to asynchronsly terminate any running workflow.
RunFiber	Begin execution of a portion of the workflow on this instance.
AwakeFiber	Resume a suspended parent fiber when a child fiber has completed.
ResumeFromCall	Resume a suspended fiber when a remote operation completes.
JoinProcess	Resume a suspended fiber when any arbitrary process has completed.

Table 4.1: Vinz Service Operations

with any other service, Vinz always provides a standardized set of operations that are used to manage the functionality of distributed workflows. These operations are enumerated in table 4.1. The implementation code for these operations is the same across all workflow services (in fact, it is contained in an extension to the BlueBox service framework itself); the only thing that differs is the Gozer program.[1]

Execution of a workflow is commonly initiated by invoking the Start operation with a set of workflow-defined parameters. This causes the creation of a task and its corresponding ProcessID, which uniquely identifies that particular running instance of the workflow (a particular workflow service may have many outstanding tasks at any one time). Every task contains one or more uniquely identified fibers (initially one). As mentioned previously, a fiber encapsulates a Gozer flow of control that may be advancing on only a single node at any given time. A task is somewhat

[1]Combining this fact with the `eval` capability of the Gozer language and the ability to provide parameters to a workflow service leads to the interesting ability to define and execute new distributed workflows on the fly simply by writing Gozer scripts and using that as input to a workflow service which `eval`s them. This is useful for testing purposes, but is not used in production because it complicates monitoring.

analogous to an operating system process, while a fiber is analogous to a thread within that process.

Once Start has created a task and fiber, it prepares the environment in which the main fiber will execute, and, with the support of the GVM, saves its initial continuation (state) to persistent storage. The Start operation then issues an asynchronous invocation of the RunFiber operation with a parameter identifying the newly created fiber. Its job complete, Start now returns the task's ID to the caller, often (indirectly) the system operator, who can use it to monitor the progress of the task. The Run and Call operations proceed similarly, postponing the returning of results until the entire workflow task is complete, thus allowing the caller to treat a workflow as a synchronous service invocation.

When the message queue delivers a RunFiber request to an instance of a Vinz workflow service, the fiber's continuation is loaded from persistent storage, and the GVM begins executing that continuation. The GVM continues to run until the program has been completed, or until the next continuation is requested, at which point the fiber is halted and its state stored for later execution. At this time, continuations are always implicitly or explicitly requested by the Gozer program, but they could also be requested by the system operators for management purposes.

While running, a fiber can create and execute (via RunFiber) other fibers. These fibers are *children* of the first fiber. The for-each and parallel macros described in Section 4.2.4 create and manage fibers automatically. The fork-and-exec and join-process forms of Section 4.2.4 allow advanced programmers to manually create and wait for their own fibers.

4.2.2 Non-Blocking Service Requests

In practice, Vinz workflows largely consist of requests to other BlueBox services. In a traditional synchronous service invocation, the sender is blocked until its request

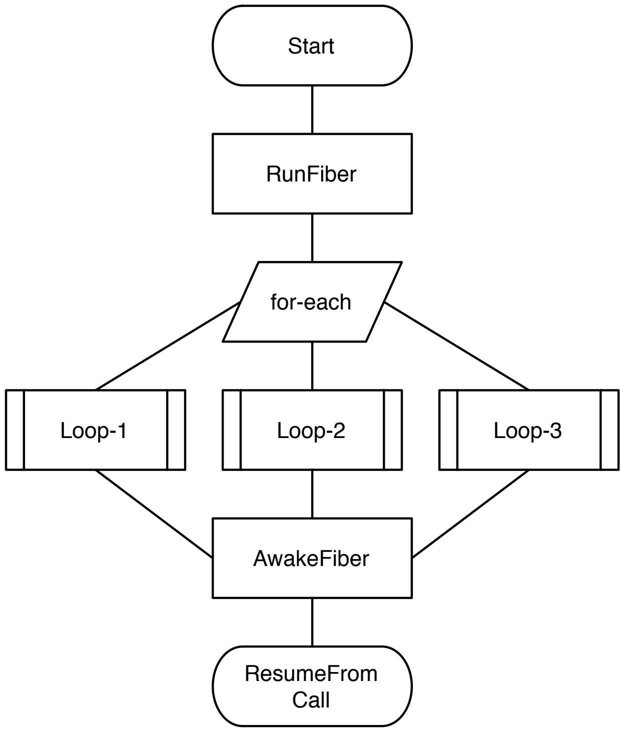


Figure 4.2: Sample Workflow Lifetime

has been delivered by the message queue, the service has completed processing, and the results are returned by the message queue. During this time, the sender is consuming resources (physical memory and a BlueBox request "slot") without making any progress; in essence, those resources are being wasted. An ordinary asynchronous request does no better unless the sender can find other work to do, a situation that is complex to achieve.

Vinz workflows address the problem of wasted resources by automatically executing a Gozer yield statement when a fiber makes a service request. The request message is sent asynchronously, and the message queue is instructed to deliver the response to any workflow service instance by means of its ResumeFromCall operation, not necessarily to the sending thread or instance (as with a traditional approach). While the service request is being delivered and processed, the sending fiber's state is saved to persistent storage. Later, when the response message is available and delivered to some service instance's ResumeFromCall operation, the fiber's state is restored and processing is resumed in that instance.

Overall, this allows many more tasks to be in progress at any one time. Wall-clock time, CPU resources and memory that would otherwise have been wasted blocking can now be used by a different task to make progress. Because the message queue is constantly load balancing in-progress requests and making decisions based on priority, this also helps to improve interactive usage (interactive requests are less likely to be held up by individual long-running batch workflows). In addition, together with the entire state of the task being regularly persisted to stable storage and the message queue providing buffering and re-delivery of messages in case of instance failure, this makes for a highly robust system: one in which the failure of any instance will result in minimal delays as other instances automatically compensate.

There are some cases where the service operation is expected to execute very quickly, and therefore the relative overhead of the aforementioned task migration might be considerable. In these cases, the programmer can, statically or dynamically, choose to require Vinz to make a traditional synchronous request. Another situation that calls for the synchronous approach is when a service request is attempted from a future's background processing thread. It's generally not possible for process migration to occur for a single future (what state should be persisted? what about other futures, or the main fiber thread?) so Vinz detects this and automatically makes a traditional synchronous request from that thread.

4.2.3 Deflink

Vinz takes advantage of the dynamic code-generation features of Gozer macros and the published service interfaces to make it easy for a workflow to interact with any service. A macro called deflink provides this functionality. This macro requests a service's interface in the form of an XML document, parses it, and then generates a set of functions to invoke each operation the service publishes, together with the appropriate placement of yield statements to make the request non-blocking. The XML message structure is flattened into a set of parameters for the function, and the function is capable of coping with complex XML trees by using corresponding Gozer data structures.

As part of the source code for the workflow, each deflink is evaluated when the source code for the workflow is loaded. This ensures that the functions generated will be appropriate for the service version currently operating, which provides protection against minor version incompatibilities. If for some reason an operation cannot be interacted with from a Gozer function, deflink instead generates a Gozer macro that signals an error. In this way, if and only if the workflow tried to invoke that operation, a compile-time error will occur and the workflow will not be loaded, thus avoiding runtime errors.

Listing 4.2 shows an example invocation of the deflink macro, and some of the generated functions (edited for size). Notice that the documentation specified in the interface document is preserved for the Gozer programmer. The function SM-ListSessions-Method provides the high-level interface using named keyword function arguments. The function SM-ListSessions actually invokes the service, handling the case of background threads as well as using Gozer's condition system to provide optional restarts in the event of error (see Section 4.2.5 for more on error handling).

4.2.4 Forking Fibers

As discussed earlier, the fundamental unit of computation in a Vinz workflow is a fiber. A fiber can be running at most once on one node in the BlueBox environment, and every workflow begins with a single fiber created by the Start operation. Therefore, in order for workflow processing to take advantage of multiple nodes in the cluster, multiple fibers must be created.

Listing 4.2: Deflink

```
(deflink SM :wsdl "urn:security-manager-service"
            :port "SecurityManager") ⇒

(defun SM-ListSessions-Method
   (&key FilterParams WithinRealm)
 "Returns a list of sessions visible to the..."
 (let ((msg (create-message "SM-ListSessions")))
   (. msg (set "FilterParams" FilterParams))
   (. msg (set "WithinRealm" WithinRealm))
   (SM-ListSessions :message msg)))

(defun SM-ListSessions (&key message)
   "Returns a list of sessions visible to the..."
  (restart-case
      (let ((response
             (cond
               ((%is-fiber-thread)
                (call-wsdl-operation-async
                  :soap-action "...:ListSessions"
                  :message message)
                (yield))
               (otherwise
                (call-wsdl-operation
                  :soap-action "...:ListSessions"
                  :message message)))))
        (parse-wsdl-response response))
    (ignore () (log "Ignoring an exception"))
    (retry  () (SM-ListSessions :message message))))
```

Fiber creation requires a parent fiber and a user provided function (often an anonymous lexical closure is used to pass information to the new fiber). This parent fiber and all of its state is first cloned to create a new child fiber. The child fiber's Gozer function call stack is replaced with a call stack invoking the user provided function, and it receives a new ProcessID. (These operations are combined in the primitive function fork-and-exec.) Finally, the newly created child fiber is scheduled for execution by placing a RunFiber request on the message queue.

Fiber creation is somewhat analogous to Unix process creation with the fork system call. Although the programmer visible values, objects, and variables in the child fiber start out equivalent to those of the parent, subsequent changes by either fiber will not be visible to its counterpart. This eliminates any burden of synchronization when mutating variables and values, thus simplifying the programming model both for the workflow author and the Vinz implementation, and it drastically reduces the amount of distributed coordination that Vinz must perform.

Workflow authors have direct access to the ability to create new fibers through the fork-and-exec function, which returns to the parent fiber the ProcessID of the child fiber, while executing a supplied function in the child. A fiber can wait for any other fiber to terminate using the join-process function (analogous to the Unix wait function). A fiber that calls join-process ultimately invokes yield and so relinquishes its resources until such time that the requested fiber terminates. If called from a future's background processing thread, join-process only suspends that thread, leaving the rest of the fiber unaffected.

For-Each and Parallel

The fork-and-exec function, together with the join-process function are enough to implement many useful distribution strategies. However, they are very low-level and as such may be error-prone. Vinz provides two macros that are conceptually

Listing 4.3: Distribution Related Vinz Forms

```
(defun fork-and-exec (function &key fiber-name argument)
  "Runs function in a new fiber, returning its ProcessID")

(defun join-process (process-id)
  "Causes this fiber to block until the fiber at process-id has completed")

(defmacro for-each ((item 'in items &key test name chunk) &body body)
  "Executes the expressions in body with the variable at
   item bound to each subsequent value returned from items
   (that pass the test, if given), each in a new fiber. Returns
   a list of the last value generated by each iteration.")

(defmacro parallel (&body body)
  "Runs each expression in body in a new fiber. Returns
   a list of the last value generated by each expression.")
```

layered on top of these functions that capture commonly used distribution patterns.[2]

The most commonly used of these macros is for-each, which implements the map step of the map/reduce paradigm. Listing 4.1 provides an example of using for-each. This macro takes as input a sequence of values, and for each value in that sequence, executes the same body of statements. The results of these executions are collected and returned to the parent fiber. The parent fiber was "blocked" (as with yield) until all the executions were complete. The for-each macro completely abstracts away the operations involved in setting up concurrent fibers and awaiting their completion.

Optionally, for-each may take additional arguments that modify its behaviour. The :test argument, if provided, specifies a function predicate that will be applied to each item. Only those items for which this predicate returns a true result will be given to a fiber, allowing the programmer to dynamically filter the sequence. The :name argument, if provided, can either be a string or a unary function that returns a string, and is used to allow the programmer to provide meaningful names for the fibers. Finally, the :chunk argument, if provided and non-nil, allows the

[2]Actually, these two macros, for-each and parallel, were implemented before the general fork-and-exec support was available. Originally, they had to be implemented as complex Gozer primitives (special forms) but over time the model has been generalized enough that now they can just be simple macros while fork-and-exec is a function.

Listing 4.4: Vinz Spawn Limit

```
(let ((parent-pid (get-process-id))
      (children   (list))
      (func       (lambda (number)
                    (* number number)
                    (awake parent-pid))))
  (append! children (fork-and-exec func :argument 1))
  (append! children (fork-and-exec func :argument 2))
  (append! children (fork-and-exec func :argument 3))
  (yield)
  (append! children (fork-and-exec func :argument 4))
  (yield)
  (append! children (fork-and-exec func :argument 5))
  (yield)
  (yield)
  (yield)
  (collect-child-results child-pids))
```

programmer to control the "chunking" of items from the sequence. Instead of using one fiber for every item, items are grouped together into a chunk and processed by a single fiber. The programmer may specify that items are to be processed in parallel (using futures) or sequentially within a chunk, and, in the sequential case, whether the first unhandled condition will halt the fiber or whether all items should be processed. For items that will be processed with very short duration, chunking can be more efficient by reducing fiber-management overhead, but it also potentially reduces the robustness of the process.

Less frequently used is the parallel macro. This macro simply executes each of the forms in its body as new fibers. The result of each form is collected and returned to the parent fiber (which was again blocked).

If for-each or parallel is used from a background thread, it cannot yield the fiber for the same reasons that a non-blocking service call cannot. The solution in this case is to have the background thread fork a new fiber which in turn executes the for-each or parallel code. The background thread synchronously (without yielding) awaits the termination of this fiber.

The for-each macro in particular may result in an arbitrarily large number of new fibers (a number equal to the number of values in the sequence) and their corresponding RunFiber requests on the service bus. In order for cluster resources to be shared among workflows in the desired way (**not** necessarily fairly), these macros introduce a configurable throttling mechanism. Called the *spawn limit* this control prevents any individual macro invocation from resulting in more than the configured number of concurrently executing fibers at one time. The spawn limit may be dynamically adjusted by the workflow. Listing 4.4 shows a simplified example[3] of what the macro in Listing 4.1 might expand as given the numbers from one to five, if the spawn limit was three. The total number of yield forms will be equal to the number of child fibers created, but their distribution will differ depending on the spawn limit.

Listing 4.4 shows that the parent fiber must wait for each child fiber to awaken it before moving on. The child fiber awakens the parent fiber by placing a message for AwakeFiber on the service bus. This is more efficient than having the parent fiber invoke join-process for each child fiber created since the child fibers may finish in any order and in a for-each the order does not matter; only that the total number of yield operations match the total number of AwakeFiber messages matters. While this is simple and robust, it is also a fairly heavy-handed way of achieving parent/child communication, however, and can introduce artificial bottlenecks and very bursty behaviour.

Although the spawn limit addresses an operational problem, its implementation currently is sub-optimal. Consider the case where the spawn limit is absent or very high relative to the number of workflow service instances available, n, and suppose that each child fiber is going to execute in approximately the same amount of time.

[3]In particular, awaking parent fibers is not performed by a function call the child fiber executes. For reliability, it's actually a property of the fiber itself. The fibers created by fork-and-exec do not notify their parent of termination, but the fibers created by these macros do.

Initially, n child fibers will be executing concurrently. When they finish, n AwakeFiber messages will be placed on the message queue and delivered for execution. A fiber can be executing on at most one instance at a time, so $n - 1$ of those AwakeFiber operations will be forced to wait while a single instance reads and updates the persistence information. Each AwakeFiber instance will proceed in turn, but for some period of time all n instances will be unavailable to process other activity such as other RunFiber requests (and because instances are often shared across services, even unrelated service operations may be blocked, something that Vinz seeks to avoid). To partially counteract this problem, AwakeFiber requests are specified to be low-priority, and a running AwakeFiber places a strict limit on how long it will wait for its turn to execute the fiber before giving up and placing itself back on the message queue for later delivery.

When the spawn limit is low (and especially if the spawn limit is low but the number of child fibers is high), the overhead of sending an AwakeFiber message for permission to spawn the next child can be high. It would be better if, as the child fiber terminated, it could simply spawn whatever sibling fiber is next without involving the parent. The main difficulty here is synchronizing access to the parent so that it continues only when all of its children ultimately complete. Further work is needed in this area.

Task Variables

Because fibers are cloned copies of their parents, side effects like variable or value mutation are not visible between fibers. Child fibers created with the macros discussed in the previous section communicate to their parents by their return values in a very functional fashion, which is a good fit for Gozer's semi-functional nature. In some cases, though, a distributed algorithm can be greatly simplified if some mutable values could be globally shared between all fibers.

Listing 4.5: Using A Task Variable

```
(deftaskvar exit-flag
  "A global flag. When this becomes true, stop.")

(defun dist-sum-squares (numbers)
  (for-each (number in numbers)
    (unless ^exit-flag^
      ;; don't do anything if the flag is set
      (if (= -1 number)
          ;; tell everybody to stop working!
          (setf ^exit-flag^ t)
          (* number number)))))
```

Vinz supports this mutable sharing through a concept know as task variables. A task variable is declared at the top-level of the workflow program with the deftaskvar macro, similar to the basic Gozer defvar macro for defining global variables. All fibers within a task of that workflow may access and update that variable. Vinz guarantees that each fiber will see a self-consistent value for that variable and will always see the latest value for that variable. Stronger promises such as access order or atomic read-modify-update sequences are not provided.

Gozer global variables are conventionally given names that start and end with the * character ("earmuffs"). Similarly, Vinz task variables are given names that start and end with the ^ character, although this is a requirement, not just a convention. In order to provide the desired semantics for task variable access, Vinz needs to replace each read or write of the variable with a function call that performs the steps of checking for a stale local cache, reading the most recent value from the persistence store, taking out appropriate locks, etc. Vinz does this by hooking into the Gozer source parser (the reader) using a reader macro defined on the ^ character (see Listing 4.6). Each occurrence of a task variable such as `^exit-flag^` in the source file is read as if it were the form `(%get-task-var '^exit-flag^)`.

Listing 4.6: Task Variable Reader Macro

```
(set-macro-character
 #\^
 (lambda (the-stream c)
   (declare (ignore c))
   ;; ^foo^ -> (%get-task-var 'foo)
   (let ((var-name (read the-stream t nil t))
         (var-str  (symbol-name var-name)))
     (unless (. var-str (endsWith "^"))
       (error "Task vars must be wrapped in ^"))
     ;; strip the closing ^ so that we only wind up with one symbol that
     ;; we need to export, the same thing given to deftaskvar. Make sure
     ;; it's in the same package as the original
    `(%get-task-var ',(intern (subseq var-str 0 (1- (length var-str)))
                               (symbol-package var-name)))))
 t )
```

4.2.5 Error Handling

An important part of a robust system is error handling, or, more generally, condition handling. Vinz provides workflow authors with some convenient extensions built on Gozer's very general condition system, described in Section 2.4.[4]

At the core of these extensions is the concept of a named *handler* which is created by the macro defhandler and utilized by the with-handler macro (Listing 4.7). A handler associates a list of conditions (whether Java classes or QNames) with an action (usually) provided by Vinz, making it possible to centralize condition-handling logic. Instead of repeating the list of conditions every time the programmer wants to take a certain action to handle a condition (in handler-bind forms spread throughout the program), the programmer can define a handler once and use it repeatedly in with-handler.

Vinz provides four actions (an action is just a function, so the workflow author is free to define additional actions). Two actions, retry and ignore, just invoke an active restart of the same name. The functions created by deflink bind these restarts,

[4]Before Gozer's condition system was fully implemented, this same functionality had to be provided with custom code.

Listing 4.7: Vinz Error Handling

```
(defhandler ignore-handler
  :java ("java.lang.Throwable")
  :action ignore)

(defhandler retry-handler
  :java ("java.net.SocketException")
  :code ("{urn:service}Connet"
         "{urn:service}Transmit")
  :action retry
  :count 5)

(with-handler ignore-handler
  (with-handler retry-handler
    (optional-socket-operation)))
```

and the programmer can also bind them. The retry restart is intended to be used to deal with possibly transient errors such as network connectivity failures without the programmer being forced to write an explicit loop. Ignoring a condition can be used to allow "optional" operations such as generating debugging data to fail without impacting the workflow.

The remaining actions are break and terminate. These actions interact with Vinz fibers and tasks. The former action is named for the Java keyword of the same name. Intended to be used around a for-each distributed loop, the break action causes the currently executing fiber to immediately terminate cleanly and return nil to the parent fiber. In contrast, terminate terminates both the current fiber and the entire task with an error status. Any other fibers that are currently running or are queued by the service bus will notice that the task has terminated in short order and also terminate in error.

Both local and distributed conditions can be handled with these extensions. When a function created by deflink invokes a service, the response from the service might be an error, conveniently expressed as an XML QName. The function arranges for this QName to be signaled as an error, thus integrating distributed error conditions into Vinz handling.

4.2.6 Implementation

Futures And Continuations

Continuations are not natively supported by the JVM (there is no way to capture a JVM call stack and re-enter it later). This implies that the JVM's operand stack and function calling operations could not be directly used (without some sort of exception-based transformation that may introduce its own limitations [39]). Instead, the GVM implements its own stack-oriented architecture, in many ways similar to the JVM's architecture. The GVM call stack consists of ordinary Java objects representing function calls together with arguments and local variables. These objects are used to create the continuations requested by yield. This is similar to the approach taken by Stackless Python [40] and SISC Scheme [28]. Compilation to bytecode (as opposed to a tree-walking interpreter) was introduced primarily as an optimization for Vinz persistence.

In contrast with continuations, the JVM does provide the concept of futures through its ExecutorService. The challenge for the GVM here was to make them transparent to the programmer, completely managing their execution and determination, while allowing Vinz to integrate them into its distributed workflows (it is problematic to migrate a fiber from one machine to another while some of its futures were still running on the first machine). To do this, the GVM adopts the rule that passing any future to a Java library or a BlueBox service will cause that future to be determined. In addition, when capturing a continuation, futures referenced from that continuation are determined (the continuation doesn't become available until all futures have completed). Finally, the BlueBox platform provides an ExecutorService that integrates with its native load balancing heuristics, and Vinz configures futures to be created using this implementation.

Workflow Distribution

The BlueBox service framework provided most of the tools required to build distributed workflows: an SOA architecture, access to the defined interfaces of services, a service bus with individually addressable services, a global process tracking service, and so on. Only a few additional features were required, as described here.

One clear need was a way to persist a fiber's state and data so that one instance could write it, and another instance could later read it and resume execution. The Java platform defines a built-in way to externalize in-memory objects called serialization, and so building on this support was the obvious approach. Vinz thus writes a fiber's state and data using Java serialization, with many customizations for efficiency and to broaden what can be successfully serialized. One instance is able to access the data persisted by a different instance because all persisted data is written to files on a shared NFS filesystem; the introduction of what might look like a single point of failure into a distributed system was averted by utilizing very highly-available (and highly-expensive) NFS servers.

Much time was spent optimizing Vinz serialization for performance. A series of tests determined that compressing the serialized data before writing it to NFS was a net win by reducing IO costs considerably, even though the Java serialization format is computationally expensive to compress with standard deflate-based compression techniques. It was also discovered that plain deflate can be made to perform approximately 30% better than the more robust and space-efficient gzip format for this data. Analysis of serialized data resulted in the introduction of a custom serialization format that stored the most commonly serialized objects more efficiently. Even after all this, reconstituting a fiber from its persisted state is still relatively slow and so a cache of recently seen fibers is maintained in the memory of each instance. Vinz executes no control over where a fiber will be asked to run however, leaving that decision in the hands of the message queue. As a result, the cache is only approximately 18% effective for mutable data (for immutable data, the cache is up to 66% effective). Further work is needed in this area (current cache effectiveness actually decreases as the cluster incorporates more nodes), perhaps by devising a way to move the processing work to the last location of the data as is done, for example, in the Swarm system [6].

Some way to prevent the same fiber from being run by different JVMs at the same time was a key need (for example, in the AwakeFiber case discussed above). Within a single JVM, the usual thread locks suffice for this, but distributed locks would be required. Because the persistence information was being shared using an NFS filesystem, the natural choice was to use file locks on the NFS files. This was simple and mostly effective, but completely opaque, and coping with quirks of the various implementations of NFS in use required a great deal of code, and prohibited some performance optimizations. To remedy this, a custom distributed lock implementation based on the Apache ZooKeeper distributed coordination system[5] has been developed, completely replacing all NFS lock usage. This same locking and persistence infrastructure is used to implement task variables.

The enterprise service bus is currently completely responsible for load balancing and prioritizing messages. Each task starts and runs to completion independently of any other tasks that may be operating, subject only to the capacity limits of the system. In effect, task scheduling is first-come-first-serve, which has been shown to be suboptimal [38]. Work is in progress to develop more efficient and proactive scheduling policies utilizing historical and current information about the state of the entire cluster (shared using ZooKeeper) and tasks in process based on research presented in [15].

[5]Available from `http://hadoop.apache.org/zookeeper/`.

4.3 Workflow Patterns

In the same way that software design patterns have been developed for general software engineering, specialized workflow patterns have been developed for the discussion of common problems and solutions in workflow applications. One well-known set of patterns has been developed by a the Eindhoven University of Technology working with the Queensland University of Technology. Beginning with a set of 20 control-flow patterns published in 2003 [41], this work has expanded over time to include more advanced and revised control-flow constructs [33], data patterns [34], resource representation patterns [36], and exception or error handling patterns [35]. This group has also evaluated many commercial and freely available workflow engines with respect to their capabilities in this system of workflow patterns. The Gozer Workflow System (GWS) is not intended to implement every possible pattern (indeed, some are mutually exclusive), but the work of this group has been useful as a standard comparison and a source of possibilities.

The original 20 control-flow patterns are divided into five groups (these groups have been much revised and extended in subsequent work): Basic, Advanced Branching and Synchronization, Multiple Instance, State Based and Cancellation. Being built on top of a complete language has advantages for the control-flow patterns, and the Gozer Workflow System directly implements 14 of the 20, while providing enough support for workflow programmers to implement at least three of the remaining six. This is detailed in Table 4.2. (In general, the more shared state or synchronization is required, the harder it is to implement a control pattern in the GWS.)

The 40 data patterns of [34] have to do with data internal to the workflow as well as data stored externally and are divided into four groups: Data Visibility, Data Interaction, Data Transfer and Data-Based routing. Again, being built on a language

	Pattern	Status
	Basic	
1	Sequence	Implemented
2	Parallel Split	Implemented
3	Synchronization	Implemented
4	Explicit Choice	Implemented
5	Simple Merge	Implemented
	Advanced Branching and Synchronization	
6	Multi-Choice	Implemented
7	Structured Synchronizing Merge	Implemented
8	Multi-Merge	Implemented
9	Structured Discriminator	Unsupported
	Multiple Instance	
12	Multiple Instances without Synchronization	Implemented
13	Multiple Instances with a Priori Design-Time Knowledge	Implemented
14	Multiple Instances with a Priori Run-Time Knowledge	Implemented
15	Multiple Instances without a Priori Run-Time Knowledge	Implemented
	State-based	
16	Deferred Choice	Supported
17	Interleaved Parallel Routing	Supported
18	Milestone	Unsupported
	Cancellation and Force Completion	
19	Withdraw Task	Unsupported
20	Cancel Case	Supported
	Iteration	
10	Arbitrary Cycles	Implemented
	Termination	
11	Implicit Termination	Implemented

Table 4.2: Original Control Pattern Support in Gozer

with full support for lexical and global variables simplifies the implementation of these patterns, and together with Vinz task variables, the GWS is easily able to support 31 of the 40 patterns, and most of the remainder could be built with further support from BlueBox services.

Workflow resource patterns are concerned with the people, machines and other assets needed to execute a workflow. Seven areas comprising 43 patterns have been identified in [36]. These areas are Creation, Push, Pull, Detour, Auto-Start, Visibility, and Multi-Resource. Resource allocation and utilization is generally determined by the BlueBox system and can only be marginally controlled or extended at the workflow level, so some entire groups (those not currently possible in the BlueBox architecture) are not possible or applicable to the GWS, notably the Pull and Visibility groups. Of the remaining 35 patterns, support for 27 is provided by the GWS.

Exception handling is less strictly structured into groups and numbered patterns [35]. Instead a matrix of exception types (for expected exceptions; unexpected exceptions are left unclassified), exception handling level and possible recovery actions is created. Once again due to the GWS's basis in a complete language with a robust condition system, the vast majority of this matrix can be considered to be filled by the GWS.

Chapter 5

Conclusion

The previous chapters described the Gozer Workflow System (GWS), including its parallel and distributed processing support, the highly-dynamic Lisp dialect in which workflows are written, and the implementation of these features. The remainder of this chapter provides a summary of the contributions of this work and suggests some areas for future work.

5.1 Contributions

Although still evolving to meet upcoming needs and extend current capabilities, the GWS addresses key requirements associated with processing a vast number and type of workflow computations submitted by, or on behalf of, numerous clients. The workflow requirements vary greatly in terms of computational complexity, overall duration, and terms defined by service-level agreements. Gozer's ability to easily exploit local and distributed resources through implicit parallelization together with its high-level language approach to workflow authoring have allowed the rapid development of scores of high-volume production workflows at RiskMetrics Group. A typical 24-hour period will see around 10,000 new top-level tasks comprising about 45,000 individual fibers. Tasks during this period may run for as long as 12 hours or as little as 20 milliseconds, with the average being about a minute. If these 10,000 tasks were run back-to-back, they would require about 190 hours to complete [25].

It is this entire production-level workflow system that is the main contribution of this work. There exist other workflow systems, and other distributed systems, and other Lisp dialects for the JVM, and other dynamic languages, and other uses of continuations. The effective combination of these ideas that creates the GWS, however, may be unique.

In that same vein, the Gozer language itself can be seen as a combination of concepts from other languages. There is little truly new and unique in the Gozer language; this is by conscious choice (language design is hard and mistakes are costly). Rather, the focus was on starting with a stable base (Common Lisp and Java) and carefully combining that with selected ideas (e.g., Mutlilisp's futures and Groovy's dynamism) in a way that resulted in a practically useful language with a pragmatic degree of self-consistency and elegance.

A system in production use is never truly finished. The next section describes some possible ways that the GWS may evolve over time.

5.2 Future Work

Something with the responsibilities the GWS shoulders has many directions in which to contemplate additions and refinements. Some ideas for relatively major enhancements are discussed here. The layered design of the workflow system suggests a way to categorize these enhancements, beginning with the concepts of distributed programming and working down to the implementation of the core language.

Distributed programming abstractions Support for ever higher-level abstractions over distributed and parallel programming will help workflow authors implement their desired processes in the shortest amount of time. For example, a partial distributed implementation of "actors" (cooperating sequential processes) built on top of fibers is already available, and should be completed.

At the same time as higher-level abstractions are created, however, it is important to allow for the possibility of penetrating or customizing these abstractions to meet the needs of advanced programmers and more complicated processes.

Performance optimizations of distributed implementation Although the current implementation techniques are adequate and functioning in production, processing volumes are always going up and there's always a desire to "do more with less." The obvious inefficiencies in caching (Section 4.2.6) and the spawn limit (Section 4.2.4) should be corrected. What's more, the intelligent placement of fibers (near their data) and improved system-wide load balancing, possibly including the automatic suspension of actively running workflow tasks of low priority, should be investigated.

Language façades and tools Not every programmer is familiar with Lisp or has the desire to learn it. Continuing to improve tool support (such as introducing a Gozer development plugin for the Eclipse IDE, in addition to Emacs) only goes so far. The audience of workflow programmers could be broadened if a more conventional syntax was supported. Javascript seems like an ideal candidate (some language with native dynamic capabilities would lessen the impedance mismatch and still allow taking advantage of facilities like deflink). This would involve a new parser and probably some new libraries, but hopefully would be able to reuse much of the existing compiler and interpreter.

Another approach would be to extend the interpreter (possibly via a translator) to support some subset of the JVM bytecodes, thus allowing the use of Java as a workflow authoring language. There would be more restrictions involved in this technique due to the static nature of the language.

GVM implementation optimizations At the GVM level, optimizations can be targeted in many different directions, not just runtime speed. For example, optimizations for debugging could include the ability to step backward through the program or capture and inspect the program's state at particular times; these capabilities are now being seen in some mainstream language/IDE combinations (e.g., Microsoft's Visual Studio 2010) and might be relatively easy with the use of persisted continuations.

Runtime performance optimizations might include the ability to compile "leaf" functions into Java bytecode.

Bibliography

[1] Abelson, H., and Sussman, G. J. *Structure And Interpretation Of Computer Programs*, second ed. MIT Press, Cambridge, 1996.

[2] Baker, Jr., H. G. Shallow binding in lisp 1.5. *Communications of the ACM 21*, 7 (1978), 565–569.

[3] Boetje, J. Common lisp for java: An intertwined implementation. In *ILC '05: Proceedings of the 25th Annual International Lisp Conference* (Stanford, June 2005).

[4] Bray, T., Hollander, D., Layman, A., and Tobin, R. Namespaces in xml 1.0. Recommendation, W3C (World Wide Web Consortium), August 2006.

[5] Ciccarelli, E. An introduction to the emacs editor. AI Memo 447, Massachusetts Institute of Technology Artificial Intelligence Labratory, January 1978.

[6] Clarke, I. Swarm: Distributed computation in the cloud. In *P2P 09: IEEE Ninth International Conference On Peer-to-Peer Computing* (Seattle, WA, September 2009).

[7] Cyphers, D. S., and Moon, D. A. Optimizations in the symbolics Clos implementation. Tech. rep., Symbolics Inc, 1990.

[8] Dean, J., and Ghemawat, S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM 51*, 1 (2008), 107–113.

[9] Edelson, J., and Liu, H. *JRuby Cookbook*. O'Reilly Media, November 2008.

[10] Findler, R. B., and Matthews, J. Revised^6 report on the algorithmic language scheme. Tech. rep., Scheme Steering Committee, 2007.

[11] Gabriel, R. P. Lisp: Good news, bad news, how to win big. Tech. rep., Lucid, Inc, 1991.

[12] Gabriel, R. P., and Pittman, K. M. Technical issues of separation in function cells and value cells. Tech. rep., Lucid, Inc. and Stanford University and Symbolics, Inc., 2001.

[13] Gabriel, R. P., White, J. L., and Bobrow, D. G. Clos: Integrating object-oriented and functional programming. *Communications of the ACM 34* (1991), 28–38.

[14] Graunke, P., Krishnamurti, S., Hoeven, S. V. D., and Felleisen, M. Programming the web with high-level programming languages. In *European Symposium on Programming* (2001), Springer-Verlag, pp. 122–136.

[15] Grounds, N., Antonio, J. K., and Muehring, J. Cost-minimizing scheduling of workflows on a cloud of memory managed multicore machines. In *Proceedings of the 1st International Conference on Cloud Computing (CloudCom 2009), in Lecture Notes in Computer Science 5931* (December 2009), pp. 435–450.

[16] Haidle, B., Stoll, M., and Steingold, S. *Implementation Notes for GNU CLISP*. GNU, 2008, ch. 37. The CLISP bytecode specification.

[17] Halloway, S. *Programming Clojure*. Pragmatic Bookshelf, 2009.

[18] Halstead, Jr., R. H. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst. 7*, 4 (1985), 501–538.

[19] Hamilton, G., Ed. *JavaBeans Specification*. Sun Microsystems, Mountain View, CA, 1997.

[20] Juneau, J., Baker, J., Ng, V., Soto, L., and Wierzbicki, F. *The Definitive Guide to Jython: Python for the Java Platform*. Springer-Verlag, 2010, ch. 1. Language and Syntax.

[21] Koenig, D., Glover, A., King, P., Laforge, G., and Skeet, J. *Groovy in Action*, first ed. Manning, 2007.

[22] Kohlbecker, E., Friedman, D. P., Felleisen, M., and Duba, B. Hygienic macro expansion. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming* (New York, NY, USA, 1986), ACM, pp. 151–161.

[23] Krishnamurthi, S. The continue server (or, how i administered padl 2002 and 2003). In *Symposium on the Practical Aspects of Declarative Languages* (2003), Springer-Verlag, pp. 2–16.

[24] Leitāo, A. M. Increasing readability and efficiency in common lisp. In *Proceedings of the European Lisp User Group Meeting* (1999), Instituto Superior Tecnico.

[25] Madden, J., Grounds, N. G., Sachs, J., and Antonio, J. K. The gozer workflow system. In *Proceedings of the 24th International Parallel and Distributed Processing Symposium* (April 2010), IEEE.

[26] Marsh, J., Orchard, D., and Veillard, D. Xml inclusions (xinclude) version 1.0. Recommendation, W3C (World Wide Web Consortium), November 2006.

[27] McCarthy, J. A. The two-state solution: Native and serializable continuations accord. In *2010 ACM International Conference on Systems, Programming, Languages and Applications* (Reno, Nevada, October 2010), vol. 12, ACM.

[28] Miller, S. G., and Radestock, M. *SISC for Seasoned Schemers.* sisc-scheme.org, 2007.

[29] Pettyjohn, G., Clements, J., Marshall, J., Krishnamurthi, S., and Felleisen, M. Continuations from generalized stack inspection. *SIGPLAN Not. 40*, 9 (2005), 216–227.

[30] Queinnec, C. *Lisp In Small Pieces.* Cambridge University Press, Cambridge, United Kingdom, 1996, ch. 7. Compilation.

[31] Renggli, L., and Lienhard, A. Seaside: Web application toolkit for squeak. In *Smalltalk Join Event* (Douahi, France, August 2002), European Smalltalk Users Group.

[32] Romer, T. H., Lee, D., Voelker, G. M., Wolman, A., Wong, W. A., loup Baer, J., Bershad, B. N., and Levy, H. M. The structure and performance of interpreters. In *In Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)* (1996), ACM Press, pp. 150–159.

[33] Russell, N., Hofstede, A. H. M. T., and Mulyar, N. Workflow control flow patterns: A revised view. Tech. rep., BPM Center, 2006.

[34] Russell, N., ter Hofstede, A. H. M., Edmond, D., and van der Aalst, W. M. P. Workflow data patterns: Identification, representation and tool support. In *Proceedings of the 25th International Conference on Conceptual Modeling* (2005), Springer.

[35] Russell, N., van der Aalst, W. M. P., and ter Hofstede, A. H. M. Workflow exception patterns. In *Proceedings of 18th CAiSE* (2006), vol. 400, Springer, pp. 288–302.

[36] Russell, N., van der Aalst, W. M. P., ter Hofstede, A. H. M., and Edmond, D. *Workflow Resource Patterns: Identification, Representation and Tool Support*, vol. 3520 of *Lecture Notes in Computer Science.* Springer Berlin, Heidelberg, 2005, ch. 16, pp. 216–232–232.

[37] Seibel, P. *Practical Common Lisp.* Apress, Berkeley, CA, 2005, ch. 19. Beyond Exception Handling: Conditions and Restarts.

[38] Shrestha, H. K., Grounds, N., Madden, J., Martin, M., Antonio, J. K., Sachs, J., Zuech, J., and Sanchez, C. Scheduling workflows on a cluster of memory managed multicore machines. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '09)* (July 2009).

[39] Tao, W. *A portable mechanism for thread persistence and migration.* PhD thesis, The University of Utah, 2001. Adviser-Lindstrom, Gary.

[40] Tismer, C. Continuations and stackless python. In *Proceedings Of The Eighth International Python Conference* (Arlington, Virginia, January 2000).

[41] Van Der Aalst, W. M. P., Ter Hofstede, A. H. M., Kiepuszewski, B., and Barros, A. P. Workflow patterns. *Distrib. Parallel Databases 14*, 1 (2003), 5–51.

[42] Verna, D. Clos efficiency: instantiation. In *ILC '09: Proceedings of the 27th Annual International Lisp Conference* (MIT, Cambridge, Massachusetts, USA, Mar. 2009), Association of Lisp Users, pp. 76–90.

.

Appendix A

GVM Bytecodes

This appendix provides an informal description of all of the individual bytecodes implemented by the Gozer Virtual Machine. The general ideas behind the design of the bytecodes were laid out in Section 3.3. The opcode values were originally chosen so that related opcodes had related values (generally sequential within a group). This property has not been maintained over time as opcodes have been added and repurposed, so some groups include values that may appear arbitrary.

Each of the opcodes is self-contained, with the exception of `ENTER_BLOCK` and `PUSH_LAMBDA` (see Table A.5) which may (depending on flags) access the operand stack; the mnemonics for these instructions expand into more than one primitive opcode. This preserves the property that every instruction consumed by the interpreter can be represented in a single 32-bit integer, and offsets are simple to calculate.

In the following tables, the mnemonics use n and i to mean a positive 7-bit unsigned integer, *name* is a symbol (possibly `NIL`) and *keyword* is a keyword symbol, while *object* is any Java object.

The function call opcodes in Table A.2 are described in more detail in Section 3.3. All of the argument-passing opcodes described in Table A.4 operate on the implicit current function and (with the exception of the last one) are conceptually found as the binding forms inside a let form that encloses the entire function body. Their design is motivated by the discussion of Section 3.2, which also provides examples.

Name	Mnemonic	Value	Comments
`CHECK_VAR`	(check-cast *class*) (check-nil)	022	The object on the top of the operand stack is checked for validity, either that it is an instance of a given Java class (which includes nil) or that it is not nil at all. The stack is left unchanged, except that the object on top may have been a future that was resolved or had its type converted if required. Generated by the special form `the` or variable assignment if declarations were applied to the variable and code is being compiled for safety.
`NOOP`	(no-op)	000	Takes no action, modifies no state. Inserted for padding purposes by the compiler, and often ultimately removed by optimization.
`YIELD`	(yield)	004	Causes execution of the GVM to halt and return (to the Java caller) a continuation that can be used to resume execution. Available to the Gozer programmer as a special operator with the same mnemonic.
`GET_CC`	(get-cc)	003	Returns a continuation that can be inspected for debugging purposes. Available to the Gozer programmer as special operator with the same mnemonic.

Table A.1: Miscellaneous Opcodes

Name	Mnemonic	Value	Comments
`JJAVA`	(call-java-name *n name*) (push-java-property *name*)	017	Implements the · special form by either calling a Java method or accessing a Java bean property (which ends up being a method call). The high bit of the flags is set for the later, which always has one argument, the object on which to find the property. Otherwise, the flags specifies the number of arguments as usual.
`JPRIMITIVE`	(call-constant *n function*)	015	Calls a function known at compile time to be a low-level function (a GVM extension function and one that does not produce a continuable state). The call is bound at compile time to the object that implements the function, which is inserted into the constant pool.
`JRECUR`	(call-recur *n* nil)	011	Recursively calls the same function as is currently executing. The flet function binding form, which allows redefining functions in a lexical environment that can access the global function, does not emit this opcode (in contrast with the labels form).
`JSR`	(call *n object*)	016	The most general function call mechanism. The *object* can either be a symbol naming a function which will be dynamically bound at runtime (the usual case) or a function object (possibly a low-level function object). The compiler will produce function object references at high speed optimization levels.

Table A.2: Function Call Opcodes

Name	Mnemonic	Value	Comments
`GO`	(non-local-goto *tag*)	021	Unwinds the call stack to the point at which the matching *tag* was established. Used by the implementation of tagbody when lambda functions are involved (and the control flow cannot be reduced to local jumps).
`GOTAG`	(label *tag non-local?*)	020	Inserts a non-local go tag that may be jumped to with the `GO` opcode; functionally the same as `NOOP` otherwise. The same mnemonic is used with a false argument when creating assembler targets for jumps.
`JMP`	(goto *label direction*)	001	Unconditionally jumps to the location in the bytecode of the *label*, which may be before or after the location of this opcode, depending on the *direction* flag (the flag compensates for the fact that bytecode locations are always positive, but a PC relative jump may need to go backwards, i.e., a negative increment to the PC). Generated by the if, while and tagbody special forms.
`JMPNZ`	(jump-if-true *label direction*)	002	Pops the top value from the operand stack (resolving a future if needed), and, if the value is non-nil, jumps to the location of the label. Generated by the if special form.
`JMPZ`	(jump-if-false *label direction*)	003	Pops the top value from the operand stack (resolving a future if needed), and, if the value is nil, jumps to the location of the label. Generated by the if special form when the predicate expression is negated with not.

Table A.3: Control Flow Opcodes

Name	Mnemonic	Value	Comments
`LL_POP_ARG`	(%pop-arg)	030	Consumes the next argument and pushes it onto the operand stack, raising an error if there is no unconsumed argument. Followed by a binding opcode.
`LL_POP_AND_BIND`	(%pop-and-bind *name i*)	031	A speed and code-size optimization, consumes the next argument and binds it into the local lexical variable of the function at index *i*, raising an error if there are no unconsumed arguments.
`LL_POP_IF_NEXT`	(%pop-if-next)	032	If there is an unconsumed argument, consumes it and pushes it onto the operand stack. Used simultaneously as both the predicate and consequent of an if and followed by a binding opcode for the implementation of optional arguments with or without a default value.
`LL_POP_IF_NEXT2`	(%pop-if-next2)	033	If there is an unconsumed argument, consumes it and pushes it onto the operand stack; also pushes either T or NIL to indicate the presence of an unconsumed argument. Similar to `LL_POP_IF_NEXT`, but uses two binding opcodes to implement supplied-p arguments.
`LL_POP_KW`	(%pop-kw *keyword*)	034	If there is an unconsumed argument named by the keyword, consumes it (and the keyword) and pushes the value onto the argument stack. Similar to `LL_POP_IF_NEXT` but for keyword arguments.
`LL_POP_KW2`	(%pop-kw2 *keyword*)	035	If there is an unconsumed argument function named by the keyword, consumes it (and the keyword) and pushes the value onto the argument stack; also pushes either T or NIL to indicate the presence of a matching unconsumed argument. Similar to `LL_POP_IF_NEXT2`, but for keywords.
`LL_POP_REST_ARG`	(%pop-rest-arg)	036	Consumes any remaining arguments and pushes a list containing their values onto the operand stack. Followed by a binding opcode.
`LL_ASSERT_EMPTY`	(%pop-no-args)	037	Emitted only when code is optimized for safety, raises an error if there are remaining unconsumed arguments.

Table A.4: Argument Binding Opcodes

Name	Mnemonic	Value	Comments
`ALLOCATE_LOCAL`	(allocate-local *i*)	042	Allocates space for *i* indexed variables within a particular block (usually a function).
`ENTER_BLOCK`	(enter-block *name len-label cleanup-label*)	010	Enters a new named block. Blocks are used for the implementation of the unwind-protect form as well as the return-from form and possibly the let form. The `ENTER_BLOCK` opcode (along with `PUSH_LAMBDA` is unique in that certain less-commonly used functionality is pushed onto the operand stack and then consumed by the opcode itself.
`POP`	(pop *object*)	013	Pops and discards values from the operand stack. If the *object* is NIL, pops only one item, otherwise pops until an identical object is found on the stack. Used to cleanup the stack following certain special operators including while, progn and unwind-protect.
`PUSH_LAMBDA`	(push-lambda *name doc ll*)	012	Creates and pushes onto the operand stack a function whose bytecode immediately follows the current PC, and increments the PC to the instruction after the lambda's code. This opcode may produce closures or top-level functions and may be followed by a function binding form. As with `ENTER_BLOCK`, certain optional values may first be pushed onto the operand stack.
`RET`	(return-from *name*)	014	Used to implement the special form of the same name, unwinds the call stack through the most recent lexically enclosing (and thus active) block entered with `ENTER_BLOCK` and a matching name. The top of the block's operand stack is transferred to the outer block's operand stack.

Table A.5: Block Opcodes

Name	Mnemonic	Value	Comments
`PUSH_CONSTANT`	(push-constant *object*)	025	Pushes an object from the constant pool onto the operand stack.
`PUSH_IMMEDIATE`	(push-constant *object*)	026	Pushes numeric immediate data (as stored in the final two bytes) onto the stack. The flags specify the Java class that should be used (Long, Integer, BigInteger) and whether the value should be negated. The mnemonic is the same as for general constants, and the assembler choose to use this opcode of the *object* can be represented as immediate data.
`PUSH_LOCAL_VAR`	(push-local *i1* *i2*)	041	Pushes the value of a local variable having index *i1* onto the stack. If *i2* is given, this index is encoded in the flag and the corresponding variable is also pushed onto the stack; this is a speed and code size optimization.
`PUSH_VAR`	(push-special-var *name*) (push-var *name*)	024	Pushes the current value of either a special (global) or heap-allocated lexical variable onto the operand stack. The flag encodes special versus lexical access.
`SET_LOCAL_VAR`	(set-local *i* *return?*)	040	Sets the value of a local variable having index *i* to a value popped off the operand stack. If return? is non-nil, then the value is (conceptually) pushed back on the stack for subsequent access. The special form let can produce this and sets return? to NIL whereas the special form setq must set it to T.
`SET_VAR`	(bind-var *name* *return?*) (replace-var *name* *return?*) (replace-special-var *name* *return?*) (set-special-var *name* *return?*)	023	Changes the value of a special or heap-allocated lexical variable having the supplied *name*. The value is obtained from and returned to the stack as with `SET_LOCAL_VAR`.

Table A.6: Variable and Data Opcodes