Structuring data via behavioural synthesis

(Thesis Format: Monograph)

by

Geoffrey Z. Wozniak

Graduate Program in Computer Science

Submitted in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

School of Graduate and Postdoctoral Studies

The University of Western Ontario

London, Ontario, Canada

August 29, 2008

THE UNIVERSITY OF WESTERN ONTARIO

SCHOOL OF GRADUATE AND POSTDOCTORAL STUDIES

CERTIFICATE OF EXAMINATION

Supervisors	Examiners
____________________	____________________
____________________	____________________
Supervisory Committee	____________________
____________________	____________________

The thesis by

Geoffrey Z. Wozniak

entitled

Structuring data via behavioural synthesis

is accepted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

____________________	____________________
Date	Chair of Thesis Examination Board

Abstract

This thesis furthers work in the dynamic analysis of programs by examining how to derive data structure definitions from the behavioural aspects of programs. We refer to this technique as *behavioural synthesis*.

In particular we first present a general data typing mechanism, called a *dynamic abstract data type* (DADT), whose instances act as proxies for other data types in a computation. Each instance of a DADT has the ability to reconfigure its internal representation and detect the context in which it is used. This is followed by an example of a DADT that represents the union of other abstract data types whose interfaces are not necessarily disjoint; techniques for specializing their use in a program are also demonstrated. The specializations are realized as code in the same language as the program, promoting program evolution. Furthermore, the code generation is based on source code locations, does not require explicit annotations and is performed interactively. Finally, we show a method of deriving the composition and relationship of classes through augmentation of a language runtime. The approach permits the use of objects without class definitions.

Behavioural synthesis is presented as a design exploration aid to support the notion of using a typical programming language as a specification language and may be viewed as extending the duality between program and data to a duality between program structure and data structure.

Keywords: abstract data type, dynamic analysis, behavioural synthesis, code generation, structural derivation, program evolution.

For Słoneczko.

Acknowledgements

I am indebted to many people for giving me the chance to pursue this work and giving me support via kind words or poignant (and deserved) criticism. I want to thank my academic supervisors and mentors over the years, specifically Lila Kari, Stephen Watt and Mark Daley. Special gratitude must be given to Doug Vancise for giving me the opportunity to teach and for showing me how rewarding it can be.

Spending time in the company of my friends has been a cornerstone of my well-being and fundamental to the maintenance of my rationality. In particular, I want thank Chris Power, Bradley Simmons and Mike Burrell for fruitful discussions, Doug Vancise (again), Maia Hoeberechts and Jamie Andrews for meeting the frequent needs of a bridge quorum, and finally Andrew, Mike, Ren, Greg, Ian and Adam for providing me with entertainment over the years.

To Annette and Stuart, my mother and father, I am indebted to you for giving me independence while growing up so that I could follow my curiosity. I would not be doing this were it not for your fierceness, tenacity and carefully placed indifference. My siblings, John and Miranda, deserve thanks for being those rare relations that are also good friends.

Finally, Słoneczko, no words are enough. Thank you so much for your love and patience.

Contents

List of Figures

Chapter 1

Introduction

According to Wirth, programs are composed of structure and behaviour [87]. The specification of structure and behaviour often affect each other: data structures may be influenced by the algorithm that manipulates them whereas algorithms may be designed to work with some given data. This thesis examines ways that the structural descriptions of data can be derived from the behavioural elements that manipulate them, a technique we term *behavioural synthesis*.

Additionally, we follow the ideas of Naur by considering programming to be a design activity [52]. We propose that behavioural synthesis can be integrated with existing programming languages, allowing it to be used as a design exploration aid by treating an implementation language as a specification language. As a specification language, details of the structure may be omitted and introduced later through behavioural synthesis techniques. Thus, the program can be specialized by generating code in the same language used to communicate the specification.

We demonstrate three methods for realizing behavioural synthesis. The first is a framework for defining data types whose instances can track their usage, detect the context in which they occur and reconfigure their internal representation in response to these stimuli. We call these *dynamic abstract data types*. Second, we define a type that represents the union of multiple abstract data types whose interfaces are not necessarily disjoint. Ambiguous situations using the type are resolved interactively and the context of the operation is remembered. Specialized versions of the type are generated based on the contexts in which the type is used. Lastly, we demonstrate a technique for employing objects in a computation without defining classes for the objects. Instead, classes are created as required, including their structural composition and relationships to other classes. In all three cases, we analyze the efficacy of the method and discuss its shortcomings.

These are presented as dynamic analysis techniques. Any static or lexical information required for deriving structure is communicated through the dynamic environment of the program by way of code transformations executed prior to evaluation.

The derivation of structure demonstrated in this thesis does not devise entirely new data structures. The structures implicitly used in the behavioural description of a program are assumed to be known, but the choice of exactly what layout to use is unspecified — all that is known is the operations to be performed on an object and their parameters. New structures can be created, but only in the sense that it makes the structure of the data fit with the operations. The creation of entirely new data structures requires the generation of original operations. No such attempt to do so is made in this work.

We define behavioural synthesis as the derivation of data structure from program behaviour. The derivation may come from a description of the behaviour in the form of program code, by observing a running program or some combination thereof. Here, data structure refers to both the actual layout of data in the sense of structures for data manipulation, such as linked lists and arrays, as well as the relationships between different data objects.

Behavioural synthesis aims to facilitate structural derivation during the development of programs. The idea is to capture the intuition of data structure that often accompanies descriptions of algorithms without explicitly defining the structure.

Thus, when writing an algorithm, the precise layout of data can be ignored, only to be described in terms of a known data type, such as a stack, list or set, eliminating the need to provide the definiteness required by the algorithm (as per Knuth's definition [41, section 1.1]). The semantics of the operations provide form to the data being manipulated. Therefore, instead of devising a new collection of operations, we leverage current ones. To this end, the techniques demonstrated are based on integration with existing languages.

The motivation for behavioural synthesis stems from work in formal specifications and agile programming [14], and software development based on volatile requirements. We argue that unifying a specification language with an implementation language could be beneficial for agile programming and that behavioural synthesis techniques are helpful for such a unification because the combination of language and technique can be used to represent entities encompassing multiple abstractions. Furthermore, code can be generated specializing the abstractions based on use.

Van Lamsweerde notes that formal specifications tend to grow out of informal specifications [80]. A formal specification, as defined by van Lamsweerde, is "the expression, in some formal language and at some level of abstraction, of a collection of properties a system should satisfy." Informal specifications are usually written in natural language as part of a requirements document. Since it is unlikely all the specifications are known at the start of a software project, the transition from informal specification to formal specification helps determine what must actually be stated formally.

Specification languages themselves are not usually designed to be implementation languages as they normally describe the functional or behavioural properties of a program and do not deal with resource management [22]. Still, there are benefits to being able to execute specifications, which effectively turns them into a form of implementation language. Making a specification language executable affords the opportunity to integrate them into the same framework as the implementation [66].

Whether integrated with an implementation language or not, specification languages are transformed into implementations to be verified against requirements. Depending on the syntax and semantics of the specification language, it may be possible to refine the specification into an implementation by way of code transformations [13]. Such transformations may be done semi-automatically, as in the case of various program synthesis systems (for example KIDS [74] with DTRE [10]). It has long been recognized that stepwise refinement in this manner is a useful way to develop programs [86].

One such programming methodology that stresses incremental development is agile programming [14]. In contrast to formal methods, agile programming techniques stress working code through rapid release cycles, and de-emphasizes documentation and formal specifications; working code is highly valued because it provides something tangible the user can evaluate. The working code may not provide all the desired functionality, but enough is present to provide feedback. The subsequent evaluation of the (partial) implementation likely leads to changes in the specification through revealed deficiencies. With the frequent changes made to the specification in agile practices, the formality of a specification that is maintained outside of the code base bogs down the process of implementation.

With stress on code over specifications, Reeves' argument that code is design [60] suggests that programming is a design activity in the sense proposed by Naur. Naur put forth the Theory Building View of software development [52] and argues that a programmer accumulates knowledge of a particular problem (and how to solve it) through experimentation, akin to a method as abstract as the scientific method. In short, the programmer has access to many tools and techniques to determine the structure of a program, but no particular order need be applied in order to determine that structure. Formal specifications in agile approaches are seen as a potential tool to be drawn from some collection of tools, but not a required one.

Additionally, Osterweil discusses the general notion of processes and argues that process descriptions develop out of processes [55]. That is, the description of a process comes about by working through a process one or more times. He further argues that software processes are also software and can be captured in a program. He cautions, however, that the descriptions required for such a program may be very difficult to specify precisely [56]. Filling in details can be achieved by having someone observe the processes, noting the details. Although the details may not be universally applicable, they may address the task at hand. Osterweil's argument suggests that incomplete specifications can be executed and observed to build more complete specifications based on specific inputs.

Executable specification languages afford the opportunity to generate implementations from interaction. This can be done by making constructs in a specification language act as code generators. An advantage of generating code in this way is that it does not require explicit annotations in the source code since the code itself is the "annotation" guiding the code generation. Additionally, it can be unobtrusive to the user. (These are both properties called for by Smaragdakis [73].) Interactive code generators of this form would be a useful tool for agile programming since they could mitigate the work involved in writing a specification by making it a working part of the implementation.

It may be advantageous to make the specification language work within the implementation language for a considerably more streamlined workflow when a specification is part of an implementation. Arguing against this idea, Schmid and Hofmeister [66] claim that "the combination of the specification with the implementation cannot occur on the level of source code" and outline a way to combine them at the binary level. This ignores the fact that a specification language could be embedded in the implementation language and the compiler could combine them at the binary level.

Combining the specification and implementation languages could be beneficial for agile programming. Specifications are generally more succinct than implementations which may allow for more functionality to be delivered within a release cycle. It could also provide a way to use formal methods with agile approaches without incurring too much of the unwanted overhead. By making the constructs of the specification language active agents in the development of the program by generating source code, a specification language could promote program evolution and development through evaluation of releases.

The behavioural synthesis techniques described in this thesis are useful in a setting where a specification and implementation language are combined because they can capture multiple abstractions in a single data object. Furthermore, the profiling abilities of the object, its ability to capture the context in which it is operating and the ability to communicate with other objects indirectly provide for a useful code generation scheme where a very general data type can be made specific. This is demonstrated in Chapters 3 and 4 with dynamic abstract data types and program specialization. Chapter 5 demonstrates another way it can be helpful in that class definitions can be omitted, to be devised at runtime based on the usage of objects. Both of these techniques delay precise structural definitions but permit evaluation, thus allowing for process observation as described by Osterweil.

The rest of this thesis is organized as follows. Chapter 2 describes work related to behavioural synthesis, such as choosing data representations based on profiling and analysis, dynamic reconfiguration of structure, generating programs from program descriptions and systems meant to aid in the design and development of programs. Chapter 3 describes dynamic abstract data types, how they are defined, an example and an analysis of their performance. Building on this work, Chapter 4 describes a use of dynamic abstract data types where multiple types are combined into a single type with ambiguous operations. Ambiguity is resolved interactively and, using contexts, a specialized version of the type is generated, including operations. Chapter 5 demonstrates a way to use objects without class definitions and build the class definitions at runtime. We summarize the work and present conclusions in Chapter 6.

Lastly, we note that all code in this thesis is given in Common Lisp, and all realizations of constructs referred to or used are implemented in Common Lisp. Readers unfamiliar with the definitions of object-oriented terminology in Common Lisp are referred to the lexicon at the end of this dissertation.

Chapter 2

Related work

The sections following summarize works that are related to the current dissertation. Section 2.1 surveys approaches to automating choice of data representations for objects employing abstractions to varying degrees. For the most part, the works involve analysis of the program, possibly combined with its behaviour, and may involve querying the user for unknowns. The survey concentrates on work related to that presented in Chapter 3.

In Section 2.2, we cover the notion of dynamic adjustment of structure, concentrating on data structure. We seek to describe the unifying themes that form the basis of the methods undertaken in this thesis. Section 2.3 reports on different ways of synthesizing programs based on behavioural descriptions, possibly with feedback.

Lastly, Section 2.4 describes techniques and tools used to help refine or verify designs. Section 2.5 briefly summarizes the chapter.

2.1 Automatic data structure selection

Automatic data structure selection is the act of choosing concrete data representations for objects through analysis, possibly in conjunction with profiling. The idea is that the mapping of the interface to an implementation of an abstract data type is not revealed to the programmer. Emphasis, therefore, is placed on use of the interface instead of relying on knowledge of the data layout when writing code. Analysis consists of examining the use of the interface in the program and choosing representations that maximize the efficiency with respect to some resource, usually computation time.

Selecting data representations

The general problem of selecting data representations is discussed by Rosenchein and Katz [65]. They define two extremes: selecting one of many possible representations (the easy problem) and creating a representation from an abstract description (the hard problem). They outline a way to handle something between the extremes by combining elements of known representations into one meeting the requirements. This is done in either a hierarchical or cross-product fashion. In the former, representations can be built up from each other while in the latter, representations must exist separately. They go on to describe a data specification language to capture the requirements of a data structure so that a representation can be synthesized from a set of known representations.

In the current work, we acknowledge the continuum defined by Rosenchein and Katz, but use a considerably different methodology. The concerns addressed by their data specification language are significantly low-level, such as describing the number of words required for a datum. Also, it is unclear from their examples whether providing descriptions of data structure requirements is simpler than describing the data structure. Furthermore, their approach is based purely on the operations to be supported and do not (directly) take into account the data to be manipulated or usage patterns.

Booth and Wiecek describe an approach to abstract data types that associates performance information with functional aspects [11]. Multiple representations are associated with an interface augmented with cost equations for each operation using probabilistic models to describe possible inputs. Cost expressions are evaluated with respect to a particular machine and input type to provide the best representation for the task at hand. Their approach is compatible with dynamic abstract data types as presented in Chapter 3 and could be captured in a trigger.

White analyzes the costs involved with using multiple representations for variables [84]. He assumes that representations are chosen during program design, but can be assigned either to variables or program regions. Changes can be made at runtime but only at designated points. White's work generalizes Booth and Wiecek's approach.

More recently, Yellin has looked into the cost of switching representations of a component at runtime, termed the *adaptive component problem* [92]. Here, a component is assumed to be a subsystem of a larger system, not necessarily an object. He presents a nearly-optimal 3-competitive algorithm for the case when a component has exactly two implementations. Furthermore, a switch happens only after an action request for the component is processed. Yellin's work demonstrates that for autonomic systems, the ability to adjust dynamically can be cost effective.

Using Wirth's idea of stepwise refinement for program development [86], Kant and Barstow describe a system for refining programs incrementally in a semi-automatic fashion [35]. The user provides information about the expected usage of a data structure, then the system refines the specification to reflect the changes using coding rules and automated efficiency analysis via exploration of the design space; the specification is not strictly stored as program code, but program code is eventually produced.

An expert system for choosing data representations has been described by Johnston [34]. Johnston's system considers parameters such as the desired operations, data persistence, concurrency, type of contents to be held, and space and time considerations. No code is synthesized from the selection, although Johnston proposes ways to do so.

The work of Kant, Barstow and Johnston aims to absolve the programmer from making concrete choices about data representation by describing behavioural elements, for the most part. Their approaches fall closer to the easy problem on Rosenchein and Katz's spectrum because they are working with collections of known data types with a single interface, substituting in implementations.

Finally, we note an approach to detecting where data representations can be changed in program given by O'Callahan and Jackson with their *Lackwit* tool [54]. Using a form of type inference, Lackwit determines what variables interact in C programs, grouping them together by type. O'Callahan and Jackson propose that such groupings allow for different representations to be used, but do not follow up on it, instead using the type information to look for unused data and memory leaks. A similar approach was taken by Hall using the Hindley-Milner type system to optimize list representations (without runtime coercion), however Hall *does* use the information to produce efficient list representations for use in the program [30].

Low's work and SETL

Two early efforts in choosing data structures automatically can be found in the works of Low and efforts to optimize the programming language SETL.

Low's methodology focusses on compilation techniques combined with execution monitoring and user interaction [48]. After the program is written, it is executed so that the compiler can collect usage information regarding the frequency of certain operations. The input provided is expected to be typical for the program. A static analysis is then performed, using the collected information, to determine what variables and expressions interact by grouping them into equivalence classes so that common representations can be used; common representations are preferred in order to avoid the need for conversion. (In Low's work, compatibility is required at the data layout level for operations that operate on two or more elements of the same type.) Next, representations are chosen for the equivalence classes. This is done via a hill climbing technique in an attempt to minimize the space and time costs of executing the program. Representations are chosen from a fixed library of representations, complete with data regarding the costs of the operations. The user is consulted to address certain variables in the cost equation, such as the probability of some condition, the average size of data to be processed or a preferred representation for certain variables or expressions. These parameters are used to select representations for variables.

Low's approach has the potential to add considerable time to compilation and analysis, especially considering the need for user intervention. The basic methodology is reasonable and adopted in our approach but with more emphasis on working while the program is running. We provide a mechanism for objects to evolve to account for changes in the source code while Low's approach does not. Furthermore, cost equations in Low's work are given in precise terms based on the number of machine instructions required to perform the data structure operation [47]. Specifying costs in such a manner requires intimate knowledge of not only the machine architecture, but also the compiler. Given the speed of processors at the current time, such equations seem overly pedantic, unreliable to obtain and likely of little value. Low's approach could be modernized by measuring the costs of operations for small data sets to approximate a cost equation, where constants likely have more impact on the execution speed. This would also make it more amenable to use on different architectures.

A prominent undertaking with respect to automatic data structure selection is as an optimization technique in the programming language SETL [70]. SETL's defining characteristic is its native support for sets, tuples and mappings. Due to the generality of these constructs, SETL supports multiple representations of them. Schonberg *et al.* describe an analysis to determine certain shared properties of sets in order to create specific representations that permit efficient execution of operations [67, 68]; Dewar *et al.* present similar work as a form of program refinement, where compiler annotations in a sublanguage of SETL can be added to SETL programs to designate data representation choices [17]. Schwartz details compiler optimizations used in the SETL compiler to determining suitable representations for high-level objects such as sets and mappings [69].

These techniques go to considerable effort to describe what is in a set at a certain time using a form of type analysis (be it manual or automatic). This illustrates the main difference with Low's approach in that user interaction is not required. Instead, the problem is defined so that it can be dealt with purely by static analysis.

Other efforts

Another useful application of data structure selection is in the field of matrix manipulation. Bik and Wijshoff provide a compiler technique for automatically changing code managing dense matrices to that for managing sparse matrices, saving the user the need to handle complex data structures directly [9].

Working in the compiler, Peterson shows how to adjust tagged representations of data to be more efficient [58]. This is done by analyzing type information of variables. Peterson's motivation stems from the fact that hardware and software representations of data objects may not be compatible, namely, the tags are not used in hardware. Thus, Peterson changes the representation from tagged to untagged at different points in the computation to obtain more efficient use of the data without affecting proper type identification. Similarly, Richardson performs a data-flow analysis to determine when and where a different representation of a data type can be applied to improve the efficiency of a program [63, 64].

Carriero Jr. describes a static analysis technique to determine efficient data layouts for tuples spaces used in tuple space machines with languages such as Linda [12]. Tuple spaces themselves are interesting because they have very little form in and of themselves: They are simply large collections of associations [25]. Roughly speaking, this is the way we have viewed objects in Chapter 5. We view Carrerio Jr.'s efforts to structure tuple spaces for more efficient lookup as a form of behavioural synthesis.

Tables in Symbolics Common Lisp can change representation based on the options provided when a table is created, the current size of the table and type of the data set [1]. For example, a table may start out as an association list with a small number of elements, but change to a hash table when the number of elements grows beyond an efficient size for association lists. These are a special case of the dynamic abstract data types discussed in Chapter 3 and do not allow for general monitoring capabilities.

Note that with all the work surveyed in this section, the approach to choosing data structures is mainly a static affair. Most methods involve analysis through a compiler taking behavioural cues from the program text. In all of our methods, we combine information from program text with dynamic properties to construct structural descriptions or provide suitable representations for objects.

2.2 Dynamic reconfiguration

In broad terms, dynamic reconfiguration applies to many fields: just-in-time compilation [7], continuous program optimization [40], self-adjusting computation [2] and type-feedback optmization [3, 33], just to name a few. It can also be seen in more basic computational elements, such as balanced trees and path compression [36]. These fields will not be looked at in detail since they are not immediately applicable to the current work. Instead, we will concentrate on the dynamic reconfiguration of structure in the sense of changing the layout of some entity. At the same time, we will summarize the unifying themes of the aforementioned works and examine some specific applications of dynamic reconfiguration that are judged to be more pertinent to the current discussion.

Filesystems and databases

A primary factor involved in using dynamic reconfiguration is the condition that certain information affecting the efficiency of a system is not available until the system is working. Consider filesystems. Madhyastha and Reed demonstrate that by changing policies based on input/output patterns, such as prefetching aggressively during periods of sequential read activity, significantly fewer I/O operations need be performed compared to a static policy [49]. ZFS[1], a filesystem developed by Sun Microsystems, employs dynamic striping to adjust to newly added disks automatically, a pooled storage model that adjusts to changes in disk configuration, and automatic error detection and correction to simplify administration [75].

Databases are another application, closely related to filesystems, where data usage has an affect on how the data is stored and accessed. Agrawal *et al.* show that changing the database layout in response to the workload requested of the database can provide noticeable improvement in query running times [4]. At the level of file layout, Ghandeharizadeh, Ierardi and Zimmerman describe a way to minimize seeks in disk activity to increase throughput speeds by careful rearrangement of blocks when files are deleted [26]. Wolfson, Jajodia and Huang consider the case of distributed database systems by allowing database objects to replicate based on how often they are read from or written to [88]. Ng *et al.* show how to change the structure of a long-running query to handle changes in the underlying database

[1]Formerly known as the Zettabyte File System.

[53].

Filesystems and databases demonstrate that altering structure to account for a highly variable environment can be beneficial, despite the overhead involved in monitoring environmental change.

Type-based restructuring

For instances of a given type, reconfiguring them to work in different situations is based on the notion that the data can be looked at in different ways, although it is conceptually the same from each perspective.

Lazy types are a device presented by Berzal *et al.* that provide a mechanism for an object to alter its structure over time, within some predefined set of possibilities [8]. A lazy type is a class with a set of attributes and methods, but instances of the class may only have a subset of the attributes implemented. Methods are chosen based on the subset of attributes currently defined for an object. The defining feature of lazy types is not so much their ability to reconfigure themselves by occasionally adding attributes, but the fact that they represent a large number of possible types. Instead of creating variants of a certain type, you could use a lazy type to unify the type and use operations uniformly throughout.

One of the themes of lazy types is that data should drive the structure of objects. This is echoed by Skopin who argues that data should be multiply structured [72]. Skopin's point is that data is often viewed in different ways and using an appropriate view (or structure) makes working with that view simpler. The idea, then, is to make the data fit the problem instead of the problem fit the data.

Wadler's *views* provide a way of achieving this by having abstract data types export views of the data in addition to operations [82]. A view is a set of functions that convert between two representations. The different representations can then be used to define operations on the type as desired. For example, lists could be viewed in the traditional manner (an element followed by another list or the empty list) or the traditional manner in reverse (a list, possibly empty, followed by an element). Defining functions to operate on lists can be simplified by choosing the appropriate view. Continuing from the previous example, retrieving the first element of a list is easily defined using the traditional view, whereas retrieving the last element is easy using the reverse view.

Views make defining operations more natural and provide for a simple conversion mechanism, but the cost of conversion must be considered carefully in the use of views. Furthermore, the places of conversion are fixed by the library implementor. Thus, while they do provide for multiple structures for a particular datum, efficiency over the lifetime of the datum is not considered. That is, they are operation-centric and not data-centric.

Another approach to views was proposed by Andrews and Dobkin in the form of active data structures [5]. Their intuition was that a data object could occupy its own processor in a multi-processor environment and contain a single, conceptual collection of data that is accessed in different ways. We share some of this intuition, that is, seeing the data in different ways, but we use a very different implementation tactic.

While views and their ilk provide dynamic reconfiguration in the sense of changing data from one form to another, another form of dynamic reconfiguration closer to that described in Chapter 3 is provided by Hailpern and Kaiser [29]. They describe their PROFIT language that provides a mechanism for its fundamental data elements, known as facets, to be of a certain *breed*[2]; a breed corresponds roughly to an interface. Containers for breeds allow for different facets to be swapped in and out to provide access to different information collections (their work is motivated by online stock trading applications). Breeds with facets are a limited form

[2]Hailpern and Kaiser use a delightful ranching metaphor to describe their dynamic reconfiguration constructs, complete with breeds, stalls, pens and herds.

of dynamic abstract data type (described in Chapter 3) but without autonomy.

The work of Hölzle and Ungar with the Self compiler provides a slightly different approach to type-based restructuring in that it encompasses feedback [33]. The basic principle of their work is that type information gleaned from use is fed back to the compiler to optimize certain aspects of the code, such as method dispatch. For example, the most often used method can be inlined at the site of the call, leaving other methods to be found via dynamic dispatch. (Similar compilation techniques are used in just-in-time compilation systems, such as the HotSpot Client Compiler for Java [42].) In general, we can say that if code is data, then dynamic information can alter the way code is handled. In this sense, all dynamic optimization techniques can be viewed from the same perspective of DADTs as described in Chapter 3 on code objects.

Dynamic algorithm selection

Dynamic adjustment has also been considered for algorithms. We mention a few of these results even though they are not about structural reconfiguration because they use the same underlying principle as the current work and further support the notion of employing online observation of a process for its own refinement. In general, this is known as use of a polyalgorithm, that is, a set of algorithms and rules for choosing when to use which one.

A rudimentary example of dynamic algorithm selection can be found in the GNU MP library [27]. Here, the same operation may use different algorithms based on the number of machine words required to hold an operand and its type. Furthermore, how this choice is made is user-configurable, although made at compile time. This is similar to the work described in Section 3.2.

More complex examples are found in the work of Armstrong *et al.* and Park *et al.* [6, 57]. Armstrong *et al.* show how to alter an algorithm for a given task using reinforcement learning at runtime; these changes are made to a binary (that is, compiled) version of the program when the current algorithm is seen to be lagging in performance. Park *et al.* present an expert system for choosing algorithms for finding keys in a set with some guidance from the user on desired properties. Code is synthesized to describe the algorithm which also resulted in a specific data representation.

Long-term considerations

Most of the effort put toward dynamic reconfiguration addresses issues of efficiency with respect to time. Gabriel and Goldman take a slightly different view [23]. They assert that software must be more conscientious and take into account its environment. Specifically, they comment that software must consider long-term aspects of its use, such as its installation and maintenance by becoming a collaborator in its own future. In this sense, ZFS is more conscientious than Madhyastha's policy changes since it addresses issues of administration. Gabriel and Goldman's argument suggests that dynamic reconfiguration may be better suited for processes characterized by long running times, possibly with periods of inactivity, such as the use of interactive code generation in Chapter 4.

The ideas of Gabriel and Goldman have a direct influence on the methods described in this thesis. Specifically, we take the notion of software being involved in its own deployment and maintenance to include the notion of its development. The constructs and techniques presented in this work explore their ideas of software as a collaborator to aid in continuous redesign of the system. These considerations are the impetus for using a dynamic framework as opposed to a set of static analysis tools for the operation of the work presented herein.

2.3 Program synthesis

The work described in this section is primarily about program synthesis and the adjustment of the program while under development. The techniques presented could be used in a running system (and in some cases are) but the overhead involved is likely too great, unless given sufficient time to stabilize.

Structural synthesis

An important effort in the synthesis of programs was introduced by Tyugu [78]. Tyugu derives programs from structural descriptions of the variables involved in the computation and relationships between them. The specification of structure is written in a logic language which formalizes the problem as a theorem in a form of intuitionistic logic. Program synthesis involves proving the theorem and translating the proof into a program, usually expressed in the lambda calculus. The approach is referred to as the *structural synthesis of programs* (SSP) because it only relies on structural information about the problem domain and produces a behavioural specification for relations automatically. SSP has been used successfully in various industrial applications [79].

It may be tempting to consider our work the compliment to Tyugu's, given the complementary terminology; however, this would be inaccurate. Part of the reason for this is that we have not grounded our derivations in precise logical formulations. Since information we use to infer structure is based on the semantics of existing programming language constructs and the intent of the construct may be ambiguous, we cannot guarantee correctness in our presented results. This is easily seen with the results presented in Chapter 5.

High-level synthesis

In hardware circuit design, a technique thematically similar to our methodology has been developed termed *high-level synthesis* or — appropriately enough — *behavioural synthesis* [24]. To avoid potential confusion, we will refer to this synthesis technique as high-level synthesis. High-level synthesis builds circuit designs from algorithmic descriptions in a programming language above that of a hardware description language; typical input languages consist of C and VHDL. The programming language enables the circuit designer to specify a circuit using behavioural elements without concern for low-level structural details of circuit design, such as timing data, resource allocation or module selection for various abstract operations like addition.

As detailed by Singh et al., high-level synthesis has received considerable attention in the circuit design community, but still fails to produce designs as efficient as those produced manually [71]. Part of this is because the code must be written in a certain style to deal with syntactic variance in the expression of algorithms.

While there are certainly parallels between high-level synthesis and the current work, there are key differences. The most obvious difference is that high-level synthesis is intended to produce output in a different language from the source language. The assumption in high-level synthesis is that both structure and behaviour are specified in the source language to the extent that another realization of the behavioural aspects of the specification can be produced. As a result, different designs are found through various code transformation techniques applied to the source language or some other intermediate form. The technique of high-level synthesis is similar to that demonstrated in Chapter 4, where the description of structure is synthesized from known elements. Our approach is more “active” in the sense that the execution of the behaviour provides the structural elements; high-level synthesis, as with many of the techniques described regarding automatic data structure selection, is primarily a compilation technique. (This does not preclude feedback via evaluation of synthesized circuit designs or manual analysis, of course. As an example, see Catapult C from Mentor Graphics [50].)

Generative programming

Program synthesis shares some qualities found in *generative programming* [16]. Program synthesis seeks to produce programs in accordance with an abstract specification describing some aspect of the desired program. Generative programming provides meta-programming facilities combined with code transformation procedures for generating custom versions of abstractions. In relation to the current work, our efforts are to provide abstractions that become specialized over time. In particular, our efforts resemble that of active libraries, that is, entities that apply generative programming techniques [76, 81].

Aspect-oriented programming and open implementation

Our approach to behavioural synthesis could be seen as using one aspect of the program to generate another, where an aspect refers to those in *aspect-oriented programming* as put forth by Kiczales *et al.* [39]. They define an aspect as “properties that affect the performance or semantics of the components in systemic ways” where a component is something cleanly captured as an object, procedure or other language-level construct. In their examples, they use matrix algorithms as an application, citing matrix representation (structure) and permutation (behaviour) as aspects. If we take the application to be the act of programming, it is reasonable to consider structural and behavioural elements to be separate aspects that are usually woven together by the programmer. (Weaving is the act of combining the aspects to make a program.) Thus, it may be more accurate to say that we synthesize a program aspect with our technique, not so much a program itself.

As a brief digression, we examine whether behavioural synthesis is itself a form of aspect-oriented programming. According to Filman and Friedman, aspect-oriented programming combines quantification of program statements with obliviousness on behalf of the programmer regarding those quantifications [21]. While it is debatable whether behavioural synthesis meets this requirement, our implementation does use many of the dynamic quantifications they mention.

Open implementation, also proposed by Kiczales [37], is a software design technique where modules are augmented with meta-programming constructs to control the implementation. A Collection DADT (dynamic abstract data type) presented in Chapter 4 works as a form of open implementation. Here, a Collection DADT represents a module with a meta-interface that controls the underlying implementation based on use. The details of the implementation are partially hidden, but manipulable.

We mention one other effort in this area: *presentation extensions* as described by Eisenberg and Kiczales [18]. Presentation extensions are a way to change the expressivity of a program based on how it is presented to the programmer, with or without semantic extension to the language. Behavioural synthesis as presented in this work could be realized as a form of presentation extension since it is more suitable as a tool used in development and not in deployment.

Adaptive programming

A specific form of aspect-oriented programming and open implementation that our work closely resembles is that of Lieberherr's *adaptive programming* [43]. In adaptive programming, the goal is to make programs "structure-shy", that is, avoid relying on structure in the description of the behaviour. Thus, emphasis is put on behavioural descriptions by decoupling them from structural elements. This is done by making methods less dependent on class structure allowing for methods to rely only partially on a class descriptions. Lieberherr's realization of adaptive programming is known as the *Demeter Method*.

The primary intent of adaptive programming intersects greatly with our own; however, adaptive programming has a much larger scope and a considerably different implementation in the Demeter Method. Adaptive programming defines an entire software development methodology whereas we do not take behavioural synthesis so far. Also, adaptive programming requires only that behaviour be partially specified; we often rely on the full specification.

The Demeter Method extends the syntax of C++ for the specification of the stages of adaptive programming. One way it does this is with object descriptions so that classes can be derived from them [44]. This is similar to the work presented in Chapter 5. The class derivation strategy takes concrete descriptions of objects and factors out common attributes to come up with a suitable class hierarchy. The fundamental difference with our technique is that objects are described before running the program and are not immediately used in the desired behavioural context. In short, deriving classes from object examples using the Demeter Method is a static derivation scheme using syntactic extensions that, while evolvable, is based on structural descriptions.

Learning from examples

Providing example object descriptions in the Demeter Method or in the way presented in Chapter 5 is suggestive of Lieberman's characterization of programming by example [45]. The similarities are mostly nomenclatural though, since Lieberman's work tries to generalize behaviour given example computations, but does not focus on devising a structure for the computation.

Some methods for deriving structure from examples can be found in the works of Hoff *et al.* and Winston [32,85]. Hoff *et al.* look to derive structure using logical rule descriptions and do so successfully in various cases. However, they acknowledge their approach suffers from the need of the user to have a deep understanding of the approach in order to use it effectively. We actively tried to avoid this problem by looking to harness conventions so that the user can go about using our tool in a usual development setting. Winston's work argues that good descriptions are required to obtain acceptable results, which supports the notion of assuming full behavioural descriptions before attempting to derive structure, a requirement used in Chapter 5.

Ignoring type definitions

The work presented in Chapter 5 provides a way for objects to be used in a program without declaring their structure ahead of time. In effect, the same result was achieved by Mishra and Reddy for abstract data types in ML-like languages [51]. Their approach uses a static type-checking over the program text to construct abstract data type definitions for function parameters, although it is not necessary to synthesize these type definitions. The key difference with respect to our work is that their method relies on syntactic recognition of constructors, that is, functions that create objects of a certain type whose names correspond to the descriptive name given in a type definition; related constructors are grouped together to determine the data type declaration. Our work in Chapter 5 cannot use this method without relying on convention since Common Lisp does not have such syntactic markers. (Regardless, the difference in type systems makes their technique cumbersome to use in Common Lisp.)

2.4 Development aids

The works discussed in this section fall across the categories described earlier in different ways. In general, these works view the program as something that changes over a period of time or as something that is immediately manipulated for the purposes of helping the programmer develop the program.

Deriving program knowledge dynamically

In what is closely related to our work with respect to determining structural descriptions, Guo *et al.* detail an approach to inferring the abstract data type of an object through dynamic analysis [28]. Basic elements in the code are observed and related to each other based on the operations performed on them. In their analysis, observations are done at the binary level, thus interactions are restricted to the rudimentary, namely arithmetical operations. Abstract data types are found by using a union-find algorithm to relate objects; each object starts out as its own abstract data type.

The most striking relation to our work is how this infers information that is implicit in the program. The abstract data types found in their experiments are grouped in ways that are not expressed directly in the program, such as two integers used to hold the same kind of measurement. More tellingly, it infers structure from behaviour, given rudimentary structures to start. It does this in a bottom-up fashion, similar to the approach taken in Chapter 5.

Still, Guo *et al.* do not go as far as to relate abstract data types with each other, nor do they attempt to determine the structure of aggregate types. Furthermore, the objects observed are limited to what is discernible in binary representation of a program; this design limitation is likely why the analysis is basic compared to ours.

Similar results have been achieved by Ernst *et al.* for finding program invariants using the *Daikon* dynamic analysis tool [19, 20]. The invariants are used to infer properties about the program, including relations between data. The properties reported by Daikon are mostly related to conditions that occur in the program, such as the range of values a variable takes on. These derived elements of the program wouldn't necessarily be taken and written directly as code, but the methodology of studying behavioural properties of the program guided our work.

KIDS and DTRE

The KIDS program development system [74] is a program transformation system that provides the programmer with tools to turn specifications into running programs. One of the components of KIDS is DTRE, the *Data Type REfinement* system [10]. DTRE allows types to be refined by correctness-preserving transformations, starting from abstract types with multiple realizations, such as sets and mappings. (In this sense, KIDS with DTRE is similar to SETL in how programs are specified.)

KIDS and DTRE are concerned mostly with formal specifications. They do not make extensive use of dynamic qualities of the code, although they stress specification of behaviour over that of structure. DTRE supports semi-automatic selection of data structures for variables and selection can be guided by annotations. In many ways, the methodology is the same as optimization techniques used in the SETL compiler.

While DTRE has similar goals to behavioural synthesis — that is, techniques for the exploration of program designs and late binding of data representations — it takes a considerably more formal approach. Refinement in DTRE is about fine-tuning known types whereas we look to form a description of the type in the first place.

A programmer's apprentice

Work on a project to produce a *Programmer's Apprentice* by Rich, Shrobe and Waters relates to our work in that they attempt to use what a programmer does to intelligently assist in program development [61, 62, 83]. Themes found in their work that are used in the design of our methods are discussed here.

One of the principles of the Programmer's Apprentice is that it is additive and incremental; it does not interfere with regular activities and works mainly to augment the programmer's efforts by passively interpreting his intent. As such, its functionality is optional. This is in contrast to the "do what I mean" approach taken by some systems, such as the one found in the Interlisp Programming Environment [77], where the system takes an active role in interpreting intent. We strove to use the passive approach as much as possible in this work.

One of the key elements of the Programmer's Apprentice is the use of *clichés*. A cliché is a common structure or behavioural pattern used to describe a situation that can be reused; it is essentially a kind of template with constraints. Waters captures clichés in a special format so that they can be edited and generate code with his KBEmacs tool [83]. We have adopted a limited form of clichés by employing convention (see Chapter 5). This prevents the need to represent them in a fashion external from the source code.

Typed feature structures

An interesting approach from the field of architecture by Woodbury *et al.* demonstrates a form of design space exploration similar to the class hierarchy derivation work in Chapter 5 [89]. Woodbury *et al.* describe the use of *typed feature structures* as a formal framework for design space exploration using an example of a simple house layout. The framework involves mapping descriptions to structures, then refining the structure by refining the description. Descriptions are given in a language that combine types with structures. Structures are generated based on the descriptions, which can then be translated back into descriptions for further refinement. In their example, they show how a general description of a basic house layout can be used to generate different designs that are considerably more specialized.

The refinement process for typed feature structures is similar to the refinement of class descriptions presented in Chapter 5. There, class structure is generated from behavioural descriptions. The class structure can then be added to the program to refine the program. Furthermore, different class structures are generated in order to fit the behavioural description. Our work differs somewhat in that changes to structure are made by altering the behavioural descriptions and not by explicitly describing the structure.

Context-oriented programming

Context-oriented programming (COP) is an approach to programming that treats context explicitly in the definition of a program [31]. This supplies a mechanism for behaviour to be dynamically adapted to changes in context. For example, Costanza and Hirschfeld demonstrate COP language constructs for Common Lisp that can change class definitions based on layers [15]. Layers capture the notion of a specific context and the definition of a class can be different within different layers. Context is used frequently throughout the work in this thesis, although we do not explicitly define layers as Costanza and Hirschfeld do; instead, the context is taken from the environment of the computation. Nevertheless, the notion of behavioural changes based on context is integral to the behavioural synthesis techniques presented here, notably in Chapters 4 and 5.

2.5 Summary

The works presented show that there have been a variety of ways that behavioural elements of programs or processes are harnessed to influence structure. Most of the approaches described make use of static descriptions. Those that use runtime characteristics tend to use them as input to further static analysis. Our work makes more of an effort to integrate evaluation with analysis, permitting closer interaction with the process on the part of the developer.

Chapter 3

Dynamic abstract data types

The first approach to behavioural synthesis we describe is a framework for defining abstract data types, called *dynamic abstract data types* (DADTs)[1]. This chapter defines DADTs, a way to use them in a program, analyzes their runtime overhead and describes a way to reduce this overhead.

The benefit to DADTs over typical abstract data types is that instances of DADTs can be configured to collect information about the context of their use and manage their internal representation; operations for DADTs follow a protocol describing procedures to facilitate such collection and management. In effect, DADTs act as proxies for other objects, such as an object whose instances can use different implementations that work with the interface defined for the type. This provides a mechanism for objects to change their internal representation in order to choose different algorithms for a single conceptual operation. Furthermore, DADTs afford the opportunity for optimizations at the level of individual objects since each object has an independent tracking mechanism. Section 3.1 describes DADTs and its associated protocol.

In Section 3.2, an example of a DADT is presented that demonstrates how to define a DADT and describes an example implementation. The DADT is used for a set data type whose instances adjust to the types of elements they hold. Additionally,

[1]The work in this chapter is based on a paper presented at the International Lisp Conference in 2007 [90], co-authored with Mark Daley and Stephen Watt.

the DADT for sets demonstrates a simple example of a data type that supports the union of two interfaces — that is, sets whose elements can be totally ordered and those that cannot — and manages the disparity between them.

Lastly, in Section 3.3, we analyze the overhead of DADTs and present a way to mitigate the overhead through a technique called *stabilization*. We then discuss the shortcomings of DADTs and situations where they may be employed effectively. Section 3.4 summarizes the results of the chapter.

3.1 Protocol

A high-level overview of dynamic abstract data types is given in Figure 3.1. Instances of DADTs are associated with measurements and actions as well as the actual representation of the data. Code for monitoring objects is invoked by certain operations that have access to the internals of the instances, allowing it to take measurements and invoke the actions associated with the instance. The actions associated with the instance can use the knowledge base to make changes to the actual data representation.

Recall that an abstract data type is an abstraction that defines objects based on the set of operations available for those objects [46]. We call the set of operations that define the interface to the objects the *interface operations*.

A *dynamic abstract data type* (DADT) is an abstract data type whose instances contain a set of data objects that implement the structure necessary for use with the interface operations as well as extra information. The set of contained data objects is called the *current representation* of the DADT. Instances of DADTs are known as *DADT objects*. An interface operation that operates on a member of the current representation is known as an *actual (interface) operation*. Note that each member of the current representation need not work with all the interface operations. The ex-

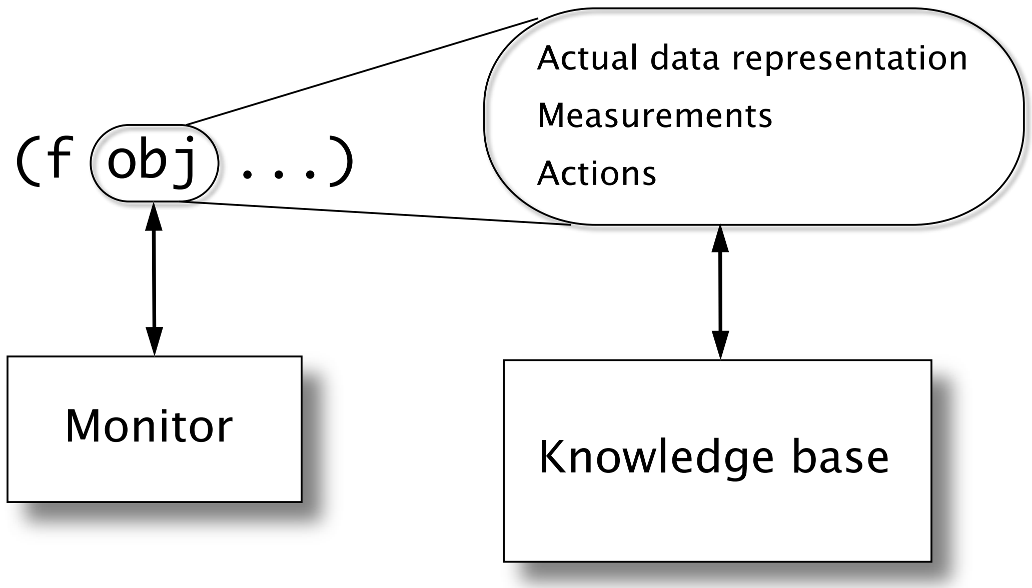


Figure 3.1: Overview of the DADT protocol used in a program.

tra information contained in a DADT is used for managing the mapping of interface operations to members of the current representation.

The management of DADT objects is achieved by collecting information about the use of the object and invoking actions when certain conditions arise. Information collection and action execution take place when interface operations are performed, that is, an operation from the interface for a DADT is executed and DADT objects of the type designated by the DADT are passed as parameters. We limit the investigation to interface operations because the work involved in tracking other operations introduces much overhead for arguably little gain in information. Non-interface operations do not take advantage of the internals of DADT objects and thus, say little about how the structure of the object is used. While it is possible to expand the contexts in which DADT object management can take place, such as events outside of interface operations, that is not explored in this work.

Each interface operation is associated with a DADT. A certain subset of the arguments passed to an interface operation are known as the *significant arguments*. The significant arguments are DADT objects that will be monitored or manipulated by the interface operation. These arguments are indicated when interface operations are defined.

An interface operation must perform three tasks: execute the expected functionality associated with the interface operation, supply some way for the significant arguments of the interface operation to collect information about their use into objects called *resources* and provide an opportunity for the significant arguments to react to situations using *triggers*. Additionally, the interface operation must obey certain directives provided in the environment of the computation. These are described below, followed by details on the control flow for the various stages of the protocol.

Executing actual interface operations

DADT objects are meant to be effective realizations of data structures in a running program. Thus, when an interface operation is performed on them, the operation must be executed in accordance with the expected semantics of the operation. Depending on the members of the current representation, this may be as straightforward as delegating the operation to the members of the current representation or it may involve a complex dispatching mechanism.

Proper execution also requires returning the proper values. In many cases, the values returned by the actual operation can be passed through with no processing. There is a special case, however, that must be handled: if the actual operation returns a value that should be turned into an instance of a DADT object.

Suppose we have an abstract data type whose instances have type A with an interface operation f whose type is $f : A \times A \rightarrow A$. We define a DADT for this abstract data type whose instances are of type D and define an interface operation f_D for use with objects of type D. The implementation of f_D employs f, using its return value, to prevent the need to re-implement the semantics of the actual operation. However, without proper processing of the return value, the type of f_D is $D \times D \rightarrow A$. To accommodate for this, the DADT interface operation should transform the return value into an object of type D.

Another special case follows from this. Again, suppose we have an abstract data type A with a DADT D and an interface operation $g : A \rightarrow A$. Furthermore, the value returned by g is the same object that was passed to it with the internal state of the object altered. That is, it preserves object identity but does not preserve value equality. In the definition of the DADT interface operation g_D, care must be taken to preserve the identity of objects returned by actual operations.

Resources

A DADT object should collect information by making observations about aspects of the program that pertain to the management of the DADT object. Thus, each DADT is associated with a set of other objects called *resources*. Resources are simply data stores for observations.

Two points in the execution of a DADT interface operation are designated for the collection of information: before and after the actual operation is performed. These are known as the *initial* and *final measurement stages*, respectively. Designating two stages for information collection provides a mechanism for measuring change with respect to some resource. The canonical example is time: to measure the time elapsed by an operation, we must note when the operation started and when it finished.

As the example of time demonstrates, it may be necessary for information to be communicated from the initial measurement stage to the final. Such information is known as an *intermediate resource value*.

Triggers

In addition to resources, each DADT object is associated with a set of functions called *triggers*. A trigger is used to detect conditions (if necessary) and react to them by performing some set of actions. When a trigger on a DADT object is executed by the protocol, we say the trigger has been *run*. A trigger may or may not affect the state of the object's current representation at the time it is run. When a trigger has a direct effect on the current representation it is called *active*, otherwise it is called *passive*. Note that this is a dynamic designation and not a static one. A trigger may be active in some situations and passive in others.

Triggers have access to the context of the interface operation being performed. The context consists of the DADT object the trigger is associated with, the interface operation being performed when the trigger was run and the arguments passed to the interface operation.

There are three important points to consider when running triggers:

- dependencies between triggers;
- order of execution;
- the application of triggers from multiple DADT objects within a single interface operation.

To deal with dependencies between triggers for a single DADT object, we stipulate that triggers must signal whether they are active or passive after performing their action. If a trigger signals that it is active, no further triggers in the set of triggers are executed. (This is analogous to the evaluation of a series of *if-then-else* clauses.) Additionally, the values returned by the triggers that have been executed so far are provided to each trigger so that triggers can communicate down the chain.

In the case of execution order, a sorting mechanism is provided to order the triggers if necessary. Once sorted, triggers are run in succession until an active one is found, if any.

Triggers, as defined so far, are associated with a single DADT object and are ill-suited to cooperation with other DADT objects. Consider the following scenario: Two DADT objects O_1 and O_2 are passed to a single interface operation such that their current representations must be synchronized in order to complete the operation. It is possible that the triggers for O_2 will alter its current representation to be different than that of O_1's, thus preventing the correct execution of the operation. The isolation of triggers makes it difficult to handle inter-object communication.

To solve this problem, we further subdivide the collection of triggers into *local triggers* and *class triggers*. Local triggers are associated with individual DADT objects and are only used when an interface operation has one significant argument. Conversely, class triggers are associated with the class of DADT objects and are used when an interface operation has two or more significant arguments. The triggers themselves are not different except in what is processed; both kinds of triggers are called in the same manner. However, class triggers will have access to all significant arguments whereas local triggers will only have access to the single significant argument they are associated with. Although the choice of which set of triggers to run is mutually exclusive, class triggers have access to, and can run, local triggers. The opposite case is also possible. Using class triggers provides a way for custom protocols to be used with respect to communication between DADT objects.

Finally, we note that resources may depend on the state of the current representation. For example, a resource may be tracking the number of operations between changes to the current representation. When a trigger indicates it is active, the resources associated with a DADT object are adjusted, if necessary. This is known as *resetting* the resources.

Directives

Directives are a general mechanism by which DADT objects can communicate with each other as well as control the protocol. They facilitate shared "experiences" by DADT objects to guide triggers and various optimizations.

Directives are found in directive environments. A directive environment consists of bindings of names to values. There are three kinds of directive environments used in the protocol: global, local and class.

Global The global directive environment is, as the name implies, global to the entire computation and unique. All program elements can add and remove bindings from the global directive environment. If a binding is requested from the global directive environment and no matching binding is found, a default null value is returned.

Local Local directive environments are contained within DADT objects. It is possible for other elements of the program to add and remove bindings from the local directive environment of a DADT object. If a binding is requested from a local directive environment and no matching binding is found, the request is forwarded to the global directive environment.

Class Each DADT has its own class directive environment. Class directive environments have the same characteristics as local directive environments, save for where they are found.

There are three directives that are always present in the global directive environment: RUN TRIGGERS, MEASURE INITIAL and MEASURE FINAL. By default, they are all bound to the value *true*. The standard directives control whether certain stages of the protocol are executed, namely the running of triggers, the initial measurement stage and the final measurement stage, respectively. Collectively, these stages are known as the *extra stages*, so called because they represent extra work over that of the work for the actual interface operation and can be disabled.

Control flow

The precise order of operations for the protocol is not defined outside of certain restrictions. One restriction was given earlier in the discussion on resources, namely that the initial measurement stage must be run before the actual interface operation and the final measurement stage after. A further restriction is that triggers must be run either before the initial measurements stage or after the final measurement stage. This is to prevent problems with measurement of changes, specifically time.

Another constraint on the flow of control pertains to conditional execution of certain stages of the protocol, specifically, running triggers, and the initial and final measurement stages. Each of these stages is executed only when the corresponding standard directive is bound to *true* in the appropriate directive environment. If the DADT interface operation has exactly one significant argument, the local directive environment of the significant argument is used to lookup the necessary directive. When there are two or more significant arguments, the class directive environment of the DADT the operation is associated with is used.

The default flow of control is shown in Figure 3.2. The solid lines are the flow of control and the dashed lines represent the communication of information between different stages.

3.2 Example

To demonstrate the use of dynamic abstract data types, this section presents a simple DADT for a set data type. The Set DADT will define objects that adjust their current representation based on the configuration of the types contained within.

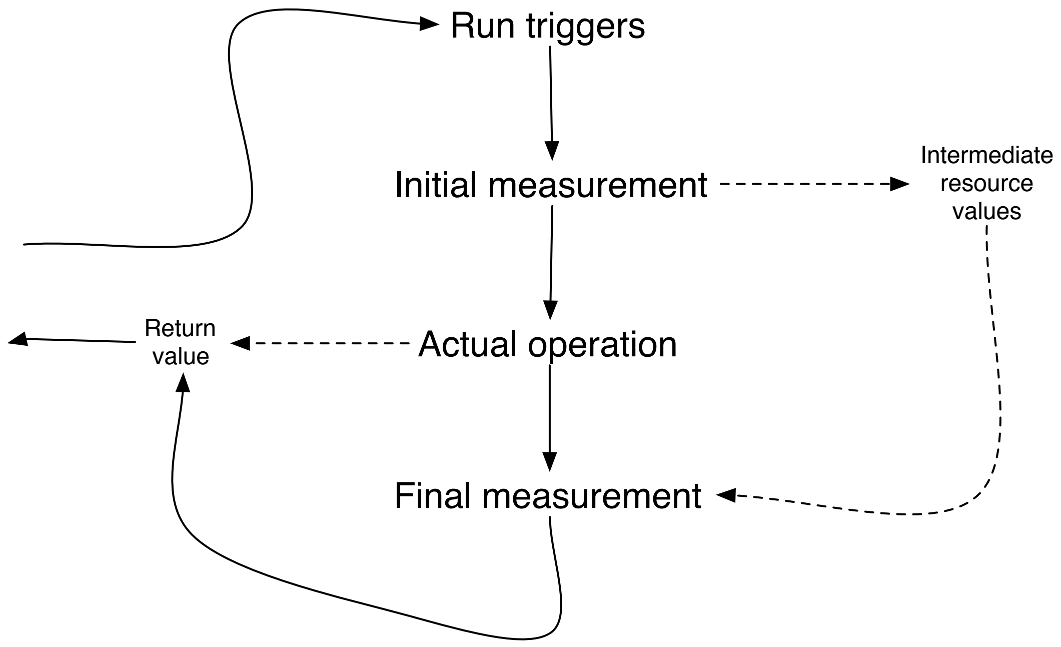


Figure 3.2: The default control flow of the DADT protocol. Solid lines represent the flow of control and dashed lines represent communication between stages.

The goal of the Set DADT is to define a set data type that does not require the programmer to be sensitive to the parameters defining a set. Instead, a set will adjust its internal configuration based on its contents. Extra operations will be supported in the event that the contents support them, such as ordering statistics when the elements make up a set with a total order.

Sets require a membership test to operate properly, meaning some definition of equality must be provided. A common approach to handling this is to associate an equality function with each set object. This can be problematic if different types of elements are to be put into a set. For example, in Common Lisp the equality function `eql` will compare characters in an expected fashion: ‘a’ is the same as ‘a’. However, it may consider two instances of the string literal “hello” to be different[2], even though it is normal to think of them as the same. Proper use of a set from

[2]This is because `eql` tests for object equality and it is implementation dependent how string literals are allocated.

the programmer's perspective means knowing how it was created to manage the different types that may be present, possibly by defining a custom equality function.

To assist the programmer in ignoring such details, the Set DADT is furnished with a knowledge base that associates types with equality functions that provide for usual expectations, such as assuming that two strings with the same contents are to be considered the same. Furthermore, a way of identifying the type of an object in accordance with this baseline knowledge is provided through the function `type-of*`. Details of the knowledge base will be provided as necessary throughout the discussion.

Defining the Set DADT is achieved with the following steps:

1. Determine what must be observed in set objects and define resources to do so.
2. Determine the actions that must be taken to maintain expected behaviour of set objects and the conditions under which these actions must take place, then define triggers capturing this information.
3. Define a class for Set DADT objects.
4. Define the interface operations that implement the protocol.

The interface for the Set DADT contains the usual set operations in addition to some statistics operations, such as `nth-smallest` and `nth-largest`. To keep track of the types of elements found in a set, the only operations that can change the contents of a set are `insert-element`, `delete-element` and `clear`.

Lastly, note that the current representation of a Set DADT object only ever has one member; we do not manage multiple representations in parallel. Of the collection of representations available for use in the current representation, all support the usual set operations and some also support ordering statistic operations. The lone member of the current representation is changed to meet the expectations of the operation, if possible. For example, if the operation `nth-smallest` is executed on a Set DADT object where the only member of the current representation does not support the operation and the contents can be restructured to support that operation (perhaps the contents consist of only integers), then the member of the current representation is converted to a structure that does support `nth-smallest`. Thus, there is no need to maintain multiple set structures to implement the required interface operation semantics; the elements of the set can be transfered to another structure, replacing the current one.

Defining resources

Recall that Set DADT objects must properly manage membership tests and support ordering operations when possible. These properties are dependent on the types of the elements in a set. In addition to tracking the types of elements in a set, we will also track the number of elements of each type so that we know immediately when a type is no longer present. Thus, we have to track when elements are added and removed from a set, which amounts to watching three operations (`insert-element`, `delete-element` and `clear`).

Defining resources requires that we define a class for the resource and indicate how to measure it. This class will be called `type-counter`. Measurement is achieved by defining methods on the generic functions `measure-resource-initial` (initial measurement stage), `measure-resource-final` (final measurement state) and `reset-resource` (resetting resources). In this example, we only require an initial measurement.

Example code for taking the initial measurement is given in Figure 3.3. The context of the measurement is provided through the arguments passed to the method. The first argument is the resource itself, the second is the DADT object being operated on, the third is the interface operation being performed and the fourth is a

```
(defmethod measure-resource-initial
      ((tc type-counter) set (op (eql 'insert-element)) args)
  (let ((elem (args-required args 1)))
    (unless (member-p set elem)
      (increment-counter tc (type-of* elem))))))

(defmethod measure-resource-initial
      ((tc type-counter) set (op (eql 'delete-element)) args)
  (let ((elem (args-required args 1)))
    (when (member-p set elem)
      (decrement-counter tc (type-of* elem)))))

(defmethod measure-resource-initial
      ((tc type-counter) set (op (eql 'clear)) args)
  (clear-counter tc))
```

Figure 3.3: Code to take the initial measurement of the `type-counter` resource for use with the Set DADT.

structure containing the arguments passed to the interface operation. The function `args-required` returns the second required argument (indexing starts at 0) which is the element to be inserted or deleted. We then check to see if the element is present or not and adjust the counter for the type of the element accordingly. For simplicity, code to verify the element was inserted or deleted has been omitted, although it may not be necessary to second-guess the actual interface operation.

Defining triggers

Defining triggers for DADTs can be complicated because the conditions that we want to detect may be intuitive, but difficult to capture programmatically. In the case of the Set DADT, the knowledge base provides the "fuzzy" information used to detect the conditions we need. Thus, we discuss the organization of that information before describing how the triggers will be defined.

The knowledge base associates types with equality functions and ordering functions. Initially, only the standard equality functions[3] and standard types are included, although other types can be added. The ordering functions define the natural ordering on the elements of their associated type. Querying the knowledge base for an equality function consists of providing two sets of types T, T' and an equality function f. The knowledge base searches for an equality function f' such that if $t \in T \cap T'$ and $e_1, e_2 \in t$ then $f(e_1, e_2) = f'(e_1, e_2)$. That is, the new equality function preserves the semantics of the previous equality function. A similar querying procedure exists for ordering functions. Thus, the knowledge base provides a way to determine what equality functions are suitable for sets of types.

As an example, suppose $T = \{\texttt{character}\}$, $T' = \{\texttt{character}, \texttt{string}\}$ and $f = \texttt{eql}$. The knowledge base could return $f' = \texttt{equal}$ since `equal` behaves the same as `eql` for characters and provides the expected behaviour for strings. This

[3] `eq`, `eql`, `equal` and `equalp`.

functionality allows triggers to determine what the configuration of a set object should be when elements are added or removed.

Note that the semantics of f' are dependent on what is in the knowledge base. Consider the case of equality functions for strings. The functions `equal` and `equalp` behave differently for strings (the former is case sensitive, the latter is not); however, it is legal for the knowledge base to return `equalp` in the previous example, even though this would result in different semantics for the program. The intention of the knowledge base is to describe usual behaviour for use in the program, but it must be configured to provide the desired semantics.

Triggers themselves are functions that take four arguments: the DADT object the trigger is associated with, the interface operation the trigger is being called from, a structure containing the arguments passed to the interface function and a structure containing the return values of all previously run triggers. It must return two values, the second of which is a Boolean indicating whether the trigger was active or passive.

Recall that the insertion, deletion and clearing operations can affect the current representation of a Set DADT object. All of these operations present situations that we must write triggers for, although the deletion and clearing operations are analogous. We must also consider the case of performing an interface operation asking for ordering statistics.

For insertion, we define a trigger that detects the insertion of new types of element and adjusts the current representation to accommodate the new type. This may be as simple as changing the set's membership test function, but may involve allocating a new structure and transferring the elements. Performance considerations are also taken into account. If the trigger must alter the current representation with a new structure, it strives to choose the most efficient one given the parameters. Practically speaking, this amounts to choosing a native hash table representation

```
(defun compaction-trigger (dset op args collected)
  (when (compaction-condition-met-p dset op)
    (compact-set dset)
    (values nil t))
```

Figure 3.4: A simplified version of the trigger for insertion

whenever possible since in most cases, hash tables provide the best performance. In the case of elements that have incompatible testing functions, the set is partitioned based on testing functions and different structures are used for each partition.

When deleting elements, we are only concerned with the case when the number of elements of a specific type reaches zero. In this case, the structure of the set may be amenable to compaction to undo changes made earlier. Care must be taken when applying such changes since they must be done *after* the actual interface operation — to do otherwise might affect the semantics of the actual interface operation. This will be addressed more fully when defining the interface operations.

Handling ordering statistics operation is straightforward: If the types found in the current representation support an ordering, change the structure in the current representation to one that supports the ordering and perform the operation. The structure is only changed to something else when an element of a type that does not work with the ordering function is inserted.

The definition of the triggers is involved and a full code listing would be of little value because most of the code involves the particulars of choosing an appropriate structure and changing the structure; it is specific to the particular implementation. A simplified version of the trigger for compaction can be found in Figure 3.4.

```
(define-dadt-class dset ()
  ()
  (:local-triggers insertion-trigger
                   compaction-trigger
                   orderable-trigger)
  (:resources type-count)
  (:default-rep unordered-set))
```

Figure 3.5: The Set DADT class.

Defining the DADT class

DADT classes are classes that provide the necessary information for use with the DADT protocol. The implementation used in this example provides a construct `define-dadt-class` that extends the usual class definition mechanism (`defclass`) to define DADT classes.

The DADT class definition for the Set DADT is found in Figure 3.5. It defines a class `dset` whose instances will be created with no slots, three local triggers and a single instance of `type-count` making up its resources. The default representation for any instance of `dset` will be an instance of `unordered-set`. This makes up the current representation for the instance when it is created.

Defining the DADT interface operations

While DADT classes provide information for the protocol, DADT interface operations actually execute the protocol. The definition of a DADT interface operation is done with the `define-dadt-operation` construct. It provides many options for defining functions that implement the protocol in addition to using reasonable defaults to minimize code required to specify an operation.

DADT interface operations are defined as methods on generic functions. (The

(a)
```
(define-dadt-operation member-p dset ((set dset) elem))
```

(b)
```
(define-dadt-operation insert-element dset ((set dset) elem)
  (declare* (returns (self set))))
```

(c)
```
(define-dadt-operation delete-element dset ((set dset) elem)
  (declare* (manual t))
  (measure-initial)
  (let ((val (delete-element (dadt-rep set) elem)))
    (measure-final)
    (run-triggers)
    val))
```

Figure 3.6: Example DADT interface operations for the Set DADT.

specification of a DADT interface operation is very similar to `defmethod`.) Some examples can be found in Figure 3.6. Each DADT operation must specify the name of the operation, what DADT it is associated with, the arguments, an optional set of special declarations and an optional body of code. By default, arguments specialized to the DADT associated with the operation are taken to be the significant arguments. In the examples, the `set` argument specialized to the class `dset` is considered to be the significant argument for each operation.

If the body of the operation is omitted, code is generated to execute the protocol (see Figure 3.6a). This includes properly saving and passing intermediate return values in the measurement stages and handling the return value. The body of the operation can be provided directly in which case local functions that implement various stages of the protocol are provided.

Returning the proper value(s) must be done carefully. The actual operation may return the element of the current representation that was passed to it, in which case the current representation should be updated with the potentially altered value. Another possibility is that one of the returned values must transformed into a DADT object – this may happen when the interface operation creates new instances of the element of the current representation passed to it. This is can be seen in Figure 3.6b which specifies that the argument `set` is returned as the first (and only) value.

As mentioned above, the deletion operation should run its triggers after the actual operation. This is accomplished by the definition in Figure 3.6c. The function `dadt-rep` returns the current representation of a DADT object. (Recall the current representation of the Set DADT is a single object, so it constitutes all of the current representation.) The special declaration indicates that the body of the function is provided manually.

Discussion

We have defined a Set DADT that provides the programmer with a way to ignore details about the parameters defining set objects and rely on these parameters to reveal themselves dynamically. Since the structure is dependent on these parameters, this permits the structure to be determined by behaviour and delays the choice of structure until runtime.

Another way to use the Set DADT would be to use multiple implementations in the current representation and perform each operation on every member. This would provide for profiling of each implementation to see which was the most efficient with respect to some resource. If choosing the most efficient representation is the goal, then this approach is feasible, if not potentially resource-heavy when done for every object. Compared to a single implementation in the current representation, use of multiple implementations may be more accurate due to the ability to make more accurate measurements.

That being said, it does not help much with respect to the goal of the Set DADT, which is to alleviate the programmer from the need to parameterize the set object. In this case, multiple implementations do not help much because reconfiguration may still be necessary. For example, suppose the only two implementations are a hash table and a balanced tree. If integers and characters are entered into the set, the balanced tree cannot be used and thus, should be eliminated. However, if the hash table gets to a state where it contains only integers and it is passed to an ordering statistic function, it may be beneficial to convert it into a balanced tree. This series of changes is very similar to what would happen with a single implementation.

The Set DADT provides some initial insight into the potential uses for behavioural synthesis. Set DADT objects are, to some extent, an abstraction on conventions. Certain common configurations are captured by the knowledge base and leveraged by the objects. This prevents the need to fully describe objects in the code in some circumstances. In a sense, some of the responsibility for object definition is shifted to the runtime and away from the programmer.

3.3 Optimizations

Dynamic abstract data types present many opportunities for runtime adaptation and profiling, but incur much overhead due to the execution of interface operations. This section demonstrates the approximate cost of using the DADT protocol and some simple optimizations to alleviate that cost. The results in this section are based on the implementation of the DADT protocol outlined in Section 3.2.

In approximating the cost of DADT use, it should be made clear that the cost is dependent on the triggers and resources associated with DADT objects. DADTs are highly configurable — down to the level of individual objects — so it is impossible to make accurate predictions about the upper bound on their operating costs.

A universally applicable configuration for DADTs is the empty configuration, that is, a DADT with no resources or triggers associated with its class or its objects. The current representation has only one member, minimizing the complexity associated with applying actual interface operations. In effect, the DADT becomes an elaborate wrapper for an existing data type. This configuration provides a minimum operating cost and will provide an impetus for other optimizations, specific to particular dynamic ADTs.

Recall that directives play an important role in the protocol in that they dictate whether parts of the protocol are executed or not. By setting the standard directive to *false*, it is disabled for DADT objects within the scope of the directive in which it is set. The default situation is that the standard directives are set to *true* in the global directive environment. However, lookup still proceeds from either the local directive environment or the class directive environment to the global. Therefore, the protocol can be made faster — even if it is still enabled — by setting the standard directives to *true* in the scope the lookup is initiated from. This means that any measurement of execution overhead must take into account the values of the standard directives at different scope levels in the directive environment.

All tests that follow were run on a 17” MacBook Pro running Mac OS X 10.5.3 with a 2.16GHz Intel Core Duo processor. The Common Lisp implementations used are International Allegro CL Enterprise 8.1 (ACL)[4], Steel Bank Common Lisp 1.0.17 (SBCL)[5] and GNU CLISP 2.43 (CLISP)[6].

Measuring overhead

Measuring the overhead will be done by defining a simple ADT called `fnum` that is a wrapper for numbers. Two operations are defined independent of each other:

[4]`http://www.franz.com/products/allegrocl/`

[5]`http://www.sbcl.org/`

[6]`http://clisp.cons.org/`

```
(defmethod succ ((x fnum)) (1+ (val x)))
(defmethod add ((x fnum) (y fnum)) (+ (val x) (val y)))

(defclass fnum ()
  ((val :initarg :val :initform 0 :accessor val)))

(define-dadt-class dnum ()
  ()
  (:default-rep fnum))

(define-dadt-operation succ dnum ((x dnum)))
(define-dadt-operation add dnum ((x dnum) (y dnum)))
```

Figure 3.7: A simple number ADT and DADT for measuring overhead

successor and addition. Both return Lisp numbers and do not alter their internal state nor create new instances of the data type. A DADT is defined for `fnum` with the empty configuration called `dnum`. Code for these data types is found in Figure 3.7.

Timing data is obtained by running each interface operation one million times on the same parameters with code similar to the following, where t is either `fnum` or `dnum`:

```
  (loop with x = (make-instance 't)
        repeat 1000000 do (succ x))
```

When t is `dnum`, the directive environment is set up appropriately before the code is evaluated. Two parameters are required for the addition operation, in which case the `x` is used for both parameters. All code is compiled with the default compiler settings. These settings have the same settings for safety, space and speed, varying only in their debug settings.

Table 3.1 shows the execution overhead of the DADT protocol for interface oper-

Standard directives			ACL	SBCL	CLISP
Local	Class	Global			
null	—	*true*	5.0	4.7	23.8
null	—	*false*	1.7	1.4	6.7
true	—	—	3.2	2.4	19.8
false	—	—	0.6	0.1	4.9
—	*null*	*true*	7.6	8.3	36.7
—	*null*	*false*	1.5	2.1	8.5
—	*true*	—	7.6	7.8	35.5
—	*false*	—	1.3	1.5	7.2

Table 3.1: Execution overhead of the DADT protocol given in microseconds.

ations with the standard directives found in the various scopes for three Lisp implementations. The times are given in microseconds. An entry of *null* for the standard directives means the directive environment in the given context has no entry for the standard directives, thus forcing lookup in the next directive environment. An entry of '—' means the directive environment is never consulted.

The data shows that the overhead has potential to be significant over time. Perhaps more importantly, however, it shows that the overhead can be greatly reduced. When the extra stages are disabled, the overhead is a function of the cost of looking up the standard directives and the implementation of the DADT interface operation. This suggests that the overhead incurred by the protocol can be kept constant if the extra stages — the running of triggers, the initial measurement stage and the final measurement stage — are not executed.

Stabilization

Given that local and class directive environments are accessible by other elements of the program, we can define triggers that bind the standard directives to *false* in these environments. This would facilitate optimizations for specific objects and operations that manifest themselves dynamically.

Disabling the protocol via standard directives for individual DADT objects or

	ACL	SBCL	CLISP
`compute-s-prime*`	0.49	0.40	0.87
`compute-s-prime`	3.06	3.08	11.21

Table 3.2: Performance of functions to compute S', given in seconds.

classes of DADT objects is called *stabilization*; it can be useful when enough is known about the object's structure to maintain it without requiring attention. If a reasonable predication can be made about the object's future based on its history, then we can effectively render the object inert with respect to the protocol. Determining the situations when to stabilize is dependent on the nature of problem in which the DADT is being used. Thus, creating general optimizations for stabilizing in specific directive environments is not undertaken in this work. However, to demonstrate the possibilities of this optimization approach, we present a basic programming problem with a solution using the Set DADT from Section 3.2.

The problem is as follows. Given a file containing numbers and strings (evenly distributed), create a set S of the elements in the file, then compute S' where $S' = \{x|x \text{ is a string} \wedge |x| \in S\}$. Two solutions to the problem are provided in Figure 3.8. The function `compute-s-prime` uses the Set DADT; `compute-s-prime*` uses the underlying set structures that the Set DADT uses[7]. Note that two sets must be created to hold the different types in `compute-s-prime*` because the membership test required for strings and numbers is different, making the code more complex.

The performance of the functions are given in Table 3.2. Times are given in seconds and the file contained one hundred thousand items, not necessarily distinct. The same input file was used for all tests.

The overhead of the triggers and resources of the Set DADT is apparent in the data. As defined, the Set DADT has only one resource per set object and three

[7]The representation is taken from Gary King's CL-Containers package. See `http://common-lisp.net/project/cl-containers/`

```
(defun compute-s-prime (file)
  (let ((s (make-dset)))
    (ignore-errors (loop (insert-element s (read file))))
    (filter s #'(lambda (e) (and (stringp e)
                                 (member-p s (length e)))))))

(defun compute-s-prime* (file)
  (let ((str-set (make-set 'associative-array :test 'equal))
        (num-set (make-set 'associative-array :test 'equalp)))
    (ignore-errors
      (loop for elem = (read file)
            if (stringp elem) do (insert-element str-set elem)
            else do (insert-element num-set elem)))
    (let ((f (make-set 'associative-array :test 'equal)))
      (iterate-set str-set #'(lambda (e)
                               (when (member-p num-set (length e))
                                 (insert-element f e))))
      f)))
```

Figure 3.8: Functions to compute S'

triggers. With only two set objects created (one object is created by the `filter` function) and noticeable overhead, it is easy to see that a program with many set objects is likely to perform very poorly. This suggests that stabilizing set objects would be beneficial.

In defining a trigger for stabilizing DADT set objects, we will take into account functional programming style. Assuming the basic notion of functional programming is used — that is, new objects are created when necessary instead of existing objects being changed — then we can expect deletion to be rare. Furthermore, consider the mechanics of the compaction trigger described earlier, which reacted to deletion operations. If the compaction trigger is not used, it does not affect the semantics of the structure. That is, by ignoring deletions, the structure of the set still supports operations properly. Given that insertions will make up the vast majority of operations and that the situations presented by deletions can be ignored, we focus solely on insertion operations.

The insertion trigger is sensitive to the types of the data elements. We argue that we can make a reasonable guess about the future of a set object given some number of insertion operations. Suppose we anticipate a maximum of n types to be contained in a set object at any given time. If x insertions are performed on the object and there is a $1/n$ chance for each of the n expected types to be inserted, then there are n^x possible sequences of insertions, where each element of the sequence is a type. If x is large enough, the chances of seeing all n types in a sequence of insertions will be very close to 1.

In the case of the S' problem, we only expect to see two types. In any sequence of x insertions, there are only two configurations in which only one type appears: when all the types are the same. Thus, the probability of seeing both types is

$$\frac{2^x - 2}{2^x} = 1 - \frac{2}{2^x} = 1 - \frac{1}{2^{x-1}}$$

```
(defmethod measure-resource-initial ((res call-counter) rep op args)
  (when (eql 'insert-element op) (incf (counter res))))

(defmethod reset-resource ((res call-counter))
  (setf (counter res) 0)
  res)

(defun stability-trigger (dset op args collected)
  (with-resources ((ccount call-counter))
      (dadt-resources dset)
    (when (<= (or (lookup-directive :stability-threshold)
                  most-positive-fixnum)
              (counter ccount))
      (enact dset '(dadt-protocol nil))
      t)))
```

Figure 3.9: A trigger for the Set DADT to stabilize set objects

If we choose $x = 16$ we get a probability of 0.9999695. Since the file itself is randomly generated using the random number generation scheme of the Lisp implementation, this suggests the distribution of types in the files is even. Thus, we can expect to see two different types in any sequence of sixteen elements processed, resulting in a change in the current representation. Therefore, we will adjust the Set DADT to stabilize set objects if no changes have been made to the structure of the current representation for sixteen insertion operations.

Code defining a resource and a trigger to to do this is given in Figure 3.9. Note the `reset-resource` is used to reset the counter in the event the structure in the current representation changes. The macro `with-resources` extracts the resource of type `call-counter` from the resources of the set object and binds it to `ccount`.

	ACL	SBCL	CLISP
`compute-s-prime*`	0.49	0.40	0.87
`compute-s-prime`	3.06	3.08	11.21
`compute-s-prime` with stabilization	0.73	0.62	2.62

Table 3.3: Performance of computing S' with a stability trigger, given in seconds.

The call to `enact` adjusts the local directive environment of the set object and binds all the standard directives in the environment to `nil` (*false*).

The efficacy of the trigger is shown in Table 3.3, which is the data of Table 3.2 with the additional data from running `compute-s-prime` with the stability trigger. The data was obtained by running the function on the same input file used to measure overhead, as described above.

We can see that the stability trigger drastically reduced the running time of `compute-s-prime` although it still takes between 1.5 and 3 times longer than `compute-s-prime*`. This suggests that stability triggers are useful in situations where performance of DADTs is a consideration. However, defining effective stability conditions may require some knowledge of the input. In the example, we took advantage of the fact that we knew there were only two input types defined.

The data indicates that stability triggers are probably most useful on long-lived objects and of limited use for many individual objects due to the excessive overhead. Managing objects at the individual level at runtime is interesting in some cases, but it is more likely that we want to use the profiling capabilities afforded by DADTs to learn something about a collection of DADT objects and apply it to existing code. Given the overhead of DADTs at the level of individual objects and difficulty in writing general stability triggers, DADTs may be better realized as meta-programming tools used in development. The code generated by the constructs could be used to specialize the program, eliminating the use of the DADT.

3.4 Summary

We have described dynamic abstract data types, an abstract data type whose instances can monitor their usage and adjust their representation taking into account context. This is done by augmenting the instances of the type with metadata and enabling the interface operations for the type to access it and act on it. The use of DADTs was described in the form of a protocol that imposes certain restrictions on how DADTs must behave.

This was followed by a simple demonstration of a DADT for a set data type. Components of the Set DADT were defined in accordance with the DADT protocol and the implementation was analyzed. The analysis showed that there is considerable overhead in the use of DADTs. We then showed one way of reducing the overhead by disabling the protocol when certain situations arise. However, this approach tends to be sensitive to knowledge of the input. The conclusion is that DADTs are likely effective for dynamic environments where reflection is used to collect information about a group of related objects and provide information about them so that specialized versions of the program can be produced, eliminating the DADT from the program.

Chapter 4

Specializing types and operations

This chapter demonstrates the utility of DADTs defined in the previous chapter by defining a type for collections that represents the union of other types whose interfaces intersect[1]. Thus, the application of an operation may be ambiguous. Such ambiguity is introduced deliberately to provide for a single abstraction that encompasses a wide variety of structures and operations. As a result, behaviour specification can focus on manipulating data instead of creating structure to manipulate data.

In particular, if an operation on a collection is ambiguous, the ambiguity is resolved interactively. When the ambiguity is resolved, the context of the execution is remembered. After execution has finished, the contexts are used to generate specialized versions of collections. The novel aspect of this specialization is that they are based on location within the source code. Furthermore, the code generation is done interactively so that it can be approved before replacing elements of the program. The result is that we have a code generation scheme that does not involve explicit source code annotations.

Section 4.1 describes a way to specify program locations, which are used extensively for the specialization of operations. They provide a mechanism for connections to be made between the text of the program and the runtime.

[1]The work in this chapter is an extension of a paper presented at the 5th European Lisp Workshop at ECOOP 2008 [91], co-authored with Mark Daley and Stephen Watt.

A DADT for collections is described in Section 4.2. Code generation based on the specializations obtained through a Collection DADT is described in Section 4.3. Section 4.4 describes an advanced version of a Collection DADT that captures a different context to generate more specific code. A summary of the results is found in Section 4.5.

4.1 Lexical place identifiers

The context in which an operation takes place is partially defined by where in the code the operation occurs. Here, location refers to a location in the program text. Lexical place identifiers are used in contexts to identify source code expressions.

A *lexical place* is an expression in the program text. A *lexical place identifier* (LPI) is a unique label for a lexical place. If two LPIs are the same, they denote the same expression in some program. Note that two expressions with the same contents — that is, the string representation of the expressions have the same characters in the same order — could have different lexical place identifiers. Lexical places are subdivided into two classifications: *use places* and *creation places*.

A use place designates a place in which an object is passed as an argument to a function. For example, if the form `(f a)` has the LPI x, we would say that `a` is used at lexical place x. Note that this excludes `f` from the definition, since `f` is not an argument.

Creation places are lexical places that signify the application of a distinct function that returns objects of a specific type. Put plainly, creation places are places that return values of interest to our analysis. The idea is that objects created at a designated place are related by the fact they were produced by the same expression. Objects can be tagged with their creation place to create groupings that are finer than that ascribed by their designated type, but coarser than individual objects.

(a)
```
(let ((arr (make-array 5)₁))
  (loop repeat 5
        do (vector-push (random 100) arr)₂)
  (values arr (count-if #'evenp arr)₃))
```

(b)
```
(defun wrapper () (make-array 5)₁)
(mapcar #'make-array (list 5 10))₂
```

Figure 4.1: Examples of creation and use places.

For an object O, we denote that object tagged with a creation place p as O_p. The set $\mathcal{T}_p$ consists of all objects tagged with creation place p and is known as the *creation type* p. We say that an object created at place p is of type $\mathcal{T}_p$. The set $\mathcal{U}_p$ is the set of use places of which objects of type $\mathcal{T}_p$ are used during the execution of the program; it is used in the analysis of a Collection DADT to generate code.

An example is provided in Figure 4.1a. We'll consider `make-array` to be a designated function for the creation of objects that are of interest. The lexical place 1 marks the form that is the application of `make-array` and is, thus, a creation place. The type of objects created at place 1 are arrays, but also are of creation type 1 or $\mathcal{T}_1$. Lexical places 2 and 3 mark the use places of the program. The forms are function calls whose arguments are of interest. In this example, $\mathcal{U}_1 = \{2, 3\}$.

A consequence of treating creation places as types is that *type becomes an inherent property of the code describing the behavioural aspects of a program*; creation places assert that objects created at the same place are considered to be the same type. This makes the placement of creation places important for differentiating between types and can lead to unintended type classifications.

For example, Figure 4.1 (b) contains two examples of creation types at a higher level of abstraction than may be desired. Creation place 1 occurs within the function `wrapper`. Thus, any call to `wrapper` will return an object of type $\mathcal{T}_1$. If `wrapper` is used throughout the program instead of `make-array`, all the array objects will be considered to be the same creation type. (This problem is alleviated if `wrapper` is made to be the designated function for the creation of arrays.) Creation place 2 demonstrates issues that can arise from using a designated function as an argument. Since the form calls `make-array`, we designate it as a creation place. However, it is really a call to `mapcar`, which calls `make-array`. Again, both arrays will be of type $\mathcal{T}_2$ even though this may not be what we want.

4.2 A Collection DADT

We now present an example of an abstract data type that encompasses many possible types and operations, providing the capability to construct a particular type and behaviour for its instances over time. Known as a *Collection dynamic abstract data type* (Collection DADT), it is the union of a variety of other abstract data types that act as containers. A Collection DADT demonstrates that explicit type definitions can be omitted from source code and partially deduced from program behaviour. Furthermore, operations dependent on those type definitions can be defined when necessary in a fluid manner.

With a Collection DADT, data becomes the primary focus over that of structure. In this sense, code is written in a way that passes around data that happens to have structure, rather than structures that happen to have data. To put it another way, instead of making structures that are to be populated with some data, the data is treated as a single entity on which structure is imposed when necessary. A single data object can then stand for multiple structures that contain the same data. This provides a mechanism for simpler creation and initialization of structures, and parallel processing for those structures in a single operation. Also, the set of managed structures is dependent on both the parameters provided when an instance is created and the operations performed on it. Thus, its internal structure can be changed by altering either its creation or use.

Ultimately, code written using a Collection DADT is incomplete in that there is not enough specified to properly execute each operation. During execution, ambiguous elements are clarified by consulting the environment, which includes the user. Thus, using a Collection DADT interactively defines elements of the program.

Terminology related to a Collection DADT is presented below. As the name suggests, a Collection DADT is implemented using DADTs. The reader is directed to Section 3.1 for terminology related to DADTs. Following the necessary terminology, we describe the interface then the semantics of the interface operations. Lastly, an example is given to demonstrate the principles.

Terminology

The set of abstract data types making up a Collection DADT is referred to as $\mathcal{A}$. Elements of $\mathcal{A}$ are called *sub-ADTs* and are written in bold lowercase letters from the beginning of the alphabet: $\mathbf{a}, \mathbf{b}, \mathbf{c}$ and so forth. Note that sub-ADTs are types.

The current representation of an instance α of a Collection DADT may be referred to as $\mathcal{C}_\alpha$. The subscript is omitted when the context is clear. Each element $s \in \mathcal{C}_\alpha$ is known as a *structure of the instance* α and is of type $\mathbf{a}$ for some $\mathbf{a} \in \mathcal{A}$. Unless otherwise noted, it is assumed that for any two structures $s_1, s_2 \in \mathcal{C}_\alpha$, the type of s_1 is different from s_2. This means that there is at most one instance of any type $\mathbf{a} \in \mathcal{A}$ in the current representation for an instance of a Collection DADT.

Interface

A Collection DADT is an amalgamation of three abstract data types: PRIORITY QUEUE, SET and ASSOCIATIVE ARRAY. These make up the elements of $\mathcal{A}$. Each ADT is a traditional container, that is, an entity that holds items for the purposes

PRIORITY QUEUE	SET	ASSOCIATIVE ARRAY
`empty?`	`empty?`	`empty?`
`size`	`size`	`size`
`iterate-elements`	`iterate-elements`	`iterate-elements`
`insert`	`insert`	`insert`
`extract-min`	`random-item`	`item-at`
`peek-min`	`member?`	`delete-item-at`
	`delete-item`	

Table 4.1: Structures and operations of $\mathcal{A}$ for a Collection DADT.

of retrieval in some specific manner. We briefly describe the semantics of each ADT with details pertinent to their use in a Collection DADT.

PRIORITY QUEUE requires a function, when an instance is created, that defines a total order on the elements it is to contain.

SET also requires a function when an instance is created. This function is used to test for membership and is called the *membership test*.

ASSOCIATIVE ARRAY acts like a hash table in that the index for elements is computed from the elements added. Thus, insertion into an associative array does not require a key/value pair. Instead, it only takes a value and the key is computed from that value. When an instance is created, the function to compute the key from a value must be provided. Elements are retrieved by using keys.

The interfaces for each of the sub-ADTs is given in Table 4.1. Only the names of the operations are provided; however, it should be noted that each operation with the same name has the same signature modulo type changes. Consequently, an application of `insert` for an instance of SET can be turned into an application for an instance of PRIORITY QUEUE by substituting the instances. Most operations are self-explanatory, but we do note the semantics of two operations for clarification: The `peek-min` operation returns the minimum element of a priority queue, but does not remove it; `random-item` chooses a random element from a set, removes it and returns it.

From these interfaces we get the interface for a Collection DADT which is the union of the interfaces in Table 4.1. Each operation takes an instance of a Collection DADT as its first argument. This is the significant argument (see Section 3.1) for the interface operation.

Collection DADT instances are created with the function `make-collection`. It takes keyword arguments that supply the parameters for the creation of sub-ADTs. If such a parameter is supplied to `make-collection`, a structure of the appropriate sub-ADT type is added to the current representation of the instance immediately. If no keyword arguments are supplied, a collection is created with an empty current representation.

Forms indicating calls to interface operations are taken to be use places and forms indicating calls to `make-collection` are taken to be creation places.

Finally, it is important to note that for operations common to the sub-ADTs, a Collection DADT does not distinguish between the different versions of those operations in its interface. This reduces the complexity of the interface as it pertains to the actual writing of code, but raises the complexity with respect to the execution of the operation. It is this unification property that provides a way for instances of a Collection DADT to represent a collection of data to be used in different ways. Each instance holds multiple structures, more than one of which may be intended for use with a single operation. This permits the programmer to capture the essence of an algorithm, leaving the details of precise structural manipulation for later. Ambiguity in operations is addressed at runtime.

Semantics of interface operations

Given that the interface allows for the same operation to be applicable to multiple structures with in instance, interface operations may not be well-defined when they are called. The proper application of actual interface operations requires that the context of the operation be considered and a suitable configuration for the actual interface operation be obtained.

A *context* is an n-tuple of values taken from the actors, environment and system involved in the computation (see the discussion in Hirschfeld et al. [31]). To briefly summarize, actors are entities that make requests for system behaviour, the system provides the behaviour by performing some action and the environment is anything external to this relationship. Informally, a context can be considered a collection of conditions that describe a situation. For example, a context might consist of $(x = 42, P(\texttt{eql}))$ where $P(f)$ is the predicate "the set S uses the function f to test for membership", indicating the context that x has the value 42 and S uses the function `eql` as its equality test. The values making up the context are determined before an actual interface operation is applied.

A *configuration* is a description of how to apply the interface operations of the sub-ADTs on the current representation. This includes what structures are to be used and in what order they are to be processed. It may also include other information pertinent to the operation, such as whether the structure is to be created and how to initialize it. A configuration can be thought of as a required parameter for interface operations that is automatically added as an argument by the runtime system when an interface operation is called.

The actual interface operation is the application of one or more of the interface operations of the sub-ADTs to a subset of $\mathcal{C}_\alpha$ for some Collection DADT instance α. For example, if the `size` operation is called with the significant argument α, the trigger on α will determine (or obtain) the configuration for the application of `size` in the current context. The configuration may say to apply `size` to the instances of PRIORITY QUEUE and SET in the current representation, which may or may not make up its entire contents. Calling `size` on these elements constitutes the actual interface operation.

Note here that contexts and configurations are the disambiguating elements of interface operations and the trigger is free to query the user to obtain information in order to proceed. It is expected that user input will be used to clarify situations in most cases.

No specific procedure is given for determining the context because its definition is quite broad and the particulars of the context can affect the configuration. It also affects analysis and code generation. Strictly speaking, the context can be empty and interface operations still can be properly applied. In this case, a configuration for an operation could be obtained each time an operation is called.

Similarly, the definition of configuration is open-ended to allow for freedom in collecting information to execute actual interface operations. Despite the lack of preciseness in the definition, we will define a default trigger using a straightforward context and configuration to ground the discussion.

For the default Collection DADT trigger (herein referred to as simply the default trigger), the context required is a pair (p, q) where p is the creation place of the significant argument and q is the use place of the interface operation. It is assumed that the significant argument is tagged with its creation place and the use place is available in the dynamic environment of the computation.

Configurations for the default trigger are a sequence made up of elements from $\mathcal{A}$. The sequence describes the elements of the current representation the actual interface operation is to use and the order in which they are to be processed. For example, a configuration of $[\mathbf{b}, \mathbf{a}]$ says that for some Collection DADT instance α with a current representation $\mathcal{C}_\alpha = \{s_1, \ldots, s_n\}$, the actual interface operation is to process the structures in the sequence $[s_i, s_j]$, where $1 \leq i, j \leq n$, s_i is of type $\mathbf{b}$ and s_j is of type $\mathbf{a}$.

The trigger also maintains a *context-configuration mapping* that associates configurations with contexts for future use. When the trigger is invoked, it uses the current context to lookup a configuration. If the configuration is known, that is the configuration used, otherwise it must obtain one.

Obtaining a configuration is done by examining the structures applicable for the interface operation. If the operation is only applicable to a single structure, then the type of that structure makes up the configuration. If there is more than one applicable structure, the user is queried to clarify what structures are to be used. The context is used to describe the situation to the user, namely where the significant argument originated and where the operation is taking place.

Word puzzle example

To solidify the definition and use of a Collection DADT, we present an example of the development of a small program to solve a word puzzle.

A word puzzle is given by an arrangement of cells that hold pieces. Both the cells and the pieces have a specific orientation that does not change (that is, pieces and cells do not rotate). The goal is to place each piece in a cell so that every cell contains one piece and every horizontal (left to right) and vertical (top to bottom) path through the cell arrangement forms an English word.

Figure 4.2 contains an example. The solution consists of the words BUY, AZURE, CORNS, TOY making up the horizontal paths, and the words ACT, ZOO, BURY, URN and YES making up the vertical paths.

A puzzle is given as a set of cell descriptions, each cell having horizontal or vertical orientation with a unique identifier. Another set provides the connections between cells using their identifiers.

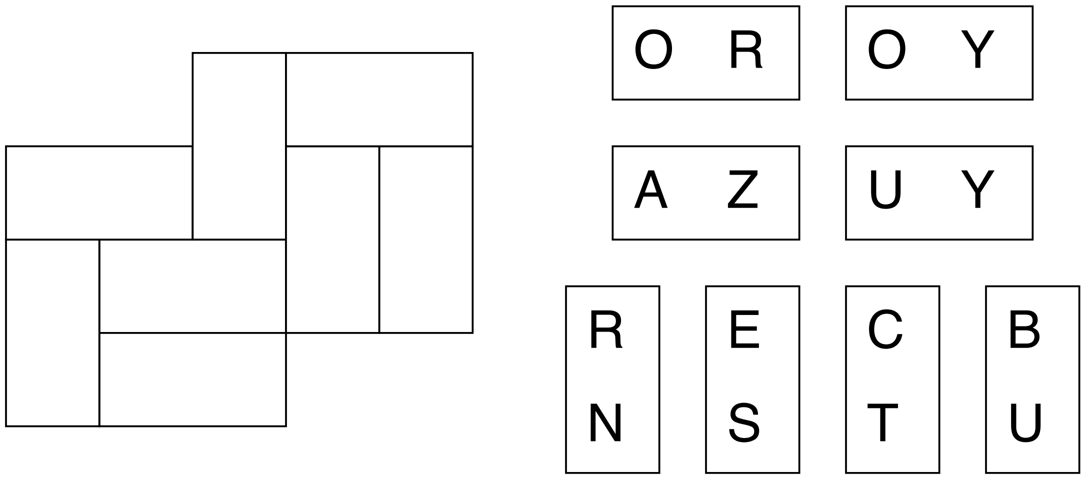


Figure 4.2: A word puzzle for demonstrating use of a Collection DADT. This example is copyright 2008, Fraser Simpson, taken from *The Walrus*, volume 5, issue 5, p. 96, June 2008.

To solve the puzzle, we use a simple backtracking search algorithm: Choose an empty cell and find a piece that fits in the cell that does not violate the constraints, that is, if the piece would complete a word, that word must be valid. If no such piece can be found, backtrack to the last cell filled and try a different piece. Continue until either all the pieces are placed or no valid configuration can be found.

Studying the problem description, we see that cells can be referred to by identifiers, suggesting that an association of identifiers to cells could be useful. Cells are also an integral part of the search algorithm and how they are selected will have an effect on the algorithm's efficiency[2]. We may want to hold empty cells in a set and randomly select them, or we may impose an ordering so that we may select the "best" one during the search. Using a Collection DADT, we can write the algorithm to support all these structures without having to provide the specifics until we run it.

Code to create and solve a puzzle is given in Figure 4.3. Creation and use places

[2]We will ignore the fact that puzzles are probably small and efficiency is likely of minor concern.

```
(defun make-puzzle (&key horizontal-cell-ids vertical-cell-ids
                         connections)
  (let* ((cells (make-collection :index-by #'id :test #'eql)$_1$)
         (puzzle (make-instance 'puzzle :cells cells)))
    (loop for id in horizontal-cell-ids
          do (insert cells (make-instance 'horizontal-cell :id id)))$_2$
    (loop for id in vertical-cell-ids
          do (insert cells (make-instance 'vertical-cell :id id)))$_3$
    (loop for (out a in b) in connections
          for from = (item-at cells a)$_4$ and to = (item-at cells b)$_5$
          unless (and from to) return nil
          do (connect-edges out from in to))
    puzzle))

(defun solve-puzzle (puzzle pieces)
  (let ((next-cell (unless (empty? (cells puzzle))$_6$
                     (random-element (cells puzzle))$_7$)))
    (unless next-cell (return-from solve-puzzle puzzle))
    (loop for piece in pieces
          if (and (place-piece next-cell piece)
                  (configuration-legal-p puzzle)
                  (solve-puzzle puzzle (remove piece pieces)))
            do (return-from solve-puzzle puzzle)
          else do (withdraw-piece next-cell))
    (insert (cells puzzle) next-cell)$_8$
    nil))
```

Figure 4.3: Code for solving the word puzzle.

$(1, 2) \rightarrow$ [SET, ASSOCIATIVE ARRAY]
$(1, 3) \rightarrow$ [SET, ASSOCIATIVE ARRAY]
$(1, 4) \rightarrow$ [ASSOCIATIVE ARRAY]
$(1, 5) \rightarrow$ [ASSOCIATIVE ARRAY]
$(1, 6) \rightarrow$ [SET]
$(1, 7) \rightarrow$ [SET]
$(1, 8) \rightarrow$ [SET]

Figure 4.4: Context-configuration mapping for code from Figure 4.3.

have been marked for future reference. The full program is not presented as it is not germane to the discussion, although it does contain other use places.

The call to `make-collection` provides the parameters for the sub-ADTs: the indexing function for ASSOCIATIVE ARRAY is `id` (which retrieves the identifier from a cell object) and the test function for SET is `eql`. The object created will contain an instance of ASSOCIATIVE ARRAY and SET. Collections created at lexical place 1 are considered to be of type $\mathcal{T}_1$ and we can see that the set $\{2, 3, 4, 5, 6, 7, 8\}$ will be a subset of $\mathcal{U}_1$.

It is worth reflecting briefly on the use places (lexical places 2 through 8) in the code. The calls to `insert` at places 2 and 3 are meant to populate the instances of both ASSOCIATIVE ARRAY and SET within the current representation of the collection. However, the call to `insert` at lexical place 8 is only intended for use with the instance of SET, as is the call to `empty?` at place 7. Calls to interface operations at places 4, 5 and 6 are not ambiguous.

When this code is evaluated, the trigger associated with a Collection DADT instance must disambiguate the appropriate calls. Assuming the default trigger behaviour, it will query the user to indicate what structures are to be used for the actual interface operation. This configuration is saved and associated with the context so that future evaluations of the lexical places with objects of type $\mathcal{T}_1$ no longer require clarification. The complete mapping of contexts to configurations is given in Figure 4.4.

Given the semantics of the `make-collection` function and the fact that the default trigger does not introduce any elements to a current representation, we know that for all objects α of type $\mathcal{T}_1$, $\mathcal{C}_\alpha = \{\textsc{Associative Array}, \textsc{Set}\}$. The default trigger's work is solely to disambiguate structures for use in actual interface operations. Thus, types of the form $\mathcal{T}_p$ for some creation place p are discernible by examining creation places.

Partial manifestation of type in the behavioural aspects of the program are useful for brevity in code. Separate structural descriptions, to some degree, become unnecessary. In the example, we did not have to say how to allocate and store anything of type $\mathcal{T}_1$. Instead, a general approach was used and an effective definition was created internally. Furthermore, operations applied to multiple structures could be realized with a single interface operation without an immediate indication of what structures were to be used. Thus, behaviour based on type is defined when required. In the end, type definitions and operations based on those types can be excluded from the code, with effective definitions created dynamically as placeholders.

Let us now look at how to alter the code to use a PRIORITY QUEUE to store the cells for use in the `solve-puzzle` function. Doing so will illustrate some of the shortcomings of the default trigger and provide insight into the creation of more comprehensive trigger behaviour.

To put cells in a priority queue, we must decide on a way to rank cells. We will use a basic greedy strategy by saying that for any two cells c_1, c_2, $c_1 < c_2$ if the number of connections of c_1 is greater than that of c_2. That is, if c_1 is adjacent to more cells than c_2, then c_1 is considered a better choice than c_2. The intuition here is that c_1 has more words in the puzzle that depend on the proper piece placement, so it is fitting to tackle them first. The function to compare the connection ranking of two cells is `cell-compare`.

```
(defun make-puzzle (&key horizontal-cell-ids vertical-cell-ids
                         connections)
  (let* ((cells (make-collection :index-by #'id
                                 :prioritize-by #'cell-compare))
         (puzzle (make-instance 'puzzle :cells cells)))
    (loop ... (insert ...) ...)   ;; Add horizontal cells
    (loop ... (insert ...) ...)   ;; Add vertical cells
    (loop ...)                    ;; Add connections
    ;; Add to the Priority Queue
    (iterate-elements cells #'(lambda (c) (insert cells c)))
    puzzle))
```

Figure 4.5: Adding to a priority queue in `make-puzzle`.

It may be tempting to think that we can simply re-run the code provided in Figure 4.3 after changing `random-element` to `extract-min` and providing the necessary keyword argument to `make-collection`. However, this will not work. The problem lies with the calls to `insert` at lexical places 2 and 3. When the cells are inserted into the collection, the connections have yet to be made. Thus, every cell will have zero connections and the choice of cells via `extract-min` will be functionally equivalent to `random-element`.

Solving this problem means that each cell has to be inserted into the collection again, which requires another call to `insert`. One way of doing this is demonstrated in Figure 4.5. Elided code is represented with ellipses.

This solution is unsatisfactory for a number of reasons, not the least of which being that the programmer must keep track of the state of multiple internal structures in order to properly initialize the collection. Furthermore, the code is unintuitive because of its self-referential quality: We are iterating over the cells in the collection to add the cells to the collection.

Using the default trigger, we cannot get around this problem unless we redesign the specification of cells so that when they are created, the number of connections is known. One issue is that the priority queue must be created when the collection instance is created. This is because configurations obtained by the default trigger only discern the structures to which the interface operations of the sub-ADTs are applied. Not enough information is garnered to be able to eliminate the need to "doubly insert" elements into a collection.

Discussion

A Collection DADT represents the union of n abstract data types (the set $\mathcal{A}$) and the current representation of an instance holds instances of the sub-ADTs. The semantics of `make-collection` are such that a current representation can have at most one instance of any sub-ADT. Ignoring the case when no structures are created in a collection, there are $2^n - 1$ possible combinations of $\mathcal{A}$, where $|\mathcal{A}| = n$, that describe a current representation, namely the set $\mathbb{A} = 2^{\mathcal{A}} - \{\emptyset\}$.

Interestingly, each element of this set forms a variant type. Each element $\{\mathbf{a}_1, \ldots, \mathbf{a}_m\}$ in $\mathbb{A}$ where $m \leq n$ is represented by the type $\langle a_1 : \mathbf{a}_1, \ldots, a_m : \mathbf{a}_m \rangle$. Thus, we can view the instantiation of a collection (using the default trigger) at a creation place p as an instantiation of a type from the set $\mathbb{A}$, where that type is $\mathcal{T}_p$.

It is in this sense that interface operations represent many possible operations. Recall that interface operations are use places for a Collection DADT. Thus, a single use of an operation is applicable to many combinations of the elements in the current representation because *the creation places of the significant arguments may differ*.

Roughly speaking, interface operations can be thought of as generic functions whose methods have yet to be defined. If we assume that the application of an interface operation f to an instance e of $\mathcal{T}_p$ applies the appropriate sub-ADT interface operation to the fields of e, then there are finite number of ways to define f. Evaluating the code using a Collection DADT is a way to determine what methods are required and where they are used.

In effect, a Collection DADT and its operations represent a large search space and are a form of genericity. By changing the variant designated at a creation place p, different methods can be used at different places, effectively creating a different program.

4.3 Analysis and code generation

Looking at the definition of a Collection DADT, we see that it creates effective definitions of entities in order to properly execute interface operations. However, it still suffers from the overhead problems described in Chapter 3. In fact, it is worse because the creation of the definition cannot happen without user intervention[3]. Avoiding this overhead can be done by shifting the role of a Collection DADT from a general abstract data type used at runtime to a meta-programming framework to generate code.

The code we look to generate is meant to capture the internal effective definitions created through the trigger based on a Collection DADT instances. Generating this code means that a new program is produced specialized to the configurations provided during the execution. What is interesting about this approach is that the code generation does not involve source code annotations. Instead, the parameters required to guide the code generation are obtained interactively.

Furthermore, the code we want to generate is meant to replace the code employing a Collection DADT in the program. Generated code should resemble what

[3]One practical way around this is to introduce the necessary information into the environment and make it accessible without the need for interaction. Assuming use of the default trigger, this would mean specifying the structures to be processed at each interface operation. Doing so would effectively defeat the purpose of a Collection DADT.

is written by programmers for the purposes of refactoring. Capturing programmer style is not the intent, but providing a mechanism for seamless integration into the workflow of the programming environment is desired.

For this reason, we take the unusual approach of generating the code interactively. Interaction is achieved by presenting generated code using a text editor as the interface. This permits generated code to be adjusted before being substituted into the program.

A discussion of code generation for the default trigger follows. No general procedure for code generation is defined as it is dependent on what information is collected. Notwithstanding the dependencies, any code generation scheme for a Collection DADT must provide valid code describing the type $\mathcal{T}_p$ for a creation place p and code for interface operations at each use place in $\mathcal{U}_p$.

Interactive code generation

Generating code interactively is accomplished by communication between a text editor and the Lisp system. Requests to generate code are initiated by the user in the editor which sends a request to the system to provide the generated code. In the editor, the code is presented to the user and can be edited before inclusion into the program. In the following discussion, we refer to an implementation of interactive code generation using Emacs[4] and the SLIME[5] development environment.

Creation and use places are the focal points of code generation. The definitions created by a Collection DADT describe the types of elements (based on their creation places) and how operations on them are to be carried out (their use places). As a result, code generation is controlled by managing these places in the editor, with

[4]See `http://www.gnu.org/software/emacs/`

[5]The Superior Lisp Interaction Mode for Emacs. See `http://common-lisp.net/project/slime/`. Despite the name, at the time of this writing it is a popular open-source environment for developing Common Lisp programs.

the system associating generated code with creation and use places, and indicating what code is independent of them.

Lexical place identifiers facilitate this communication. The editor generates LPIs that are made available to the system. This is done by having the editor transform the code such that the LPI is available within the dynamic environment of the computation. The system must provide an interface for setting a lexical place identifier in the dynamic environment. Consequently, evaluation of the code must be initiated in the editor to apply the transformation.

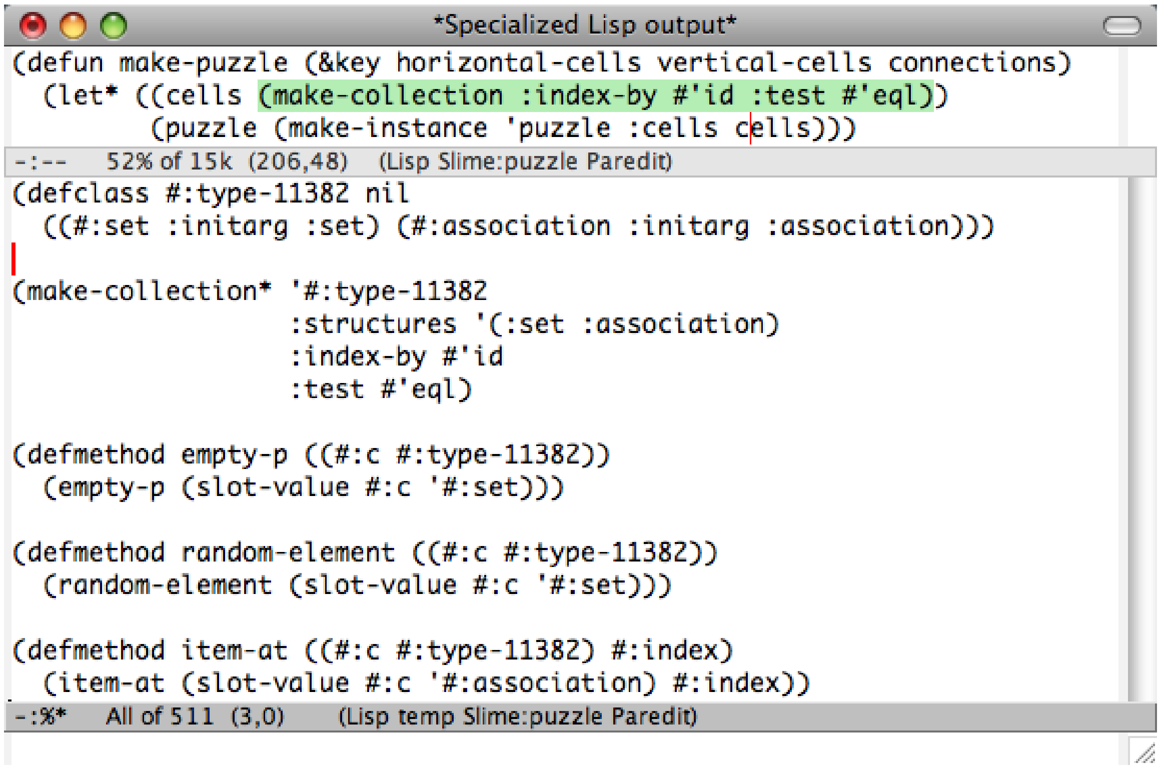
Forms that indicate creation and use places are marked in the editor. Only marked forms are considered by the system, so it is important that all relevant places are marked. (This is much less cumbersome if calls to the interface operations of a Collection DADT are marked automatically by the editor, in addition to the user being able to mark (and unmark) forms.)

To add a lexical place identifier to a marked form, the form is wrapped in a `let` expression that binds a special (fluid) variable to the LPI. Interface operations and triggers assume that the special variable is only bound as a result of code inserted by the editor and is not changed by any other program entity. For example, the call to `make-collection` in Figure 4.3 would be transformed into

```
(let ((*lpi* 1))
  (make-collection :index-by #'id :test #'eql))
```

where `*lpi*` is special variable holding the lexical place identifier of the currently executing form. During evaluation, `make-collection` does not alter or change the binding of `*lpi*` and uses its value to tag Collection DADT instances.

Once the program has been evaluated — both in the sense of actually running it and verifying that it is correct — we can request a specialization. Specializations are forms that represent the definition of and operations for a specific type. Thus, requests are made for creation places and are initiated in the editor.



```
(defun make-puzzle (&key horizontal-cells vertical-cells connections)
  (let* ((cells (make-collection :index-by #'id :test #'eql))
         (puzzle (make-instance 'puzzle :cells cells)))
-:--  52% of 15k  (206,48)   (Lisp Slime:puzzle Paredit)
(defclass #:type-11382 nil
  ((#:set :initarg :set) (#:association :initarg :association)))

(make-collection* '#:type-11382
                  :structures '(:set :association)
                  :index-by #'id
                  :test #'eql)

(defmethod empty-p ((#:c #:type-11382))
  (empty-p (slot-value #:c '#:set)))

(defmethod random-element ((#:c #:type-11382))
  (random-element (slot-value #:c '#:set)))

(defmethod item-at ((#:c #:type-11382) #:index)
  (item-at (slot-value #:c '#:association) #:index))
-:%*  All of 511  (3,0)    (Lisp temp Slime:puzzle Paredit)
```

Figure 4.6: A screenshot of the interactive code generation interface in Emacs. For brevity, the name `association` is used instead of `associative-array`.

When a specialization is requested, the forms are presented to the user in a "staging area", that is, an area that shows the generated code in conjunction with the program without actually inserting it into the program. The staging area provides a way for the user to peruse the code, make edits and see where in the program the generated code would be substituted. The editor employs a variable, local to the editor, known as a hook that indicates how to submit a specialization request to the Lisp system.

A screenshot of a specialization request is given in Figure 4.6. The code being specialized is highlighted in the upper pane of the window and some of the specialization is presented in the lower pane. The form calling `make-collection*` would replace the call to `make-collection` in the original program when the user initiates the request. Note that the name of class is cumbersome and can be replaced in the lower pane before being inserted into the program. Explanation of the generated code is provided later in the discussion.

The interactive approach we have outlined addresses the challenges put forth by Smaragdakis regarding code generation as a tool for use in development [73]. Specifically, he calls for code generation tools to use unobtrusive annotations, generate code that is idiomatic for the target language, and exhibit openness and configurability.

Our approach does not use any textual annotations that require maintenance and in this sense, meets Smaragdakis' requirement. On the other hand, it does require user interaction. The shift from textual annotations to user interaction reflects the idea of using a Collection DADT for exploring the design space of a problem by writing prototype programs. Smaragdakis suggests that to facilitate discreet annotations, they should be found in separate files or in source code comments. Given the transient nature of code for a Collection DADT, we felt it was more suitable to avoid a textual representation of the elements needed for code generation.

Openness and configurability is immediate, given use of the DADT. Triggers are used to collect the information for generating code and are easily substituted. Code generation is extensible by way of a hook used in the editor to request a specialization. Combined with a trigger, this provides enough customization to write entirely new code generation schemes.

Naturalness of the generated code is addressed in the sections that deal with the generated code. However, in Figure 4.6, we can see that the code resembles standard Lisp style. The generated symbols (those prefixed with "`#:`") present the biggest difference between human-generated code. Generated symbols are used to avoid conflicts pertaining to Common Lisp's package system. The most confusing symbol is the name of the type, but devising a suitable name for types was deemed to be a problem outside the scope of the work.

Analyzing the default trigger

Recall that an entry in the context-configuration mapping for the default trigger is of the form $(p, q) \to [\mathbf{a}_1, \ldots, \mathbf{a}_n]$, where p is the creation place of the significant argument of the interface operation performed, q is the use place of the interface operation performed, and $\mathbf{a}_i \in \mathcal{A}$ for each $1 \leq i \leq n$. Let Φ represent the entire content-configuration mapping with $\Phi(p, q) = [\mathbf{a}_1, \ldots, \mathbf{a}_n]$. We also define $\Phi(p)$ to be the set $\{q \mid \Phi(p, q) \text{ is defined}\}$. Note that $\mathcal{U}_p = \Phi(p)$. Finally, let $s(\Phi(p, q)) = \{\mathbf{a}_1, \ldots, \mathbf{a}_n\}$.

With these definitions, we can determine the set of structures making up the current representation of an instance of $\mathcal{T}_p$. Given a creation place p, the set of structures is

$$\bigcup_{q \in \mathcal{U}_p} s(\Phi(p, q)).$$

Denote this set as $s(\mathcal{T}_p)$. We can represent $s(\mathcal{T}_p)$ using a class with a slot for each element.

Generating the code for $s(\mathcal{T}_p)$ is straightforward. A template of the form

```
(defclass t () ((e1 :initarg k(e1)) ... (en :initarg k(en))))
```

suffices, where t is a generated symbol for the name of the type, e_i is the name of type $\mathbf{a}_i \in \mathcal{A}$ and $k(e_i)$ is the name of e_i as a keyword.

In conjunction with the definition of the class, code to create instances of the class must be provided. This can be done by generating a call to the standard function `make-instance`, indicating the creation of an object of type t and passing it the initialization arguments for the contained structures. The creation of the contained structures employs the keyword arguments passed to `make-collection`.

Code for the operations is not as simple because of the fact that an operation may not be used in a uniform manner. By uniform manner, we mean that whenever an operation f is used for a type $\mathcal{T}_p$, the configuration is the same. For example, looking

```
(dapply (#:association #:set)
        insert cells (make-instance 'horizontal-cell :id id))


;;; The above would expand into the following
(let* ((#:g01 cells)
       (#:g02 (make-instance 'horizontal-cell :id id)))
  (insert (slot-value #:g01 '#:association) #:g02)
  (insert (slot-value #:g02 '#:set) #:g02))
```

Figure 4.7: Specialized call for an interface operation at a use place.

at Figure 4.4 and considering the code in Figure 4.3, we see that the operations `item-at` and `empty?` are used in a uniform manner, but `insert` is not. (The use of `item-at` will always be uniform because it is only applicable to one member of $\mathcal{A}$.)

When an operation is used in a uniform manner, we can define a method on the generic function for the operation specialized to the type t. The body of the method applies the operation to the appropriate structures in the order found at $\Phi(p, q)$ for some $q \in Q$. Going back to Figure 4.4, the code for `item-at` would be

```
(defmethod item-at ((#:c t) #:index)
  (item-at (slot-value #:c '#:association-array) #:index))
```

assuming that the name of the slot used to hold an instance of ASSOCIATION ARRAY in an object of type t is `#:association-array`.

If an operation is not used in a uniform manner, it must be specialized for specific use places. Since the intent of the generated code is to help maintainability, we should not make use of the lexical place identifiers in the invocation of the operation. Instead, we will employ a macro, `dapply`, that calls the operation on the given slots of the object. An example of a call to `dapply` is given in Figure 4.7.

Analysis and code generation from information collected from the default trigger is relatively straightforward. This is because objects of a type $\mathcal{T}_p$ do not vary beyond their initial creation. Local effects are minimized. The next section introduces local effects and a way to manage them.

4.4 Handling local effects

Use of the default trigger is limiting, as shown with the work puzzle example. The primary problem is that local effects to instances of a Collection DADT are not possible; the kinds of structures making up the current representation are immediate given its creation place.

To provide more flexibility in the use of a Collection DADT, we now define changes to a Collection DADT that permit structures to be added to a collection at different times. This is accomplished by adding a special structure to the current representation to represent the complete collection of elements that can be used to properly initialize new structures. New structures are treated as active layers as defined in context-oriented programming [31], which requires the context and configuration describing operations to capture more information. Finally, we show how to use this information to generate specializations.

Initialization and a master multi-set

Allowing structures to be added to collections somewhere other than the creation place introduces the problem of initialization. As with the word puzzle example earlier, it would be unsatisfactory if initialization required explicit use of the `insert` operation in a contorted fashion. Ideally, a Collection DADT presents the collection as a single entity, using the appropriate structure when required. What we want to do is track the usage of data through the program, using particular structures in different contexts.

This suggests that the trigger should associate the context of an operation with the structures to be processed *and* an initialization scheme. As we saw with the default trigger, configurations are useful for storing information related to code generation. If we provide for a useful way to specify initialization, and limit the parameters involved, the utility of a Collection DADT can be increased without the need for extra code to be written directly.

Looking at the problematic form to add to a PRIORITY QUEUE in Figure 4.5, we see that populating the priority queue required iterating over all the current elements of the collection. While the method of accomplishing this was arguably unattractive, it was consistent with the idea of a Collection DADT. Essentially, structures contained within the current representation of a collection are different ways to view the collection. Thus, initialization of structure based on the contents of a collection involves taking all the elements and putting them into the structure. This pattern will be used for initialization of structures added after a Collection DADT instance is created.

This raises the problem of knowing all of a collection's elements. A collection is the union of all the elements found in the structures making up its current representation. The current Collection DADT maintains separate structures whose composition forms the collection, but each instance alone may not contain all the elements of the collection. To get around this, we need a "master set" that contains all the elements. This can be used to initialize any new structures.

Considering the above discussion, we will change the semantics of a Collection DADT. The current representation will now contain a *master multi-set* that contains all the elements of the collection; its representation will be determined by keyword arguments passed to `make-collection`. The master multi-set will be of a type distinct from any elements in $\mathcal{A}$, even if is made up of only one structure.

Structures making up the master multi-set are kept synchronized. This requires that we understand the semantics of each operation and associate similar operations to work in tandem when required. For example, if the master multi-set consists of a PRIORITY QUEUE and a SET and the `extract-min` operation is performed on the master multi-set, then the `delete-item` operation must be applied to the SET.

The synchronization requirement imposes restrictions on the set $\mathcal{A}$. Let $set(x)$ be the set of elements contained in some instance x where x is of type $\mathbf{x} \in \mathcal{A}$. For any two elements $\mathbf{a}, \mathbf{b} \in \mathcal{A}$ and instances a of type $\mathbf{a}$ and b of type $\mathbf{b}$, each interface operation $f(a, x_1, \ldots, x_n)$ of $\mathbf{a}$ must have a corresponding operation $f'(b, x_1, \ldots, x_n)$ such that after applying f and f', $set(a) \cap set(b) = \emptyset$. Note that f' need not be an interface operation for $\mathbf{b}$. Mapping an operation from one structure to another may require writing other functions to simulate the operation or subverting the abstract interface for reasons of efficiency.

In effect, the master multi-set is another realization of a Collection DADT. Multiple structures are used to represent a single collection with a unified interface. The difference is that each operation applies to every structure. As alluded to above, this can cause performance problems. For this reason, it is encouraged that the master multi-set consist of a single structure. We accommodate this approach by allowing `make-collection` to take no arguments, consequently using a SET with the membership test `eql` for the master multi-set.

Adding structures

Having a master multi-set in an instance of the adjusted Collection DADT provides a mechanism for us to add new structures at different times and properly initialize them. Furthermore, these additional structures do not have to be synchronized with the master multi-set. Additional structures of this nature act as a different view imposed on the master multi-set, presumably temporarily.

Adding a new structure to the current representation can happen any time an interface operation is called. We will change the semantics of interface operations to consider the elements of $\mathcal{A}$ and the master multi-set; the master multi-set is considered as single entity. Only applicable elements — types or structures that support the operation — will be considered.

More formally, let X represent the set of considered elements, f be the interface operation and α be a Collection DADT instance. X is constructed by the following procedure.

1. $X \leftarrow \emptyset$.
2. If the master multi-set M supports the operation, add M to X.
3. For each $s \in \mathcal{C}_\alpha - \{M\}$, if the type of s supports f, then add s to X.
4. For each $\mathbf{a} \in \mathcal{A}$, if there does not exist $e \in X$ such that the type of e is $\mathbf{a}$ and $\mathbf{a}$ supports f, add $\mathbf{a}$ to X.

Note that the type of the master multi-set is not in $\mathcal{A}$.

Configurations will partially consist of a sequence made up of elements from X. The current configuration is obtained by examining X. If $|X| = 0$ an error is signalled. If $|X| = 1$ and its element is a member of the current representation, the configuration is the lone element of X. If $|X| = 1$ otherwise, then an instance of the type contained in X is initialized (see below).

If $|X| > 1$, then the user is queried to indicate what elements of X make up the configuration. If any of elements chosen are types, instances of these types are initialized.

Initialization consists of allocating a structure of a given type and inserting the elements of the master multi-set. The structure is then added to the current representation. This is done before the actual interface operation is applied.

Proper timing of initialization is essential for the intended semantics of adding structures. Consider the following code:

```
(defun print-elements (c)
  (loop for e = (unless (empty? c) (random-element c))
        while e do (print e)))
```

Suppose we initialize a SET at the call to `empty?` and the master multi-set is made up of an ASSOCIATIVE ARRAY. If we initialize the SET each call, the loop will be infinite if the master multi-set is not empty. Conversely, if we only initialize it once for the lifetime of the program, subsequent calls to `print-elements` may not work as intended.

The problem is that we must consider more in the context of the operation. We will use a context-oriented programming approach and say that for certain function calls, the call activates a layer that represents a view of the data. A layer is a first-class entity that groups related behavioural variations [31]. They are used to activate definitions at runtime. It is possible to define layered classes and functions whose behaviour is dependent on the layers that are active at the time they are used. Using layers, we will define a specific context that augments our class and function definitions that persist for the duration of a given function call.

As mentioned, we will only concern ourselves with scope introduced by function calls. Furthermore, the function must be associated with a symbol, that is, we do not consider anonymous functions. The context is now a triplet (p, q, f) where p is the creation place of the significant argument, q is the use place of the interface operation and f is the name of the function most recently activated on the system stack.

If the configuration calls for an initialization for a type a, then we initialize only if the interface operation is the first one to be applied to an instance of type a since any activation of f on the stack. This eliminates the problem for cases such as that presented with the `print-elements` example. Using this approach, the definition of `print-elements` can be implemented as

```
(defun print-elements (c)
  (with-active-layers (some-layer)
    (loop for e = (unless (empty? c) (random-element c))
          while e do (print e))))
```

The `with-active-layers` call is a macro defined in ContextL, a context-oriented programming extension for Common Lisp [15]. This macro provides the explicit context of the operation. Further, we assume that the class for `c` is a layered class, defined to activate a certain slot when `some-layer` is activated, assuming it is not already active. Also, layered methods are defined for `empty?` and `random-element` to work on the appropriate slot in the context of `some-layer`.

While this allows us to add structures based on function scope, it is limited in its power. For example, it does not capture nested scopes correctly.

```
(defun self-cross-product (c)
  (loop with cross = nil
        for x = (unless (empty? c) (random-element c))
        do (loop for y = (unless (empty? c) (random-element c))
                 do (push (cons x y) cross))
        finally (return cross)))
```

The intent here is that we treat the collection as two sets, one in each `loop`, and create a list representing the set $S \times S$, where S is the set of elements in the collection. The call to `empty?` in the outer loop will create a SET within the current representation, but the call to `empty?` in the inner loop will operate on the same structure. The corollary to this is that recursion does not work when initialization is required upon each entry to a function.

Correcting this would require capturing more information regarding context, such as looking at blocks (as in the Common Lisp `block` special form) or capturing static scope in a dynamic context. However, such precision may require extensive user interaction, such as choosing the proper block intended for the scope of the variable. Interaction to this level may be too cumbersome.

Example

As an example of the new behaviour, consider the code for creating and solving the word puzzle, reproduced (with small changes) in Figure 4.8. This time, we will use a PRIORITY QUEUE instead of a SET for choosing cells. Note that we only insert the cells once and the master multi-set consists of an ASSOCIATIVE ARRAY.

When executing the code at use place 2, the trigger will present us with the set

$$X = \{M, \textsc{Associative Array}, \textsc{Priority Queue}, \textsc{Set}\}.$$

Since we only want to populate the master multi-set, we choose M. Similarly for use places 3 through 5.

When we get to place 6, X is the same as above, except this time we choose PRIORITY QUEUE. Since an instance of this does not exist in the current representation, it is created and initialized by adding each element of the master multi-set using the `insert` function. `empty?` is then called on it and execution continues. Place 7 is unambiguous and we again choose PRIORITY QUEUE at place 8, indicating that the operation is applied to the structure in the current representation.

The final context-configuration mapping is given in Figure 4.9.

Specialization

Generating code for operations means taking into account scope and the master multi-set. Since the master multi-set is its own type, we must generate operations for it as well. Recall that the synchronization property requires that each interface operation f for a type $\mathtt{a}$ has a corresponding operation f' for structures of type

```
(defun make-puzzle (&key horizontal-cell-ids vertical-cell-ids
                         connections)
  (let* ((cells (make-collection :index-by #'id)$_1$)
         (puzzle (make-instance 'puzzle :cells cells)))
    (loop for id in horizontal-cell-ids
          do (insert cells (make-instance 'horizontal-cell :id id)))$_2$
    (loop for id in vertical-cell-ids
          do (insert cells (make-instance 'vertical-cell :id id)))$_3$
    (loop for (out a in b) in connections
          for from = (item-at cells a)$_4$ and to = (item-at cells b)$_5$
          unless (and from to) return nil
          do (connect-edges out from in to))
    puzzle))

(defun solve-puzzle (puzzle pieces)
  (let ((next-cell (unless (empty? (cells puzzle))$_6$
                     (extract-min (cells puzzle))$_7$)))
    (unless next-cell (return-from solve-puzzle puzzle))
    (loop for piece in pieces
          if (and (place-piece next-cell piece)
                  (configuration-legal-p puzzle)
                  (solve-puzzle puzzle (remove piece pieces)))
            do (return-from solve-puzzle puzzle)
          else do (withdraw-piece next-cell))
    (insert (cells puzzle) next-cell)$_8$
    nil))
```

Figure 4.8: Word puzzle code with new behaviour

$$\begin{aligned}
(1, 2, \texttt{make-puzzle}) &\rightarrow [M] \\
(1, 3, \texttt{make-puzzle}) &\rightarrow [M] \\
(1, 4, \texttt{make-puzzle}) &\rightarrow [M] \\
(1, 5, \texttt{make-puzzle}) &\rightarrow [M] \\
(1, 6, \texttt{solve-puzzle}) &\rightarrow [\text{PRIORITY QUEUE}] \\
(1, 7, \texttt{solve-puzzle}) &\rightarrow [\text{PRIORITY QUEUE}] \\
(1, 8, \texttt{solve-puzzle}) &\rightarrow [\text{PRIORITY QUEUE}]
\end{aligned}$$

Figure 4.9: Context-configuration mapping for the adjusted word puzzle

b. Interface operations for the master multi-set can simply call the appropriate operations on each slot for the objects representing the master multi-set.

Generating code for the contexts and methods on the operations is accomplished by analyzing the contents of the context-configuration mapping. Consider an entry $(p, q, f) \rightarrow [\mathbf{a}_1, \ldots, \mathbf{a_n}]$ for some $n \in \mathbb{N}$. This says that for objects of type $\mathcal{T}_p$ at use place q, the operation is applied to the elements of the current representation of type $\mathbf{a}_1, \ldots, \mathbf{a}_n$. If we define the structure of $\mathcal{T}_p$ to be a layered class whose base representation consists of only the master multi-set, then we can represent this operation as follows.

Define a layer l and extend the class $\mathcal{T}_p$ to add slots corresponding to types $\mathbf{a}_1, \ldots, \mathbf{a}_n$ in the layer l. For the operation g at use place q, define a layered method for g that acts on the slots found in layer l. Furthermore, it initializes the slots from the master multi-set if required. (Note that the master multi-set will always be present in instance of type $\mathcal{T}_p$ and any extensions.) In the event there are two contexts (p, q, f), (p, q', f) such that the configuration is the same, we do not need to generate layers for both.

As an example, code for the `solve-puzzle` example given above can be found in Figure 4.10. Code for all the methods are not given as they have a similar pattern to `empty?`.

In order to properly activate the layers for a function f, we must wrap the body of the function in a `with-active-layers` form with the name of each layer gener-

```
(define-layer #:solve-puzzle-layer)

(define-layered-class #:t1 ()
  ((mmset :initarg :mmset)))

(define-layered-class #:t1
  :in-layer #:solve-puzzle-layer
  ()
  ((#:priority-queue :initarg :priority-queue)))

(define-layered-function empty? (#:c))

(define-layered-method empty?
  :in-layer t
  ((#:c #:t1))
  (mmset-empty (slot-value #:c 'mmset)))

(define-layered-method empty?
  :in-layer #:solve-puzzle-layer
  ((#:c #:t1))
  (unless (slot-value #:c '#:priority-queue)
    (initialize #:c '#:priority-queue))
  (code:empty? (slot-value #:c '#:priority-queue)))
```

Figure 4.10: Code generated to capture layers for temporary views on a collection.

ated within the scope of f. Formally, for each context (p, q, f), the layers defined by the above procedure must be passed the arguments to `with-active-layers` and wrapped around the body of the function f. This requires the ability to locate the body for f, which was not necessarily specified as a use place.

The disadvantage to this approach, besides the problems outlined earlier, is that we cannot have the layered versions of the functions reside in the same package as the user code, else name conflicts will occur. Thus, the call to `empty?` on a PRIORITY QUEUE must fully qualify the symbol (assumed to be in the package `code`, in this case). Note that the layer `t` is the base layer. Also, a considerable amount of code could be generated. Depending on how layered classes are represented, the definition of the class could be quite large as well.

On the other hand, the advantage to this approach is that the use places do not have to be changed. Overall, this will likely result in fewer changes to user code, apart from the need to wrap the function body with a layer activation.

4.5 Summary

We have presented a novel way to specialize types by generating code based on source code locations without using explicit annotations. Additionally, we have shown a way to realize a data type that represents the union of other data types whose interfaces intersect. The inherent ambiguity of the operations is resolved interactively, which provides a context denoting a marker acting as an annotation for code generation. Additionally, the code is generated interactively so that it can be approved by the programmer before substituting it into the program.

Specialization was done in two ways. The first defined a global class definition of the specialized type. The second defined a layered version of the class that provides a mechanism for viewing the collection in different ways at different times.

Chapter 5

Deriving class hierarchies

Previously, we showed how objects can be used to represent abstractions that are later specialized to specific types. The work presented in this chapter takes a different approach by building type definitions up from basic elements. In particular, we show how class definitions can be omitted from a program, but instances of the class can still be used in the program. Classes are defined over time based on how objects of the class are used, including the composition of slots and the relationship to other classes. The end result is a class hierarchy derived from the use of objects.

This is done by using evidence from the lexical and behavioural elements of the program. Lexical elements include function calls that implicitly define slots in a class and the methods attached to generic functions. Behavioural elements include the act of calling functions and the order in which actions take place; the behavioural elements are captured by augmenting a language runtime to handle certain situations that may normally cause the program to halt with an error.

Furthermore, we discuss the problems involved in collecting reliable evidence to discern a working class hierarchy. The lack of reliable evidence has an effect on the style in which programs should be written to ensure reasonable results.

In Section 5.1, we discuss an evaluation strategy for a language runtime to support class hierarchy derivation, as well as various assumptions that aid in facilitating it. In particular, we discuss the roles intent and convention play in the approach.

Section 5.2 describes an implementation of the evaluation strategy, taking advantage of the Metaobject Protocol for the Common Lisp Object System [38]; a detailed discussion of the issues involved in accurate class derivations is also found here.

An extended example is provided in Section 5.3 that demonstrates the technique can be effective and further discusses the conditions under which it can be expected to produce accurate results. Section 5.4 summarizes the approach and the results.

5.1 Method and assumptions

Laziness is the primary tool we employ for deriving class hierarchies through program behaviour. Elements of the program are known to be lacking definitive structure when we begin execution and are represented in a manner that permits their growth as more information is provided. These elements are formed into working definitions by reacting to various situations over time. We characterize this as *lazy definition*, similar to the notion of lazy evaluation.

The derivation strategy we describe is based on a permissive execution environment, that is, an execution environment that allows actions to be performed even if the situation is ambiguous or potentially incorrect. Loosening the restrictions on the evaluation rules of the environment permits us to learn properties of the elements involved, assuming the changes are properly targeted. We increase the number of situations in which the program will operate without errors by introducing rules that try to collect information that enable it to continue in the regular sense.

For a prognostic example, suppose we expand the ability of a Lisp system to try to resolve undefined functions, before signaling the user, in the hopes of learning something about the data passed to them. We alter the system to examine the name of the function (assuming it has a name) and its arguments, perhaps also considering other elements of the context[1] at the time. It may be that, given this information, we can discern to a reasonable extent what the function is aiming to accomplish and provide a definition to complete the operation.

Clearly, producing correct output is not the primary purpose to the changes described herein. That being said, an effort is made to produce correct output since by doing so more structural information can be obtained. The intuition is that by running more of the program, we will learn more about the class structure. The evaluation rules to derive class structure descriptions are not meant to act as a way to necessarily correct problems encountered during evaluation. Rather, we seek to present a structure that would enable the behaviour exhibited. Enforcing correctness impedes this objective.

A word on correctness: We consider correct to mean producing the results intended by the programmer outside of the desire to provide reasonable structure to the data. Plainly put, we consider the behaviour exhibited by the program to have a purpose beyond simply that of driving our analysis.

Lastly, we assume that the required behaviour of the program is specified. In particular, the algorithms exist that manipulate objects as do the functions or methods implementing those algorithms.

For our purposes, we consider classes as per the definition found in the Common Lisp HyperSpec[2]. The key property of classes is that they do not contain methods. Methods are defined separately from classes, meaning that redefinition of a class need not be concerned with the code for methods. Thus, we can alter the definition of a class independent of the methods which describe its behaviour. The elements of a class that we are concerned with are its structural elements, namely its slot specifiers and class precedence list. The slot specifiers describe slots that are

[1]We use the term “context” here in the general sense, not in the technical sense defined in Section 4.2.

[2]See Section 4.3.

components that hold values; instances of a class are made up of slots. The class precedence list is a list of superclasses describing the relationship of the class to other classes. These are the only two structural elements we are concerned with; others elements are not considered.

Evaluation strategy

Our goal is to observe the behaviour of a program lacking complete descriptions of various classes and propose structure for those classes. Structure consists of the slot specifiers and class precedence list. The only classes we provide structure for are *incomplete classes*. An incomplete class must be designated as such. Any class that is not incomplete is said to be complete.

Information to derive structural descriptions of classes comes from altering the runtime system to handle the following situations in the manner described below.

No class defined. If a class is needed but no class matching the description is defined, the system defines an incomplete class to be used instead. When created, the incomplete class has no slot specifiers and an empty class precedence list.

No slot found. If an attempt is made to access a slot in an instance of an incomplete class and there is no slot matching the description, an appropriate slot description is added to the incomplete class and a slot matching the description is added to the instance. (By access, we mean an attempt to read from or write to a slot.) Every other instance of the incomplete class must be amended with a similar slot before the that instance is accessed in any way. Furthermore, all future instances of the incomplete class are created with the new slot.

No method found. Since the required behaviour of the program is assumed to be specified, methods whose type specifiers match the type of incomplete classes exactly are not ambiguous. However, if the type specifiers do not match exactly and the method call is otherwise legal, the system must come up with a list of classes $(C_1, \ldots, C_n)$ to serve as potential superclasses for the incomplete class — its class precedence list. The order of this list should reflect the relationship between the classes it contains. For single inheritance, the list forms a chain of classes such that for any two classes C_i, C_j with $i, j \in \mathbb{N}$ and $i \leq j$, C_i is likely a subclass of C_j. With multiple inheritance, if C_i is likely a subclass of C_j, then $i \leq j$.

Note that in all the cases, the goal is to facilitate the continued execution of the program. With the first two, we add the required information and continue. With the last one, an attempt is made to infer what methods may be applicable in order to call them.

Intent and convention

Another principle fundamental to our approach is trusting the intent of the operations, meaning that we consider an operation as something that is supposed to take place. Thus, we do not seek to determine whether an action is to occur, but rather devise a way that it can occur. In a larger sense, evaluation takes on the properties of a problem solver above those of a solution verifier.

Language conventions are an important factor in fostering this approach. Conventions are useful for communicating structure through program text. Kent Pitman asserts, "Conventions are important. The reason people use them is that they want others to infer things from [the code]."[3]

For example, consider the following code that creates and allocates an instance of a class `student`.

[3]Personal communication.

```
(make-instance 'student :name "Julius" :dob "1988-02-29")
```

When initialized, the value `"Julius"` is inserted into the slot whose initialization argument name is `:name` (similarly for `:dob`). It need not be the case that the name of the slot matches the name of its initialization argument, but a common convention is that the initialization argument name is the same as the name of the slot (save for the latter being a keyword). If we assume use of this convention, we can surmise that the class `student` has at least two slots: `dob` and `name`. Note that no formal definition of the class is required to ascertain this information.

However, if we do not know if the convention is followed, then determining the mapping of initialization argument name to slot name may be impossible without sufficient context. Furthermore, obtaining sufficient context itself may be impossible. Suppose the initialization argument `:dob` is used for the slot `date-of-birth`. To a system whose concept of a student is limited to that of an object composed of slots, the context required to reliably map `:dob` to `date-of-birth` may take considerable effort. Thus, without leveraging convention, our approach may be impossible.

5.2 Implementation

We now describe an implementation of the evaluation strategy in Section 5.1 and discuss the mechanics involved in building class structure and relationships at runtime. Each of the three situations is described in turn, followed by examples and a discussion of the approach in the case of the latter two.

Defining incomplete classes

When an object is created, we must indicate the class of the object. Denote this class as C. If C exists, then the object is created in the usual fashion. If C does not exist, then we create an incomplete class C_I, represented as a class metaobject of type `incomplete-class`.

Incomplete classes are created when `find-class` signals a condition indicating that the requested class was not found. We wrap `find-class` with a condition handler that preserves the execution stack and creates an incomplete class, associating it with the identifier that indicates the name of the class. The incomplete class is returned and computation is resumed from the point the condition was signalled.

When a class is named as a specializer for a parameter on a method, it must exist. It is implementation dependent whether `find-class` is used to find classes named as specializers. Thus, it is possible that an incomplete class will not be created when the first mention of a class is found as the specializer on a method.

In general, the handler described for `find-class` can be bound in the dynamic environment of the entire computation. Localizing it to invocations of `find-class`, however, reduces the chances of an intervening handler being established.

Defining slot specifiers

Slot specifiers are added to an incomplete class C_I when the program attempts to access a slot in an instance of type C_I that does not name a slot specifier in C_I. Three elements are required for slot specifiers: its name, its initialization argument name and its initial value. The initial value for automatically defined slot specifiers is always `nil`, so the details of slot specifiers are limited to describing the names.

For the purposes of the descriptions below, the slot name and initialization argument name are taken to be the same if they have the same string representation, modulus the colon for keywords. Thus, the symbol `example` has the same name as the keyword `:example`. When a slot name is derived from an initialization argument name, unless specified otherwise, it is always represented as a symbol with the same string representation, but never as a keyword. Conversely, if an initialization argument name is derived from a slot name, it is represented as a symbol with the same string representation, but always as a keyword.

There are three methods by which slot specifiers can be added to incomplete classes: when the object is created, when a slot is accessed directly and when a slot is accessed indirectly using an accessor method. Each of these are described below.

Object creation Instances of objects are created with the standard generic function `make-instance` called with an *initialization argument list* which is used to provide initial values for slots (and other things). This list is a property list $(p_1 v_1 \ldots p_n v_n)$ with $n \in \mathbb{N}$ where each p_i is a property indicator and v_i is a property value ($1 \leq i \leq n$). We leverage the convention that a property indicator is the initialization argument name of a slot with the same name.

A method is added to `make-instance` that runs before an object of an incomplete class C_I is created and verifies that the property indicators in the initialization argument list name slots in C_I. For each property indicator p_i that does not name a slot, a slot specifier with the name derived from p_i is added to C_I with the initialization argument name p_i. This assumes the standard initialization protocol for the creation of objects is used.

Direct access Aside from `make-instance`, slot specifiers can also be created by attempts to access non-existent slots. The general mechanism for this is the standard function `slot-value`, which takes an instance of a class and the name N of the slot to access, possibly with a value v if the access is a write operation. Furthermore, when such an attempt is made to access a slot that does not exist, the generic function `slot-missing` is called.

A method is added to `slot-missing` that is invoked when the instance passed to `slot-value` of an incomplete class C_I and a slot named N does not exist in C_I.

A slot specifier with the name N is created, along with an initialization argument name derived from N. When the access is a read operation, the value `nil` is returned. When the access is a write operation, the value is updated with the value v and v is returned.

Accessor methods Usually, when defining classes in Common Lisp, methods for accessing slots are provided through automatically defined methods. Names for generic functions to hold the methods are provided when the class is defined. The appropriate methods are then created and attached to the generic function. These are known as *accessor methods*.[4]

Accessor methods are not defined for incomplete classes when slot specifiers are created. Instead, accessor methods are defined when they are used. The problem becomes recognizing when the use of an accessor method is intended.

We rely on a convention to determine whether or not a function call should invoke an accessor method. Although accessor methods can have any name — including the ability to name reader and writer methods separately — they tend to have names that are easily derived from the slot name. Four common conventions for naming accessor methods are as follows:

- Use the same name as the slot.
- Prefix the name of the slot with the name of the class followed by a hyphen.
- The name of the slot prefixed with `get-`.
- The name of the slot with the suffix `-of`.

The accessor method names considered consist of these conventions.

[4]Although methods do not, strictly speaking, have names, we call them accessor methods as a short form for an accessor function with the appropriate method attached.

Defining accessor methods at runtime requires that we obtain the name and arguments of the function call in order to determine whether it appears to be a call to an accessor method. Since we assume desired behaviour is specified, we need only examine functions that are undefined.

The dynamic environment of the computation is extended with a condition handler for undefined functions. In order for the accessor method to be defined properly, the condition signalled must contain the name of the function. If the (evaluated) arguments are not supplied in the condition, the handler preserves the execution stack, installs a function f on the supplied name and resumes the computation by calling f; f accepts any number of arguments and has access to its own name. When f is called, it examines the arguments and determines whether its invocation matches that of an accessor method.

There are two invocation patterns that describe an accessor method: one for a reader and one for a writer.

Readers A reader takes one argument that must be an instance of an incomplete class C_I. Let N be the name of the function. Then N must be a symbol.

Writers A writer takes two arguments, the second of which must be an instance of an incomplete class C_I. The name of the function must be of the form `(setf` N`)`, where N is a symbol.

If the attempted invocation of an undefined function matches one of these two patterns, a generic function is associated with the given name. The generic function is of a special type that is distinctive from standard generic functions, either `auto-reader` or `auto-writer`. No methods are every installed automatically on these generic functions. Instead, we extend the standard error mechanism to deal with the case of not finding an applicable method. When the error mechanism is invoked, the slot N is accessed using `slot-value` in accordance with the type of the generic function. Note that this means the method for defining slots using direct access is used to actually add the slot specifier to C_I. For convenience, both an `auto-reader` and `auto-writer` are defined when an accessor method is defined in this fashion.

Example Suppose that `student` is an incomplete class with no slot specifiers. The call

```
(make-instance 'student :name "Julius" :dob "1988-02-29")
```

would add slot specifiers to the slots `name` and `dob` with initialization argument names `:name` and `:dob`, respectively. The instance created by the call to `make-instance` will contain those slots with the values `"Julius"` and `"1988-02-29"`, respectively. Due to the rules of the Metaobject Protocol of CLOS, any other instance of `student` in the system will be updated to contain the `name` and `dob` slots the next time they are accessed.

Assuming `s` is an instance of the incomplete class `student`, the forms

```
(setf (slot-value s 'student-id) 56)
(grade-of s)
```

would add a slot specifiers with the names `student-id` and `grade`. In the former case, the slot is set to the value 56. In the latter case, `auto-reader` and `auto-writer` generic functions are associated with the symbol `grade-of` and the function specifier `(setf grade-of)`, respectively. In the example, the error mechanism will use `slot-value` to read the `grade` slot.

Discussion The choice of `nil` for the initial value is based on convention and pragmatism. In Common Lisp, `nil` is frequently used as a default value. Furthermore, using some other value introduces complications. Suppose the slot specifier is added by a write operation and the value supplied with the operation was used.

Using this value as the default may require copying in order to prevent aliasing. Proper copying would require that copying behaviour be specified [59]. This introduces a dependency that betrays the transparency we are striving for and hence, is not used.

Another approach would be to specify default values for the types of the elements involved. This would require the system to be aware of the different standard types and to provide a hook for describing user-defined types. Again, this introduces a dependency that violates the design principles of the work.

Note that we do not attempt to discern whether a slot is class-allocated (shared) or instance-allocated (local). There are no immediate indications in how slots are accessed to be able to differentiate between the two in either language constructs or conventions.

The patterns describing accessor method invocations have the potential to define functions intended for other purposes. Although we assume that behaviour has been specified, practically speaking, this will not always be the case. In a less ideal environment, we could require a slot matching the name N exist in the incomplete class before defining the accessor method.

Dynamic method selection

Choosing methods when no relationships are known between classes is the most involved aspect of the implementation. The problem is that there is no reliable way to recognize whether one class is a superclass (or subclass) of another.

Inheritance in the Common Lisp Object System states that for any two classes A and B, if A is a superclass of B, then $s(A) \subseteq s(B)$ where $s(C)$ denotes the slot names in the slot specifiers of a class C. This implies that if $s(A) \not\subseteq s(B)$ then A is not a superclass of B. Thus, the only definitive statements we can make about the relationships between incomplete classes and other classes are exclusionary.

For this reason, we say that for any two classes A and B, if $s(A) \subseteq s(B)$ and $A \neq B$ then A is a *potential superclass* of B. However, if A is an incomplete class and B is a complete class, then A cannot be a superclass of B. This is seen by realizing that any slots added to A must also appear in B, but B is already fully specified. Since B is fully specified, we know the class precedence list and the slots of each class in the class precedence list. If A is an incomplete class, any change in A would result in a change in B. However, the definition of B is fixed. Thus, relationships between incomplete classes, as suggested by the definitions, will be based on incomplete information.

One measurement we use is *structural similarity*. Given two classes A and B, their structural similarity is given by the number of slot specifiers in A and B that share the same name. Formally, it is $sim(A, B) = |s(A) \cap s(B)|$. Note that for any three classes A, B, C where A is a superclass of B and B is a superclass of C then $sim(A, B) \leq sim(B, C)$.

Suppose B is a subclass of A. Then B inherits the behaviour of A, meaning instances of B can be passed as an argument to a method m that expects instances of class A. If a call to a generic function is made with an instance of an incomplete class C_I as the i-th required argument, then we look at the set of specializers for the i-th required parameter on the methods of that generic function. This set is known as the *applicable specializers* for the class C_I and is written $\Delta_f(C_I)$ where f is the generic function called.

We take applicable specializers to be classes that are likely related to C_I. Again, this is based on the notion that behaviour has been specified to the desired extent. If two classes are related to each other by behaviour, then the relationship will manifest itself through the applicable specializers. Despite the high chances of false positives — there may be many other classes naming specializers in the same position that are not related, such as with generic functions like `print-object` and `documentation` — we see no other ways to obtain evidence of relationships at run-time.

Also, the issue of the volatility of the slot specifiers of an incomplete class factors heavily into revealing the relationships between classes. New slots can be added any time a function is called, potentially invalidating the current standing of relationships. However, choosing methods relies on knowing something about the relationships between classes. For this reason, relationships are calculated each time a generic function is called involving an instance of an incomplete class.

The class precedence list of a class C is the complete list of superclasses of C in order of precedence. It is the basis by which applicable methods are chosen when a generic function is called. In complete classes, the class precedence list is fixed. In incomplete classes, it is variable. Note that an incomplete class cannot be in the class precedence list of a complete class, by definition. We write the class precedence list for a class C as $cpl(C)$.

We assume here that the program, while running, does not attempt to define complete classes that are based on incomplete classes. This could be accomplished, for example, by constructing a `defclass` form and passing it to `eval`. If this occurs, the behaviour is undefined.

Let f be a generic function called with the positions of instances of incomplete classes at required argument as $i_1, \ldots, i_n$. Denote the incomplete class for the instance at position i_j as C_{i_j}, where $1 \leq j \leq n$. Note that for any two positions i_j, i_k, it may be the case that $C_{i_j} = C_{i_k}$. For each j from 1 to n, we compute

$$cpl(C_{i_j}) = sm(C_{i_j}, \Delta_f(C_{i_j}) \cup cpl'(Ci_j))$$

where $cpl'(C)$ returns the result of $cpl(C)$ as a set. $sm(C_I, X)$ returns a sequence $(X_m, \ldots, X_1)$ where X_i is a potential superclass of C_I and for any two X_j, X_i, if $j > i$

then $sim(C_I, X_j) \geq sim(C_I, X_i)$. If two classes have the same structural similarity, incomplete classes take precedence. If both are the same kind, the tie is broken arbitrarily.

Once the class precedence list has been computed for all the applicable incomplete classes on a call to a generic function, the applicable methods are computed in the usual fashion. However, if a method is called that is based on the class precedence list of an incomplete class, the ability to automatically define classes, slots and adjust the class precedence list is disabled and the method is called. If an error condition is signalled, the superclass determining the choice of method is eliminated from the class precedence list of the incomplete class affecting the choice of method and the next applicable method is tried.

Example Suppose we have the incomplete classes A, B, C and D with the following slot names.

$$s(A) = \{x_1, x_2, x_3\}$$
$$s(B) = \{x_1, x_4, x_5\}$$
$$s(C) = \{x_2, x_3\}$$
$$s(D) = \{x_1\}$$

When calling the generic function f, an instance of A is passed as the i-th required argument. The set of specializers on the i-th argument is $\{B, C, D\}$. Further, suppose that $cpl'(A) = \emptyset$. Then we have $\Delta_f(A) = \{B, C, D\}$ and we must compute $sim(A, \{B, C, D\})$.

B is eliminated because $s(B) \not\subseteq s(A)$. Thus, $cpl(A) = (C, D)$ since $sim(A, C) > sim(A, D)$. The method specialized to C is called first, but we disable creation of incomplete classes, slot specifiers and class precedence lists. If the method fails, C

is removed from $cpl(A)$ and the next one (in this instance, the one specialized to D) is tried.

An extended, concrete example is presented in Section 5.3.

Discussion The approach to calculating the class precedence list is intended to capture only basic information. This is due to the fact that there is little evidence to discern complex relationships. Unlike the case with slot specifiers, class relationships do not rely on any conventions, providing us only structural information through known slot specifiers and behavioural grouping in methods.

Actual class precedence lists can be quite complex when using multiple inheritance. The relationships themselves can influence how behaviour is specified. For example, suppose we intend for a chair to be considered an item for seating people over that of a stand for a potted plant, although it could be used for both. We define a class for a chair where the superclasses prioritize the property of seating people before plants. Furthermore, we have a generic function for placing an item in a room based on its properties, either people-centric or plant-centric with methods for both. The generic function will then consider a chair with respect to seating people before it considers it as a stand for plants. However, the priority of these two properties is not immediate by considering merely the structural elements making up the class and its behavioural grouping in the method definitions. Instead, it requires grounding it in some realistic situation where providing seating for people is more important than that of plants. This is only exhibited by extensive knowledge of the context in which the class and generic function was defined, or by examining a definition of the chair class.

Thus, it is not certain that by studying behaviour of instances a complete description of the relationship between classes will be made. This is most plainly seen with abstract classes, that is, classes that are not instantiated.

When an abstract class A appears in the applicable specializers of an incomplete class, it will always be considered as part of the class precedence list. When an abstract class is in the set $\Delta_f(C_I)$ for some function f and incomplete class C_I, it will be processed by the function sm. Since $s(A) = \emptyset$, trivially, $s(A) \subseteq s(C_I)$. Thus, A is a potential superclass of C_I.

The problem with abstract classes is that of the earlier example with chairs. Without any specific knowledge of the situation outside of the code itself, discerning the intended order of abstract classes in the class precedence list is largely a matter of chance.

Additionally, ordering elements by structural similarity can be incorrect, even when instances of classes are instantiated. Consider a case where we intend for the class precedence list for a class A to be (B, C, D) where A, B, C, D are incomplete classes and

$$\begin{aligned}
s(A) &= \{x_1, x_2, x_3, x_4\} \\
s(B) &= \{x_1\} \\
s(C) &= \{x_2, x_3\} \\
s(D) &= \{x_4\}
\end{aligned}$$

The function sm will produce either (C, B, D) or (C, D, B). Again, there is no clear indication that B should come before C, let alone D. These factors indicate the method of computing the class precedence list will generally perform poorly with complex class hierarchies.

Consequently, we concentrate on capturing simpler hierarchies. Single inheritance is much simpler than multiple inheritance since there is at most one immediate superclass for any given class. The class precedence list for single inheritance forms a chain as described in Section 5.1. We examine whether the procedure for computing $cpl(A)$ for some class A is effective at capturing single inheritance.

Suppose that we intend for a class C_I to have the single-inheritance precedence list $(X_n, X_{n-1}, \ldots, X_1)$. For this to manifest in the class precedence list — but not necessarily be equal to the class precedence list — it must be the case that for any two classes X_j, X_i with $j > i$, X_j precedes X_i in $cpl(C_I)$. This occurs when $sim(C_I, X_j) > sim(C_I, X_i)$. If we assume that all intended slot specifiers in C_I, X_i and X_j are present when the comparison is made and that $s(X_i) \subset s(X_j) \subset s(C_I)$, then the desired result is achieved.

An important assumption in the argument above is that the slot specifiers are present. If we compute $cpl(C_I)$ before slot specifiers are present in X_i and X_j, the order of X_i and X_j may or may not be correct as intended. For example, if they are both incomplete classes, their order is arbitrary with respect to each other. In general, this is the case when $s(X_i) = s(X_j)$.

It is not required, however, that the full set of intended slot specifiers be present. If it is the case that $s(X_i) \subset s(X_j) \subset s(C_I)$, then X_j will precede X_i in $cpl(C_I)$. This suggests that if X_i or X_j are incomplete classes, instances of them should be created and accessed before computing the class precedence list of C_I.

False positives in the superclass relationship are a problem when basing the relation on structural similarity. If $s(A) = \{x_1, x_2\}$ and $s(B) = \{x_1\}$ for some incomplete classes A and B, then B is potential superclass of A. However, it may be the case that when B has all of its slots specified, $s(B) = \{x_1, x_3\}$. This again demonstrates the volatility of the class precedence list in the face of incomplete information.

It is clear there is a dependency on time for the calculation of the class precedence list for an incomplete class when it depends on incomplete classes. This should not be surprising given that incomplete classes start out with no slot specifiers and that slot specifiers are the most concrete evidence we have for discerning superclass relationships. This could be mitigated by performing a static analysis of the code to determine some of the slots present in a class. Static analysis would not eliminate any of the problems outlined above, but it does provide the opportunity for the analysis to have a potentially better start point, thus making fewer mistakes during the initial steps.

Part of the approach to building class descriptions is relying on the accumulation of knowledge, meaning that during the process, some decisions are based on incomplete information with respect to what is intended. It is for this reason that class precedence lists are recomputed regularly and that decisions based on potentially incomplete information are monitored for errors.

The frequency with which class precedence lists are computed is the primary reason that the approach relies on as few dependencies as reasonably possible. When computing $cpl(C_I)$, we do not use the class precedence lists of any of the elements of $cpl(C_I)$. If some X_i in $cpl(C_I)$ is also incomplete, then at the time we compute $cpl(C_I)$, we should recompute $cpl(X_i)$ as some classes within $cpl(X_i)$ may have changed. In effect, we could end up recomputing the relationships between all incomplete classes on each generic function call. Despite the fact performance is not the goal with this approach to class hierarchy derivation, such a performance penalty might be onerous.

Lastly, we point out that when the class precedence list is not required for any methods when incomplete classes are involved, there are no problems with execution, save for the established handlers being overridden. In order for the class precedence list to be ignored, each generic function call in which an incomplete class factors into the choice of applicable methods must have a method specialized precisely to the type of the incomplete class in the appropriate required argument. Furthermore, `call-next-method` is never invoked in the body of these methods. Assuming the conventions are followed, the class is merely an aggregate. In this case, slots will be created as required and no relationships will be indicated between classes. Thus, for every pair of incomplete classes A, B, we have $s(A) \not\subseteq s(B)$.

The approach we have described may capture single inheritance and some cases of multiple inheritance, given certain conditions. Specifically, it requires a lack of abstract classes and enough slot information to discern provably incorrect relationships between classes. Also, if the applicable specializers for some incomplete class C_I on a generic function only contain classes intended to be related to C_I, then false positives will not be present. Given the lack of evidence for detecting relationships and that some hierarchies can be determined or closely approximated with these conditions, we conclude that the approach is worth pursuing.

Summary

The implementation described above for the evaluation strategy given in Section 5.1 augments the runtime of a Lisp system to recover from three situations that normally cause an error: no class defined, no slot defined and no method found. By handling these cases, class definitions can be elided from program text and constructed as required at runtime.

In particular, the implementation augments the following functions. Each are briefly summarized below.

- `find-class` is wrapped with a condition handler to create incomplete classes when no class is found.
- `slot-missing` and `make-instance` have methods specialized to incomplete classes to create slot definitions that do not exist within the incomplete class definition.
- `function` and `fdefinition` are wrapped with a condition handler to define generic functions for slot accessor functions when a function is undefined and

the call matches a pattern for a slot accessor function.

- `no-applicable-method` and `no-next-method` have methods defined that compute the applicable specializers when instances of incomplete classes are passed in for required parameters, then compute the class precedence lists of the incomplete classes. Methods are also defined for the generic functions `compute--applicable-methods` and `compute-applicable-methods-using-classes`, specifically for the purpose of computing applicable methods.

5.3 Effectiveness

We now show the effectiveness of the technique presented in the previous section on an extended example. The technique is shown to derive a working class hierarchy that behaves as intended in the program and we discuss in more detail why this happens, demonstrating the conditions under which the technique works. We also discuss ways to address the shortcomings described previously.

A small program for generating rudimentary reports about a course is given in Figure 5.1. The program contains no class definitions for the six intended classes: `course`, `person`, `student`, `lecturer`, `ta` (teaching assistant) and `course-head`.

When evaluated with the runtime extensions described, incomplete class metaobjects are created for the undefined classes making up the specializers on the generic functions `report-on` and `associate-person`. This constitutes the entire set of intended classes, although no slot specifiers are present.

The first task of the program is to create a `course` object and associate people with it. Instances of the incomplete classes are created from `*person-data*`; note that each element of the data constitutes parameters to `make-instance`.

Calls to `associate-person` always match their parameters specializers precisely and do not invoke any other methods, so the need for class precedence lists is

```
(defparameter *person-data*
  '((student :name "Alice"   :sid 100)
    (student :name "Bob"     :sid 101)
    (student :name "Charlie" :sid 102)
    (student :name "Diane"   :sid 103)
    (ta :name "John"  :sid 200 :office "HR1")
    (ta :name "Sally" :sid 201 :office "HR2")
    (lecturer :name "Al"   :office "R1")
    (lecturer :name "Mike" :office "R2")
    (course-head :name "Mary" :office "R10" :phone "x759")))

(defgeneric report-on (obj)
  (:method ((obj person))
    (format t "Name: ~A~%" (name obj)))
  (:method ((obj student))
    (format t "Student id: ~A, " (sid obj))
    (call-next-method))
  (:method ((obj lecturer))
    (format t "Office: ~A, " (office obj))
    (call-next-method))
  (:method ((obj course-head))
    (format t "Phone: ~A, " (phone obj))
    (call-next-method))
  (:method ((obj course))
    (mapc #'report-on (students obj))
    (mapc #'report-on (tas obj))
    (mapc #'report-on (lecturers obj))
    (report-on (course-head obj))))
```

Figure 5.1: A program lacking class definitions

Continued from previous page

```
(defgeneric associate-person (p c)
  (:method ((p student) (c course))
    (push p (students c)))
  (:method ((p ta) (c course))
    (push p (tas c)))
  (:method ((p lecturer) (c course))
    (push p (lecturers c)))
  (:method ((p course-head) (c course))
    (setf (course-head c) p)))

(let ((course (make-instance 'course)))
  (loop for data in *person-data*
        for person = (apply #'make-instance data)
        do (associate-person person course))
  (report-on course))
```

Class	**Slot specifier names**
`course`	`students, tas, lecturers, course-head`
`person`	—
`student`	`name, sid`
`lecturer`	`name, office`
`ta`	`name, sid, office`
`course-head`	`name, office, phone`

Table 5.1: Slot specifiers of incomplete classes before `report-on` is called

alleviated, although they are still computed.

Each method on `associate-person`, when called the first time, attempts to call an undefined function. For example, when the method specialized to `person` and `course` is called, an attempt is made to call the function `(setf students)`. (This is done by the macro `push`.) Since that function does not exist and its call matches that of a writer, accessor methods are defined for `(setf students)` and `students`. The accessors use `slot-value` to access the slot, which creates the slot `students` in `course`. The case is similar for the other methods on `associate-person`.

Before `report-on` is called, it is instructive to examine the state of each incomplete function with respect to its slot specifiers. The current state is given in Table 5.1. The intended relationships between the classes are reasonably apparent, even ignoring the semantics behind the names of the classes. Also note that `person` is an abstract class.

At the invocation of `report-on` with the instance of `course` passed to it, no class precedence list factors into the choice of method. However, within the body of that method, `report-on` is called on the objects contained within the instance. The first such call is performed on an instance of a `student`. There is one method specialized to the `student` class which invokes `call-next-method`. It is here that the class precedence list is used. We have

$$\Delta_{\texttt{report-on}}(\texttt{student}) = \{\texttt{course-head}, \texttt{lecturer}, \texttt{ta}, \texttt{student}, \texttt{course}, \texttt{person}\}$$

and

$$cpl'(\texttt{student}) = \{\texttt{ta}, \texttt{lecturer}, \texttt{course-head}\}$$

Note that the value of $cpl'(\texttt{student})$ is the set of the specializers on the first argument of `associate-person`, excluding `student`. This is because the last time an instance of `student` was passed to a generic function (`associate-person`), none of the classes appearing in the specializer of the first argument had any slot specifiers and thus, were all trivially related to `student`.

Looking at potential superclasses of these two sets, only `person` meets the requirements since all other classes have slots with names that are not in `student`. Thus $cpl(\texttt{student}) = \{\texttt{person}\}$. This is true for each instance of `student` that `report-on` is applied to in the `mapc` call.

Next, we look at the call to `report-on` on instances of `ta`. In this case we have

$$\Delta_{\texttt{report-on}}(\texttt{ta}) = \Delta_{\texttt{report-on}}(\texttt{student})$$

and

$$cpl'(\texttt{ta}) = \{\texttt{student}, \texttt{lecturer}, \texttt{course-head}\}$$

The potential superclasses consist of `student`, `lecturer` and `person`. Since $sim(\texttt{ta}, \texttt{student}) = sim(\texttt{ta}, \texttt{lecturer})$, the order of `student` and `lecturer` is arbitrary. We will use lexicographic ordering of the class names. Thus,

$$cpl(ta) = (\texttt{lecturer}, \texttt{student}, \texttt{person}).$$

The program will evaluate without error and produce the class precedence lists given in Figure 5.2. The corresponding class hierarchy is also provided.

What is interesting with this example is that it captures multiple inheritance. This is because all incomplete classes had sufficient slot specifiers to differentiate them when computing their class precedence lists. Note, however, that the class `person` has no slot specifiers when it is intended to have one (the slot `name`). The hierarchy does not capture this information because `person` was an abstract class. We see that by fully instantiating classes before they are passed to methods greatly reduces the chances of false positives, as reasoned earlier. However, the scenario is somewhat contrived. Effective use of the class derivation technique demands that we create objects with a distinctive set of slot specifiers before using them.

The fact that `person` was an abstract class is the reason that it is considered to be a superclass of `course`, although this was not intended. By creating an instance of `person`, giving it a `name` slot and running the program again, this relationship would be eliminated.

To see what problems can arise with abstract classes, suppose no instances of `lecturer` were created before `report-on` was called with an instance of `student`. Then $\Delta_{\texttt{report-on}}(\texttt{student})$ and $cpl'(\texttt{student})$ are the same as previously described, but now

$$cpl(\texttt{student}) = (\texttt{lecturer}, \texttt{person}).$$

This leads to the incorrect output, although it does not interrupt the operation of the program.

One way to deal with incorrect slot specifiers in abstract classes is to use a version of the algorithm given by Lieberherr for factoring out common attributes in classes to form class hierarchies [44]. This would find that `name` is common to all the classes and should belong in the superclass, assuming that there is no intention to override the behaviour of the slot in a subclass.

Class precedence lists

$$\begin{aligned}
cpl(\texttt{course}) &= (\texttt{person})\\
cpl(\texttt{person}) &= ()\\
cpl(\texttt{student}) &= (\texttt{person})\\
cpl(\texttt{lecturer}) &= (\texttt{person})\\
cpl(\texttt{ta}) &= (\texttt{lecturer}, \texttt{student}, \texttt{person})\\
cpl(\texttt{course-head}) &= (\texttt{lecturer}, \texttt{person})
\end{aligned}$$

Class hierarchy

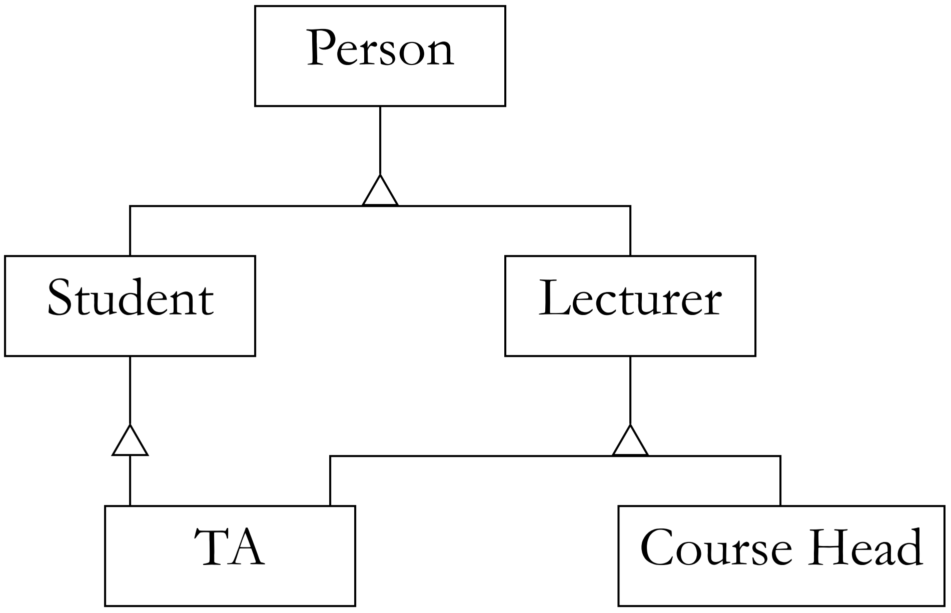


Figure 5.2: Class precedence lists and hierarchy of report program

As we have shown, the technique does effectively capture class hierarchies, although the situation described is somewhat idealistic. By following a certain approach, that is, instantiating objects with a distinctive set of slots and defining methods such that unintended class relationships are not given, it is likely the intended hierarchy will be captured. Whether this is simpler than actually writing out the hierarchy is an open question, although we do note that it is less work in that the hierarchy need not be written explicitly. This does permit flexibility in the definition of classes, however, since a change in definition is realized by simply changing how the class is used.

5.4 Summary

We have detailed an approach to deriving class hierarchies from the use of objects in a running system based on evidence that indicates slots present in classes and relationships between classes. This approach enables objects to be created without a definition of the class provided beforehand. Thus, we can eliminate class definitions from the code and still use instances of them in a program.

The technique was analyzed to show its effectiveness in programs and its shortcomings were discussed. In particular, it was shown that complex hierarchies with multiple inheritance can be difficult to capture as intended due to lack of reliable evidence. We argued that single inheritance can be captured more reliably, but it requires certain conditions be met in order to do so. An extended example was presented that demonstrated a working multiple inheritance hierarchy can be captured under these conditions, despite some errors in the derived class definitions. We conclude that the technique is effective when a certain style is used and that it is unclear whether it is better to write the hierarchy out explicitly.

Chapter 6

Conclusions

This thesis has argued that deriving structure from behaviour, what we have termed *behavioural synthesis,* is useful for writing specifications in an implementation language because it allows details pertaining to structure to be omitted in the code describing the specification while allowing the specification to be executable. Furthermore, we have argued that the abstractions represented by techniques employing behavioural synthesis can be used to produce specialized versions of the abstractions, eliminating the abstraction from the specification to produce an implementation.

The first approach to behavioural synthesis we demonstrated was dynamic abstract data types (DADTs), a framework for defining data types that provides a mechanism for dynamic reconfiguration of their structure based on the context in which certain operations occur. In particular, they are helpful for performing dynamic analysis and facilitating the definition of types that are the union of other types. Such types allow for succinctness in the definition of behaviour.

Furthermore, we showed how program code can be generated to specialize types represented by DADTs. This facilitates the evolution of a specification to an implementation. In particular, we demonstrated the specialization of a DADT that is the union of other types with overlapping interfaces, thus introducing ambiguity. The ambiguity is resolved interactively, but the context at the time of resolution is remembered and code is generated based on those contexts.

In addition to specializing type definitions, we demonstrated a technique for building class definitions from the use of objects, enabling class definitions for those objects to be omitted from the program. The technique is shown to be effective, in the cases considered, at capturing class hierarchies from lexical and behavioural evidence found in the program and its execution. Limitations to the technique and conditions under which it is expected to work were also described. Although the programming style used in code lacking class definitions affects the results, it does permit the implicit structure in a specification to be altered by only making changes to behaviour.

Behavioural synthesis presents interesting avenues for code generation through context capture and evidence of structure presented by language constructs. The problems encountered in specializing the types of DADTs show that context must be defined carefully in order to produce code capturing the intent of an operation. Also, the shortcomings of class hierarchy derivation show that further research on language constructs helpful for exhibiting structure from behaviour is required.

Lexicon

This lexicon provides definitions for common terms used in many object-oriented languages that are likely familiar to the reader, but whose definition in most object-oriented languages differs from that used in this work.

class an object made up of slots that describes the structure of other objects. In most object-oriented languages, classes are composed of slots and methods, and are not themselves manipulable objects. See Section 5.1 for further discussion.

generic function a function associated with methods that chooses a set of methods to call when invoked, thus determining its behaviour, based on the arguments passed to the function.

method an object that makes up the behaviour of a generic function when the arguments to that function are of certain classes or values. In most object-oriented languages, methods are associated with a single class. In this work, methods are associated with multiple classes.

slot a component making up an instance that holds a value; more commonly known as an *attribute* or *field*.

Bibliography

[1] *Symbolic Common Lisp: Language Concepts*, volume 7, chapter 5: Table Management. Symbolics Inc., 1986.

[2] Umut A. Acar, Guy E. Blelloch, Matthias Blume, and Kanat Tangwongsan. An experimental analysis of self-adjusting computation. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 96–107, New York, NY, USA, 2006. ACM Press.

[3] Ole Agesen and Urs Hölzle. Type feedback vs. concrete type inference: a comparison of optimization techniques for object-oriented languages. *SIGPLAN Not.*, 30(10):91–107, 1995.

[4] Rakesh Agrawal, Surajit Chaudhuri, Abhinandan Das, and Vivek R. Narasayya. Automating layout of relational databases. In *International Conference on Data Engineering*, pages 607–618, 2003.

[5] Gregory R. Andrews and David P. Dobkin. Active data structures. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 354–362, Piscataway, NJ, USA, 1981. IEEE Press.

[6] Warren Armstrong, Peter Christen, Eric McCreath, and Alistair P Rendell. Dynamic algorithm selection using reinforcement learning. *International Workshop on Integrating AI and Data Mining (AIDM)*, 0:18–25, 2006.

[7] John Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, 2003.

[8] Fernando Berzal, Juan-Carlos Cubero, Nicolás Marin, and Maria-Amparo Vila. Lazy types: automating dynamic strategy selection. *Software, IEEE*, 22(5):98–106, Sept-Oct 2005.

[9] Aart J. C. Bik and Harry A. G. Wijshoff. On automatic data structure selection and code generation for sparse computations. In *1993 Workshop on Languages and Compilers for Parallel Computing*, number 768, pages 57–75, Portland, Ore., 1993. Berlin: Springer Verlag.

[10] Lee Blaine and Allen Goldberg. DTRE – a semi-automatic transformation system. In B. Möller, editor, *Proceedings of the IFIP TC2 Working Conference on Constructing Programs from Specifications*, pages 165–204, Amsterdam, 1991. North-Holland.

[11] Taylor L. Booth and Cheryl A. Wiecek. Performance abstract data types as a tool in software performance analysis and design. *Software Engineering, IEEE Transactions on*, SE-6(2):138–151, March 1980.

[12] Nicholas John Carriero Jr. *Implementation of tuple space machines*. PhD thesis, Yale University, 1987.

[13] Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[14] Alistair Cockburn. *Agile Software Development: The cooperative game*. Agile Software Development Series. Addison-Wesley, 2nd edition, 2007.

[15] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming: an overview of contextl. In *DLS '05: Proceedings of*

the 2005 conference on Dynamic languages symposium, pages 1–10, New York, NY, USA, 2005. ACM Press.

[16] Krzysztof Czarnecki, Ulrich W. Eisenecker, Robert Glück, David Vandevoorde, and Todd L. Veldhuizen. Generative programming and active libraries. In *Selected Papers from the International Seminar on Generic Programming*, pages 25–39, London, UK, 2000. Springer-Verlag.

[17] Robert K. Dewar, Arthur, Ssu-Cheng Liu, Jacob T. Schwartz, and Edmond Schonberg. Programming by refinement, as exemplified by the setl representation sublanguage. *ACM Trans. Program. Lang. Syst.*, 1(1):27–49, 1979.

[18] Andrew D. Eisenberg and Gregor Kiczales. Expressive programs through presentation extension. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 73–84, New York, NY, USA, 2007. ACM.

[19] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *Software Engineering, IEEE Transactions on*, 27(2):99–123, 2001.

[20] Michael D. Ernst, William G. Griswold, Yoshio Kataoka, and David Notkin. Dynamically discovering program invariants involving collections. Technical Report TR UW-CSE-99-11-02, University of Washington, March 2000.

[21] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. Technical report, Research Institute for Advanced Computer Science, 2000.

[22] Norbert E. Fuchs. Specifications are (preferably) executable. *Software Engineering Journal*, 7(5):323–334, Sep 1992.

[23] Richard P. Gabriel and Ron Goldman. Conscientious software. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 433–450, New York, NY, USA, 2006. ACM Press.

[24] Daniel D. Gajski and Loganath Ramachandran. Introduction to high-level synthesis. *Design and Test of Computers, IEEE,* 11(4):44–54, Winter 1994.

[25] David Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[26] Shahram Ghandeharizadeh, Doug Ierardi, and Roger Zimmermann. An online algorithm to optimize file layout in a dynamic environment. *Inf. Process. Lett.*, 57(2):75–81, 1996.

[27] Torbjorn Granlund. *The GNU MP manual*, 2006.

[28] Philip J. Guo, Jeff H. Perkins, Stephen McCamant, and Michael D. Ernst. Dynamic inference of abstract types. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 255–265, New York, NY, USA, 2006. ACM.

[29] Brent Hailpern and Gail E. Kaiser. Dynamic reconfiguration in an object-based programming language with distributed shared data. *Distributed Computing Systems, 1991., 11th International Conference on*, pages 73–80, May 1991.

[30] Cordelia V. Hall. Using hindley-milner type inference to optimise list representation. *SIGPLAN Lisp Pointers*, VII(3):162–172, 1994.

[31] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, March-April 2008.

[32] William A. Hoff, Ryszard S. Michalski, and Robert E. Stepp. Induce 2: A program for learning structural descriptions from examples. ISG 83-4 UIUCDCS-F-83-904, University of Illinois, Urbana, January 1983.

[33] Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 326–336, New York, NY, USA, 1994. ACM Press.

[34] D. B. Johnston. An expert system for selecting and tailoring abstract data type implementations. In Barry Lynch, editor, *Ada, experiences and prospects: proceedings of the Ada-Europe International Conference*, volume 1, pages 139–148, Dublin, Ireland, 12-14 June 1990. Ada-Europe.

[35] Elaine Kant and David R. Barstow. The refinement paradigm: The interaction of coding and efficiency knowledge in program synthesis. *Software Engineering, IEEE Transactions on*, SE-7(5):458–471, Sept. 1981.

[36] J. L. W. Kessels. On-the-fly optimization of data structures. *Communications of the ACM*, 26(11):895–901, 1983.

[37] Gregor Kiczales. Beyond the black box: open implementation. *Software, IEEE*, 13(1):8, 10–11, Jan 1996.

[38] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[39] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. *ECOOP'97 Object-Oriented Programming*, pages 220–242, 1997.

[40] Thomas Kistler and Michael Franz. Continuous program optimization: A case study. *ACM Trans. Program. Lang. Syst.*, 25(4):500–548, 2003.

[41] Donald Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley Professional, 1997.

[42] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot client compiler for Java 6. *ACM Trans. Archit. Code Optim.*, 5(1):1–32, 2008.

[43] Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. ISBN 0-534-94602-X.

[44] Karl J. Lieberherr, Paul Bergstein, and Ignaciao Silva-Lepe. From objects to classes: algorithms for optimal objection-oriented design. *Softw. Eng. J.*, 6(4):205–228, 1991.

[45] Henry Lieberman, editor. *Your Wish is My Command: Programming by Example*. Morgan Kaufmann, San Francisco, 2001.

[46] Barbara Liskov and Stephen Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN symposium on Very high level languages*, pages 50–59, New York, NY, USA, 1974. ACM Press.

[47] James Low and Paul Rovner. Techniques for the automatic selection of data structures. In *POPL '76: Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages*, pages 58–67, New York, NY, USA, 1976. ACM Press.

[48] James R. Low. Automatic data structure selection: an example and overview. *Communications of the ACM*, 21(5):376–385, 1978.

[49] Tara M. Madhyastha and Daniel A. Reed. Intelligent, adaptive file system policy selection. In *FRONTIERS '96: Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation*, page 172, Washington, DC, USA, 1996. IEEE Computer Society.

[50] Mentor Graphics. Catapult C synthesis. `http://www.mentor.com`.

[51] Prateek Mishra and Uday S. Reddy. Declaration-free type checking. In *POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 7–21, New York, NY, USA, 1985. ACM.

[52] Peter Naur. *Computing: A Human Activity*, chapter Programming as Theory Building, pages 37–48. ACM Press, 1992.

[53] Kenneth W. Ng, Zhenghao Wang, Richard R. Muntz, and Silvia Nittel. Dynamic query re-optimization. In *SSDBM '99: Proceedings of the 11th International Conference on Scientific on Scientific and Statistical Database Management*, page 264, Washington, DC, USA, 1999. IEEE Computer Society.

[54] Robert O'Callahan and Daniel Jackson. Lackwit: A program understanding tool based on type inference. In *Proceedings of the 1997 (19th) International Conference on Software Engineering*, pages 338–348, 1997.

[55] Leon J. Osterweil. Software processes are software too. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, pages 2–13, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[56] Leon J. Osterweil. Software processes are software too, revisited. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 540–548, New York, NY, USA, 1997. ACM.

[57] Ki-hong Park, Jun-ichi Aoe, Masami Shishibori, and Hisatoshi Arita. An automatic selection method of key search algorithms based on expert knowledge bases. *SIGIR Forum*, 28(1):13–26, 1994.

[58] John Peterson. Untagged data in tagged environments: choosing optimal representations at compile time. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 89–99, New York, NY, USA, 1989. ACM Press.

[59] Kent M. Pitman. EQUAL rights—and wrongs—in Lisp. *SIGPLAN Lisp Pointers*, VI(4):36–40, 1993.

[60] Jack W. Reeves. Code as design. *developer.* Magazine*, 2005.

[61] Charles Rich and Howard E. Shrobe. Initial report on a Lisp programmer's apprentice. *IEEE Transactions on Software Engineering*, SE-4(6):456–467, November 1978.

[62] Charles Rich and Richard C. Waters. The programmer's apprentice: A research overview. *Computer*, 21(11):10–25, 1988.

[63] Julian Richardson. Automating changes of data type in functional programs. In *Proceedings of the 10th Knowledge-Based Software Engineering Conference*, pages 166–173, 1995.

[64] Julian D. C. Richardson. *The Use of Proof Plans for Transformation of Functional Programs by Changes of Data Type*. PhD thesis, University of Edinburgh, 1995.

[65] Stanley J. Rosenschein and Shmuel M. Katz. Selection of representations for data structures. In *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*, pages 147–154, New York, NY, USA, 1977. ACM Press.

[66] Patrick Schmid and Christine Hofmeister. Flexible incremental development by integrating specification and code. In *The IASTED International Conference on Software Engineering*, Innsbruck, Austria, February 17–19 2004.

[67] Edmond Schonberg, Jacob T. Schwartz, and Micha Sharir. Automatic data structure selection in SETL. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 197–210, New York, NY, USA, 1979. ACM Press.

[68] Edmond Schonberg, Jacob T. Schwartz, and Micha Sharir. An automatic technique for selection of data representations in SETL programs. *ACM Trans. Program. Lang. Syst.*, 3(2):126–143, 1981.

[69] J. T. Schwartz. Automatic data structure choice in a language of very high level. *Commun. ACM*, 18(12):722–728, 1975.

[70] J.T. Schwartz, R.B.K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: an introduction to SETL*. Texts and Monographs in Computer Science. Springer-Verlag, New York, 1986.

[71] Guarav Singh, Sumit Gupta, Sandeep Shukla, and Rajesh Gupta. *EDA for IC System Design, Verification and Testing*, volume 1, chapter 11. CRC Press, 2006.

[72] I. N. Skopin. Multiple data structuring. In *Programming and Computer Software*, volume 32, pages 44–55. Pleiades Publishing Inc., 2006.

[73] Yannis Smaragdakis. *Domain-Specific Program Generation*, chapter A Personal Outlook on Generator Research. Lecture Notes in Computer Science. Springer-Verlag, 2004.

[74] Douglas R. Smith. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, 1990.

[75] John Stanik. A conversation with jeff bonwick and bill moore. *Queue*, 5(6):13–19, 2007.

[76] Pankaj Surana. *Meta-compilation of language abstractions*. PhD thesis, Northwestern University, Evanston, IL, USA, 2006. Adviser-Ian Horswill.

[77] Warren Teitelman and Larry Masinter. The Interlisp programming environment. *Computer*, 14(4):25–33, 1981.

[78] Enn Tyugu. The structural synthesis of programs. In *Proceedings on Algorithms in Modern Mathematics and Computer Science*, pages 290–303, London, UK, 1981. Springer-Verlag.

[79] Enn Tyugu, Mihhail Matskin, and Jaan Penjam. Applications of structural synthesis of programs. In *FM '99: Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume I*, pages 551–569, London, UK, 1999. Springer-Verlag.

[80] Axel van Lamsweerde. Formal specification: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 147–159, New York, NY, USA, 2000. ACM.

[81] Todd L. Veldhuizen. *Active libraries and universal languages*. PhD thesis, Indiana University, Indianapolis, IN, USA, 2004. Adviser-Andrew L. Lumsdaine.

[82] Philip Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 307–313, New York, NY, USA, 1987. ACM Press.

[83] Richard C. Waters. The programmer's apprentice: A session with KBEmacs. *IEEE Transactions on Software Engineering*, SE-11(11):1296–1320, November 1985.

[84] John R. White. On the multiple implementation of abstract data types within a computation. *IEEE Transactions on Software Engineering*, 9(4):395–411, 1983.

[85] Patrick H. Winston. Learning structural descriptions from examples. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1970.

[86] Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, 1971.

[87] Niklaus Wirth. *Algorithms + data structures = programs*. Prentice-Hall series in automatic computation. Prentice-Hall, Englewood Cliffs, N.J., 1976.

[88] Ouri Wolfson, Sushil Jajodia, and Yixiu Huang. An adaptive data replication algorithm. *ACM Trans. Database Syst.*, 22(2):255–314, 1997.

[89] Robert Woodbury, Andrew Burrow, Sambit Datta, and Teng-Wan Chang. Typed feature structures and design space exploration. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 13:287–302, 1999.

[90] Geoff Wozniak, Mark Daley, and Stephen Watt. Dynamic ADTs: a "don't ask, don't tell" approach to data abstraction. In *International Lisp Conference*, pages 209–220. The Association of Lisp Users, April 2007.

[91] Geoff Wozniak, Mark Daley, and Stephen Watt. Adaptive libraries and interactive code generation for common lisp. In *Proceedings of the 5th European Lisp Workshop (ECOOP 2008)*, pages 30–38, 2008.

[92] D. M. Yellin. Competitive algorithms for the dynamic selection of component implementations. *IBM Systems Journal*, 42(1):129–135, 2003.

Geoffrey Z. Wozniak

Curriculum Vitae

Education

Bachelor of Science, Computer Science 2001

University of Western Ontario

Thesis: *Genomic Codes, Languages and Information*

Advisor: Dr. Lila Kari

Teaching

University of Western Ontario

Organization of Programming Languages, 2008, 2002

Analysis of Algorithms, 2007, 2003

Computer Science Fundamentals, 2005

Compiler Theory, 2001

Awards

National Science and Engineering Research Council (NSERC)

Post-Graduate Scholarship

Awarded 2002, 2004

Szilard Award for Theoretical Computer Science

Awarded 2001 for undergraduate thesis

NSERC Undergraduate Scholarship

Awarded 2000, 2001

Publications

G. Wozniak, M. Daley and S. Watt. Adaptive libraries and interactive code generation for common lisp. In *Proceedings of the 5th European Lisp Workshop (ECOOP 2008)*, pages 30–38, 2008.

C. Power, I. McQuillan, H. Petrie, P. Kennaugh, M. Daley and G. Wozniak. No going back: An interactive visualization application for trailblazing on the web. In *12th International Conference on Information Visualisation*, 2008.

G. Wozniak, M. Daley and S. Watt. *Dynamic ADTs: a "Don't Ask, Don't Tell" approach to data abstraction*. In International Lisp Conference, pages 209–220. The Association of Lisp Users, April 2007.

G. Wozniak. *Subword pattern matching using constraint satisfaction*. In 3rd WSEAS International Conference on Applied Mathematics and Computer Science (AMCOS), 2004.

L. Kari, S. Konstantinidis, S. Perron, G. Wozniak and J. Xu. *Computing the hamming distance of a regular language in quadratic time*. In WSEAS Transactions on Information Science and Applications, pages 445–449, 2004.

L. Kari, S. Konstantinidis, E. Losseva and G. Wozniak. *Sticky-free and overhang-free DNA languages*. Acta Informatica, 40(2):119–157, 2003.