Dynamic ADTs: a "don't ask, don't tell" policy for data abstraction

Geoff Wozniak
Dept. of Computer Science
University of Western Ontario
London, Ontario, Canada
wozniak@csd.uwo.ca

Mark Daley
Depts. of Computer Science, Biology
University of Western Ontario
London, Ontario, Canada
daley@csd.uwo.ca

Stephen Watt
Dept. of Computer Science
University of Western Ontario
London, Ontario, Canada
watt@scl.csd.uwo.ca

ABSTRACT

We outline an approach to abstract data types (ADTs) that allows an object of the type specified by the ADT to take on one of many possible representations. A *dynamic abstract data type* (DADT) is dual to dynamic algorithm selection and facilitates profiling of data in conjunction with the profiling of code. It also permits a programmer to delay or ignore details pertaining to data representation and enhance the efficiency of some algorithms by changing representations at run time without writing code extraneous to the algorithm itself. Additionally, we demonstrate that run time optimization of data objects is possible and allows for acceptable performance compared to traditional ADTs. An implementation is presented in Common Lisp.

1. INTRODUCTION

In writing a program, there are often choices regarding what algorithms and data structures to use. The justification for choosing one over another may be related to the difficulty of the implementation, availability of existing code, or the programmer's familiarity with the concepts involved. A more reasonable strategy, however, is to pick an implementation that is computationally efficient with respect to the input.

If only one algorithm or data representation is chosen, there will likely be occasions where the program will perform poorly. Special cases can be added to the code to deal with these problems, but that makes the program code more complex. One approach to solving this problem is to use a library that implements multiple versions of the desired functionality and can be tailored for certain input characteristics.

In this paper, we outline the design and implementation of a system for multiple implementations of abstract data types (ADTs) that we call *dynamic abstract data types* (DADTs). Dynamic abstract data types are dual to dynamic algorithm selection. We define a dynamic abstract data type as an abstract data type with multiple implementations that allows individual objects to change representation dynamically. Representation changes may be made by explicit request or performed automatically. Explicit changes have been considered by other authors [21, 13]. In this paper we concentrate on automatic variation of object representation. Furthermore, making use of dynamic ADTs with our implementation does not require changes to existing code.

During the development process, a programmer can spend a significant amount of time trying to come up with constructs that will exhibit acceptable performance. This effort is practically for naught if the portion of code being written contributes little to the run-time profile of the program. The practice of profiling has shown that it is just as effective to first write simple code and specialize it later after profiling has shown where the bottlenecks lie.

Dynamic ADTs are a way to do the same for data. Data objects can be monitored in order to provide the programmer with information as to what operations are actually performed on the object, how often they are performed, and the nature of the operands. It is useful to view DADTs through the lens of the code/data duality that is well understood in the Lisp world. Just as profiling and just-in-time compilation provide a metrics-based approach to dynamic improvement of program representation, DADTs provide a metrics-based approach to dynamic improvement of data representation. In both cases it is possible to vary the degree of monitoring to balance the benefits of the changes with the cost of the overhead.

Furthermore, by incorporating dynamic ADTs into a programming workflow, it delays the need for the programmer to consider representation at the time of object creation; this can prevent accidental complexity arising from the *ad hoc* creation of data representations. For example, in OCaml and Haskell, the standard libraries contain some data types (sets) that specify an internal representation and mandate certain properties of the elements that are associated with it. A dynamic ADT ignores these issues until it becomes necessary to address them, at which point it is possible for the dynamic ADT to remember the circumstances under which the problem arose and handle it cleanly in the future.

Dynamic ADTs also facilitate a reactive form of data structuring and operation support using predefined conditions. For example, the act of filling a set with integers is enough to know that the minimum element can be found without stating that the set will contain only integers. This can be achieved by monitoring the activity of a set object and responding to an operation requiring an ordering by changing the representation to accommodate the operation, assuming the preconditions for the operation are met.

It is common to have situations where changing an object's representations is beneficial. The computational complexity of many algorithms is greatly influenced by the representation of the data structure being operated on. With sets, the fundamental operation of membership can be constant when hash tables are used, but is $O(n)$ or $O(\log n)$ for lists and balanced trees, respectively. The all-pairs shortest path problem for graphs can be solved in $O(|V|^3)$ using a matrix representation instead of $O(|V|^4)$ for the adjacency list, assuming the graph is dense [5, chapter 25]. In fact, any graph algorithm that performs extensive lookup of edges is better served by a matrix representation as finding the existence of an edge is effectively constant. Other times, computational complexity may be less important than raw performance. Choosing a representation that fits in cache is preferred to one that does not, even if the complexity of the algorithm on the available representations is the same.

Reconfiguration of the underlying structure of a data type is similar to the notion of a dynamic ADT. Hash tables are commonly implemented so that they are resized when the number of keys reaches a certain threshold. This is an effective technique for preventing a high number of accesses to find a key in the average case [11, section 6.4]. Resizing involves allocating a new table and re-inserting all the keys, which is fundamentally the same as changing the underlying representation.

With DADTs, the ADT is not bound to a single representation at run time and the programmer no longer implicitly requests a specific structure when creating an object. Furthermore, the ADT is not required to hint at its internals. In this sense, we informally refer to DADTs as a "don't ask, don't tell" policy for data abstraction.

In summary, DADTs demonstrate the following advantages over traditional ADTs:

- by facilitating the profiling of data objects, a metrics-based approach to data representation can be achieved without adding special cases to existing code;
- decisions regarding representation can be delayed during program development;
- allows for optimizations at run time through the monitoring of data objects.

The remainder of the paper is organized as follows: Section 2 outlines definitions, principles and the protocol used by interface operations for DADTs; Section 3 gives an example of a DADT for sets; Section 4 demonstrates various optimization techniques; Section 5 provides a discussion of related and future work. Source code for the implementation is publicly available (see [23]).

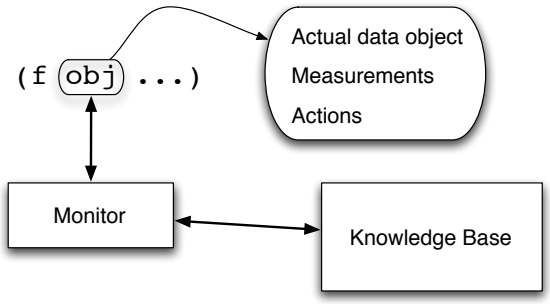

Figure 1: Basic outline of the approach. When some operations are executed, certain data objects passed to them are monitored and may be changed based on information in a knowledge base.

2. OUTLINE OF A DADT PROTOCOL

The basic outline of dynamic ADTs is shown in Figure 1. Instances of dynamic ADTs have information associated with them that is accessed by a monitor when certain operations are executed. The monitor has the ability to change the representation of the actual data object contained within the instance. In this section, we describe a protocol used by interface operations of dynamic ADTs to perform the extra work alluded to in Figure 1.

We define an abstract data type as a set of functions that define an interface, called the *interface operations*, and one or more implementations that can be captured in an object that structures data in some manner such that the interface operations adhere to their semantics when operating on the object.

Formally, a *dynamic abstract data type* is an abstract data type whose instances contain a data object that works with a specific implementation of an abstract data type as well as extra information about that data object. An instance of a DADT (also called a *DADT object*) works in the same manner as the contained data object, that is, the operations defined for the contained data object are also defined for the DADT object. It is possible (and likely) that the data object contained in a DADT object is one of multiple representations. This means the interface operations defined for the DADT cannot rely on a specific data layout and must make use of the interface operations for the data objects themselves. The data object contained with a DADT object is called the *current representation* of the DADT object. An interface operation that operates on the current representation is called an *actual (interface) operation*. A *dynamic abstract data type protocol* is a procedure for collecting information about a DADT object's current representation and altering it with respect to certain conditions.

First, we note that the class of a DADT object is not a subclass of the current representation because, strictly speaking, it is not inheriting behaviour but rather observing the original object. As a consequence, the interface operations

must also be wrapped (but not necessarily re-implemented).

DADT objects have slots to hold information from observations and associations with actions. Information can be collected and actions can be taken when the interface operations are executed. Note that an interface operation is an operation that is specifically meant to work on DADT objects. Stipulating that DADT objects can collect information in other contexts would mean changing existing code bases, something we want to avoid. We think this is a reasonable restriction. There may be merit to letting objects observe their environment in other situations, but it is not clear that this is worth the effort.

An interface operation has three primary tasks: execute the expected functionality associated with the operation (which likely involves invoking the actual interface operation on the appropriate data), allow the DADT object(s) passed to the operation to collect information, and allow the same DADT object(s) to perform actions. Strictly speaking, the only task that is required for a working program is performing the expected operation. The others constitute the *extra work*. Thus, we will examine them more closely.

Collecting information. Within the context of interface operations, there are a multitude of things that can be observed. We may want to examine the arguments to the interface operation, the data contained in a DADT object, or observe the time taken to perform the actual operation. For simplicity, we designate that there be two points where an observation or measurement take place. These two points will be known as the initial and final measurement points (or stages), respectively. The initial measurement must be taken prior to the execution of the actual operation and the final measurement must be made afterwards.

To indicate what should be measured, each DADT object is associated with a set of other objects called *resources*. Resources are data stores for observations. When a measurement point is reached, a function is applied to each resource and passed data for making an observation. Thus, different types of measurements are indicated by the different resources associated with a DADT object.

One important point remains with respect to taking measurements: communication. In order to make an accurate measurement, it may be necessary to store an intermediate value with the initial measurement and use it to compute a result with the final measurement; the canonical example here is execution time. We will call such values *intermediate resource values* and dictate that if a value is returned when an initial measurement is taken with respect to some resource, that value is made available when the final measurement (if any) is taken.

Detecting conditions. Also associated with DADT objects are actions. An action will likely only be taken if some condition is met. Similar to measurements, the possibilities for actions are numerous. What is problematic is that after one action is taken, it is likely to affect the outcome of other actions. This may or may not be desirable. To deal with this situation, we divide the types of actions into two groups: those that have an effect on the current representation and those that do not. Since the current representation is the focus of the actions in the first place, it seems reasonable to consider it the point of differentiation.

We call actions, possibly with a condition attached to them, *triggers*. Triggers that have a direct effect on the current representation are called *active*, whereas ones that do not are *passive*. Each DADT object is associated with a (possibly empty) set of triggers that are all run in succession at the same point in the execution of an interface operation. The reason for grouping them together in such a fashion is a pragmatic one. Specifying a precise time for running a trigger would become too complicated outside of simply writing every interface operation manually.

The specification for triggers, however, is more involved than would first appear. There are still some important points to consider:

- In what order are the triggers to be run?
- How can we prevent one trigger from affecting the conditions of another?
- How are triggers handled when multiple DADT objects are considered in tandem?

The first concern can be addressed by providing a mechanism by which triggers can be sorted, although for most purposes, the order in which they are provided is sufficient. For the second, we will take the approach that if a trigger is active, any subsequent triggers will not be run. (This is analogous to the `cond` form.)

The final concern (multiple DADT objects) is the most troublesome. Triggers are associated with individual objects and thus, are ill-suited for dealing with cases where objects must cooperate in some manner. For example, if O_1 and O_2 are DADT objects passed to an interface operation, it may be the case that a trigger for O_1 could alter its current representation such that it is no longer compatible with the current representation of O_2 when passed to the actual interface operation.

To solve this problem, we introduce a different classification of triggers that ensure compatibilities between multiple DADT objects as well as any other conditions. These are called *class triggers* because they are associated with the class of DADT objects and not the DADT objects themselves. A non-class trigger is called a *local trigger*. Class triggers behave as local triggers except that the conditions in which they are run are mutually exclusive: if class triggers are executed then local triggers are not and vice versa.

Finally, we stipulate that a trigger is given access to three things: the DADT object(s) the trigger is associated with, the interface operation from which the trigger is being called and the arguments passed to that operation. This provides the context in which the trigger is being executed.

Common experiences and feedback. At this point, we have enough to enable a DADT object to make observa-

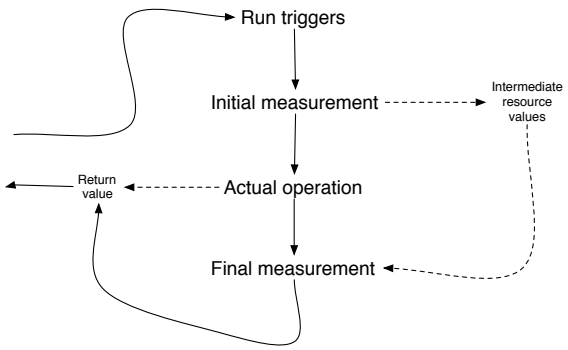

Figure 2: The usual order of operations taken for an interface operation. The dashed lines indicate values set by or passed to different stages. The solid lines represent the flow of control.

tions about the context in which it is used (to a limited extent) and act on that information. However, any such data can only be accessed by a single DADT object, that is, the DADT object that makes the observation. It would be useful to allow different DADT objects to share "experiences" with each other in order to determine what constitutes a common situation in a context outside of a single object.

We will provide three primary contexts for sharing information: local, class and global. The local context is that of an individual DADT object. Class context refers to a class of DADT objects and the global context is the environment of the computation. Each portion of information within a context will be called a *directive* because they are used to guide the behaviour of DADT objects.

The set of directives available to a DADT object can be thought of as a knowledge base from which the object can determine behaviour based on observations. By adding to the knowledge base, it is possible for DADT objects to choose reasonable defaults in the absence of certain information.

Furthermore, directives provide a convenient way to communicate information in the event of an error. If a situation arises in which a DADT object does not know how to proceed, it is conceivable that an error could be signaled which prompts the user for guidance. The answer can then be stored for future reference and if the situation is detected again, the previous experience can be drawn upon to prevent further interruption.

Summary Figure 2 provides a summary of the usual execution of an interface operation.

3. A DYNAMIC SET

To demonstrate use of the protocol, we will outline the implementation of a set data type in which individual objects observe how they are being used and adjust aspects of themselves to meet certain conditions.

We start by noting that the contents of a set have a profound impact on the behaviour of a set in a programmatic setting. Most issues that arise with the specification or representation of a set are a result of what we intend (or expect) the contents to be. One obvious problem is testing for membership by way of equality. In order for the set to behave as intended, it is important that the notion of equality meet with expectations. The situation is further complicated by the desire to provide the convenience of ordering statistics when the situation presents itself. When the elements of a set have a "natural" ordering, we would like to take advantage of it.

In both of these cases, it is the type of the elements contained in the set that have the most significant effect on its structure and the operations it supports. Thus, we will build a set that monitors the types of the elements put into it, and watch for the conditions that affect how to test for equality within a set and whether the elements can be ordered based on some commonly accepted methodologies.

First, a word on types. In order to accommodate common situations, it may be convenient to think of `5` and `5.0` as the same thing: a number. The two primitives in Common Lisp for getting the type of an object (`type-of` and `class-of`) will return a more specialized type than `number` when applied to these objects. For example, `(class-of 5)` will likely return a the `fixnum` class metaobject whereas `(class-of 5.0)` will return `single-float`. To facilitate a convenience mapping, we define a function `class-of*` that returns a common supertype for some objects (namely numbers and strings) so that the minutiae of type specialization can be avoided when desired. This mapping can be changed to meet the user's needs.

Lastly, we note that taking advantage of the common situations (such as using `class-of*`) is not mandatory. If specific parameters are provided for equality testing or ordering, such parameters take precedence over any defaults decided upon by the system.

3.1 Interface

The interface for sets will consist of the usual set functions as well as some ordering statistics operations, such as `nth-smallest` and `nth-largest`.

In order to reliably keep track of the types of elements within a set, we stipulate that the only operations that add or remove elements from a set are `insert-element`, `delete-element` and `clear`; no other operations may manipulate the underlying representation to change the contents.

3.2 Resources

We only need to keep track of the types of elements that are present in a set; however, we must know how many of each type are present to reliably handle some situations. Without the number of each type present, we would not know when some type is no longer present without investigating every element in the set. This is not desirable.

To define a resource for the protocol, we define a class that collects the information we need. Then, the protocol operations that facilitate such information collection are specialized to that class. For purposes of collecting type information about the elements in the set, we focus on the

```
(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 (class-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 (class-of* elem)))))

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

(defmethod reset-resource ((tc type-counter))
  tc)
```

Figure 3: Specialization of the protocol functions for taking measurements by counting the types of each element present in a set.

operations `insert-element`, `delete-element` and `clear`.

The class for collecting the information is called `type-counter` and the operation specializations are given in Figure 3.

The function `measure-resource-initial` is called on each resource when the initial measurement stage is executed. It takes four arguments: the resource to measure, the current representation of the DADT object the resource is associated with, the operation the resource is being measured from (as a symbol) and the arguments passed to that operation as a single object. The operations that require measurements are specified in the lambda list. The call to `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 type counter accordingly.[1]

The function `reset-resource` is called when the current representation of a DADT object is changed. The function should return an instance of the resource class. In this case, the resource is not affected by a change in the current representation so it simply returns the resource object passed into it.

The function `measure-resource-final` is called on each resource when the final measurement stage is executed. Since no measurements need to be taken in the final stage of resource measurement, we do not define any methods for it.

[1]Strictly speaking, we should verify that the element is present or missing by mandating that the insertion and deletion functions return whether the element was deleted or not, but this requires considerably more work (although doing so is possible). For simplicity, we won't second-guess the actual interface operation.

Each resource function mentioned above has a method where each argument is specialized to the class `t`; this method does nothing, so there is no need to define methods for the default case.

3.3 Triggers

In the definition of a DADT, defining triggers is the most complicated part of the process. This is because triggers are meant to detect conditions that arise from the use of the DADT objects and act on them. Some conditions can be particularly complicated to act on, especially if the fuzzy goal of familiarity is desired.

The conditions we need to detect with a dynamic set amount to ensuring that the equality function within the set properly handles the different types of elements present, and changing the current representation to facilitate ordering statistics when possible. Intuitively, these are straightforward conditions to detect and act on. In practice, there are many subtleties to consider.

Instead of discussing all the problems involved, we will outline the basic framework used to solve them. The primary tool used to infer what course of action to take is a simple knowledge base of previous situations and specified preferences made available to the triggers; these are made available through directives. For example, the knowledge base holds the names of functions used to test for equality between elements of the same type and how confident we are in those functions. In the knowledge base, the class `number` may be associated with the function `equalp` with high confidence because it meets the normal expectation of equality for numbers (that is, `(equalp 5 5.0)` is true). The other standard equality functions may also be associated with `number`, but with lower confidence: while they will not fail, they don't work in the expected fashion. The knowledge base is also used to define the "natural" ordering function for different types.

To ensure the equality testing function is reasonable, a trigger is defined that adjusts the current representation when an element is inserted into a set. If this element is not of a type that is handled properly by the equality testing function in the current representation, then the trigger adjusts the representation to work as expected with the given types.

There is also the issue of the computational efficiency of the current representation. The trigger should strive to pick the best possible representation based on the equality functions specified. For example, if all the elements work best with `equalp`, then a native hash table representation is likely to be the best choice. More complex representations can be selected when multiple equality functions are necessary to exhibit expected behaviour.

Deleting elements is somewhat simpler. In this case, when the number of elements of a certain type in the set reaches zero, we may be able to undo the changes made earlier to the current representation. In this case, care must be taken in specifying when such a trigger is run. It cannot happen before the element is actually deleted; the reconfiguration of the current representation may adversely affect the actual operation's execution. Thus, the trigger that compacts a

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

Figure 4: Example of a simplified trigger for use with dynamic sets (dsets).

representation must be run on some operation following the deletion or immediately following the deletion.

Handling ordering operations is straightforward: if an operation that requires an ordering representation is performed, the trigger verifies that the contents can be ordered in some known way and then changes the current representation to one that supports the ordering operations if it does not already. There is no need to revert the representation back since an ordered representation supports all other set operations. It will only change if an element is inserted that is not compatible with the ordering relation or the equality testing function.

The triggers outlined above are meant to make common situations easy to work with by recognizing them and adjusting the current representation to prevent an error. Nevertheless, some situations will not be recognized because the knowledge base does not contain anything meaningful about them. Some situations are legitimate errors, but others may be something we can recover from. This can be accomplished by invoking a restart when a lookup to the knowledge base returns an unrecognized condition. This restart can query the user to indicate how to correctly manage the situation, saving the information for later.

Triggers are defined as functions that take three arguments: the DADT object the trigger is associated with, the operation the trigger is being called from (as a symbol), and the arguments passed to that operation. It returns two values, the second of which is significant in the current context.[2] When the second value is `t`, it states that the trigger is active; otherwise, it is passive. We do not include the complete code for a trigger – the listing would be too long – but the basic code for the trigger used to compact the representation is given in Figure 4.

3.4 Class definition

Defining a class for DADT object is done using `define--dadt-class`. It is an extension to `defclass` that properly sets up the superclasses and default initialization arguments. It takes a list of superclasses and slots as `defclass`, but allows more class options. Most class options correspond to the slots that hold the metadata for the current representation. There is also an option to provide the default representation of the data object being observed.

The example in Figure 5 defines a class `dset`. It dictates that an instance of a `dset` will be created with the local triggers indicated above and an instance of `type-count` as its only resource. Also, the default type for the current representation is an instance of `unordered-set`.

[2]The first is reserved for a system of recommendations which is not described in this paper.

```
(define-dadt-class dset ()
  ((strict-p :initform nil
             :initarg :strict-p
             :accessor dset-strict-p))
  (:local-triggers insertion-trigger
                   compaction-trigger
                   orderable-trigger)
  (:resources type-count)
  (:default-rep unordered-set))
```

Figure 5: The DADT class definition for sets.

We also define a slot `strict-p` that indicates whether the set object is *strict* or not. A strict instance of `dset` is an instance where the user has explicitly provided certain options meaning that the triggers should not try to adjust for common situations; in this case, the user knows what should be used and is being direct.

3.5 Operation definitions

The operations defined on DADT objects must conform to the expected semantics of the operations as if they were defined for regular objects. Additionally, DADT interface operations may need to perform the extra operations of the DADT protocol. Considerable flexibility is permitted in defining interface operations so that complex situations can be addressed when needed, but reasonable defaults are provided so that the operation can be defined easily.

To demonstrate the flexibility of DADT operation definition, we give some examples with explanations of the features present. The first example is a simple operation where the defaults are adequate. This is given in Figure 6 `(A)`.

The `define-dadt-operation` macro takes four arguments: the name of the DADT class the operation is defined for, the name of the function, a lambda list suitable for `defmethod` and an optional body. In this case, the body has not been given, meaning the defaults will be used. This defines a method on the `member-p` function with the given lambda list. The arguments specialized to the name of the DADT class are assumed to be the *significant* arguments, that is, the arguments representing DADT objects whose current representations must be passed to the actual interface operation for the operation to be executed correctly.

The body of the method will perform the following actions (see Figure 2):

1. Run the triggers for each significant DADT object.
2. Take the initial measurement for each significant DADT object.
3. Execute the actual interface operation – which is assumed to be a method on the same generic function – substituting the current representation of each significant DADT object, saving the results.

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

;; (B)
(define-dadt-operation dset insert-element
    ((set dset) elem)
  (declare* (returns (self set))))

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

;; (D)
(define-dadt-operation dset filter ((set dset) fn)
  (declare* (manual t))
  (let ((new (make-dset)))
    (iterate-set set
                 #'(lambda (e)
                     (when (funcall fn e)
                       (insert-element new e))))
    new))
```

Figure 6: Interface operation examples for a DADT.

4. Take the final measurement for each significant DADT object.

5. Return the results from the actual operation.

The default is to assume that the actual operation only returns a single value that is not of the type indicated by the first argument to the macro (in this case, of type `dset`).

The issue of capturing return values is more difficult than would first appear. The actual operation can return zero or more values, but it is the nature of each value that is the problem. A value returned may be one of the current representations, in which case its DADT object should be updated. Another possibility is that a value may be a new object that should be wrapped in a DADT object. The last case is that the value is passed along without being examined or altered.

Properly specifying these types of values is done using an extended declaration for the function definition, called `declare*`. An example is given in Figure 6 `(B)`. The `returns` specification indicates how many values the actual operation returns and how to alter them. In the example given, only one value is returned, but it is the current representation of the `set` parameter, which is indicated by the `(self set)` specifier. This means that the current representation of the DADT object `set` is updated with the first (and only) value returned by the actual interface operation.

When more control is needed over how an interface operation is performed, the `manual` specification can be provided to the `declare*`, which overrides the auto-generated body and allows the user to define the method body. When this is done, each stage of the protocol is available by way of a local function of the appropriate name.

Suppose we want to define the `delete-element` to run the triggers as the last step in the protocol instead of the first. (We may want to do this so that we could adjust the compaction trigger such that the element to be removed is actually removed before the trigger is run.) An example of this is given in Figure 6 `(C)`. The local functions `run-triggers`, `measure-initial` and `measure-final` allow us to run each extra operation directly. We call the actual operation manually, passing the current representation of the only significant DADT object (the `(dadt-rep set)` form), save the result then perform the final measurement stage and then run the triggers. The value of the actual operation is then returned.

Finally, we can define operations and ignore the DADT protocol when necessary. Figure 6 `(D)` shows an implementation of the `filter` function, which takes a set and a predicate function. It creates a new set containing the elements e of `set` where `(funcall fn e)` is non-`nil`. Note that the the protocol is not needed here because the configuration of the newly created set is handled by the calls to `insert-element`.

The ability to control how the protocol is performed for each operation is very useful for non-standard cases, but also for optimization and debugging. Although not shown here, it is also possible to redefine each stage locally to inject code for debugging purposes or to alter the basics of the protocol to handle particularly difficult cases.

4. OPTIMIZATIONS

Augmenting objects and operations in the fashion outlined above, unsurprisingly, leads to code that imposes significant overhead. General statements regarding the exact overhead cannot be made in a reliable fashion because the overhead is dependent on what resources and triggers are defined for each DADT object.

Looking at a DADT object, we note that there may come a time during its lifetime where the extra work done by the DADT operations is of no help; the object may have stabilized in that no changes have been made to the current representation and information collected about the use of the object is no longer needed. At such time, the overhead of the DADT protocol is unnecessary.

Recognizing that the overhead of the DADT protocol can be significant and that its usefulness waxes and wanes, a system for controlling the application of the protocol has been devised so that the extra work can be avoided. The system can control both the generation of code for the operations as well as the effect the operations have on individual objects. For the operations, the control comes at evaluation/compile time, whereas objects can further be controlled at run time.

The system for controlling the protocol is done through directives. We define some *standard directives* that are al-

ways present. These directives dictate what is to be performed by the protocol at any given time. They can be specified at a global level, for individual operations, DADT classes, and most importantly, for individual objects. The standard directives correspond with the extra operations: `run-triggers`, `measure-initial` and `measure-final`. We also provide a directive `dadt-protocol`, which is an alias for all three. Each directive is either associated with the value `t`, `nil` or no value.

Additionally, we define the *directive environment* as follows. If a directive has no value in an individual object, look in the global environment. If a directive has no value for a DADT class, look in the global environment. If a directive has no value in the global environment, it is assumed to be `nil`. The global environment is initialized with each of the standard directives associated with the value `t`.

For operations, the standard directives dictate how the operation is to be carried out. Within the `declare*` form of a DADT operation, each standard directive can be given as a specifier (defaulting to `t` if not provided). If a directive is associated with `nil` in the `declare*` form, then the code for that stage is not generated for that operation.

If the code for a stage is generated, the value for the directive corresponding to the stage is looked up when the stage is executed. When there is only one significant DADT object for the operation, the object is queried; if there is more than one significant DADT object, the corresponding DADT class is queried. In either case, the stage is executed when the value associated with the corresponding directive is `t` and not executed if it is `nil`. (Note that it will never be without one of these two values in the directive environment.)

Directives provide a way to make run time optimizations at the individual object level. If we can determine that an object is, for the most part, not taking advantage of the extra operations, we can set the standard directives within the object to `nil` and the protocol will no longer be run on that object. Such an optimization does not necessarily preserve the semantics of every operation: it is possible that an operation will behave differently when applied to an object that is employing the DADT protocol versus one that is not. A simple example can be seen with the sets described above. If the trigger that reconfigures the set based on the elements inserted is not run, then any new elements inserted may not result in the expected behaviour (strings being tested using `eql` instead of `equal`, for example).

We justify this by noting that high-level optimizations of this kind generally take place on already working programs. Thus, some understanding of the problem likely exists before any large-scale optimization is applied. The approach to program development with the DADT protocol is one of close interaction with the program in order to facilitate rapid realization of an idea. In short, the DADT protocol was designed for working toward a specialized program by starting out with a very general one. Run time optimizations based on statistical inference are provided as another tool for creating sufficiently useful programs.

Due to the wide variance possible in overhead, it is meaningless to say much about the execution speed of DADT operations when triggers and resources are defined for DADT objects, except that it will be slower than without triggers and resources. To give some idea as to the overhead, we will examine the overhead involved in executing operations when the DADT object has no triggers and no resources. When no triggers and no resources are defined for a DADT object, the overhead is effectively constant, although the constant differs for each possible configuration.

Standard directives		SBCL	ACL	CLISP	Mean
Local	Global				
no value	`t`	7.94	4.87	23.3	11.1
no value	`nil`	1.46	1.55	5.86	2.89
`t`	n/a	2.46	3.24	19.2	8.35
`nil`	n/a	0.06	0.60	4.15	1.61

Table 1: Execution overhead in microseconds (10^{-6}) for operations on DADT objects associated with no triggers or resources. The Lisp implementations used are SBCL 1.0, ACL 8.0 Express Edition, and CLISP 2.4 running on a 2.16 GHz Intel Core Duo.

It should be pointed out that in most cases, a DADT operation will be an elaborate wrapper for an already existing implementation of the desired operation. Hence, we do not seek to address the speed of an already existing implementation, but rather analyze the overhead involved with observing and reacting to the situations surrounding the use of the operation.

Table 1 shows the overhead of the execution time in microseconds of interface operations applied to DADT objects with the value of the standard directives found in four different ways compared to actual operations across three different Lisp implementations. The operations were all applied to the same underlying data and the DADT objects had no resources or triggers associated with them. The values of the standard directives were found in four different ways: with the local values empty (that is, the DADT object has no value for the directive) and the global values being either `t` or `nil`, or the local value being either `t` or `nil`.

The values in Table 1 were obtained by running the following code (where the set is created with suitable directives, if necessary, by the omitted code represented by the ellipsis).

```
(let ((s (...)))
    (time (loop repeat 30000 do (empty-p s)))
    (time (loop repeat 30000 do (size s)))
    (time (loop for i from 1 to 30000 do
             (member-p s i)))
    (time (loop for i from 1 to 30000 do
             (insert-element s i)))))
```

The base value was taken to be the test as performed on a regular set object. The value 30000 was chosen because it provided enough repetitions to account for anomalous times and higher values tended to trigger a garbage collection. The four operations were selected as a sample of simple operations applied to individual sets. The mean overhead across

the four operations provides the results found for each implementation. The code was compiled using the default compiler settings.

The results show that the overhead can change drastically based on the values of the standard directives and where their values are found; such overhead is directly related to the time needed to execute a function call in the Lisp implementation being used (SBCL seems to be quite good when the directives are set locally to `nil`). Thus, over many operations, it will be considerably more efficient to disable the protocol locally. This is not surprising. Indeed, it is considerably more efficient to set the directives locally in either case. As these overhead values are for DADT objects with no resources and triggers, the overhead will likely get much higher when the protocol is enabled in an actual program. We would expect that when the protocol is disabled, the overhead would be nearly constant regardless of what resources or triggers are present since the extra work involving them is not performed.

The overhead can be lessened further by defining the DADT operations in such a way that the code to perform the extra operations is not even present. This means that the run time checks to are not done, further reducing the extra work involved. When the protocol is disabled within a DADT interface operation, the operation becomes little more than a wrapper call. There is great flexibility allowed when it comes to enhancing the performance of DADT operations, although removing the ability of an operation to perform any of the protocol should be done with care.

We see that disabling the protocol has a significant effect, but we must find a way to do this at run time. An obvious solution is to use the protocol itself by defining a trigger that will turn the protocol off for individual objects when a certain condition is detected.

The conditions that dictate when a DADT object should have the protocol disabled locally (that is, when the object should be *stabilized*) are not entirely clear. It is likely that complete information is not available and that a reasonable guess has to be made.

We will build on the previous example of sets. The trigger most likely to alter the current representation is the insertion trigger because sets are going to be populated with elements at some point. (Lots of empty sets aren't generally useful.) However, once a set has "seen" the different types of elements that will populate it, the insertion trigger is not of much use. It is likely that the different types that will be present in a set will be seen relatively early with respect to the insertion of elements. If we program in a functional style, then it is also likely that once a set is populated, it will not be changed again.

Using these assumptions, we can define a trigger that stabilizes an instance of `dset` in the following manner. We count the number of operations performed on the set since the last change triggered by an insertion or deletion. When the number reaches a certain threshold n, we will disable the protocol for that particular object. In other words, when a dynamic set has had n operations performed on it without any changes having taken place to the current representation, we assume that it won't experience anything new or profound in the future and we can effectively remove it from consideration in the DADT interface operations.

```
(defclass call-counter ()
  ((count :initform 0 :accessor counter)))

(defmethod measure-resource-initial
    ((cc call-counter) rep op args)
  (incf (counter cc)))

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

(defun stability-trigger (dset op args)
  (declare (ignore op args))
  (with-resources ((ccount call-counter))
      (dadt-resources dset)
    (when (<= (or (lookup-directive
                       :stability-threshold)
                  most-positive-fixnum)
              (counter ccount))
      (run-ordering-trigger mset)
      (enact mset '(dadt-protocol nil))
      (values nil t))))

(direct (stability-threshold 50))
```

Figure 7: Resources and triggers used to stabilize `dsets`.

Determining a threshold value is likely to rely on some intuition about the task at hand and defining it to be static is probably unwise. Hence, we will define it as a directive so that we can change it when necessary.

There is one problem with this kind of definition: ordering statistics. Stabilizing a `dset` instance in this manner means that it may preclude the ability to support ordering functions. We can get around this problem by having the stability trigger run the trigger that adjusts for an ordering operation.

Some of the code for adding this ability is given in Figure 7.

We define a resource `call-counter` that increments a counter whenever an operation is called.[3] This gets reset to zero when `reset-resource` is called on it. Note that `reset-resource` is called on every resource object associated with a DADT object when the current representation changes.

`stability-trigger` is simple: it first finds the counter resource associated with the dynamic set and checks to see if the counter is greater than or equal to the stability threshold as defined in the global directives; a default of `most-positive-fixnum` is given if the directive has not been defined, which effectively means the trigger will not do any-

[3]More correctly, whenever the initial measurement stage is executed for a DADT interface operation.

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

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

Figure 8: Two programs to compute S. The first takes advantage of the `dset` implementation; the second uses a more traditional approach.

thing.

If the threshold has been reached, it first runs the trigger that adjusts the set in the event of an ordering operation being called. If the set supports ordering functions, then the representation will be adjusted to one that can support those operations before the set is stabilized. It calls `enact`, which takes a DADT object or class and sets the directives to that object or class as provided. In this case, the standard directives are set to `nil`. The function then returns indicating it has changed the current representation. In reality it hasn't done so, but it has turned off the protocol for the object, so further processing of triggers should be avoided.

The call to `direct` (which is actually a macro) sets directives at the global level; here we have chosen a stability threshold of 50. (The syntax for directives is analogous to specifications for `declare`, and `enact` and `direct` are similar to `proclaim` and `declaim`.)

To examine the efficacy of the stability trigger, we will write a simple program that does the following. Given a set D, find the subset S of D where $S = \{s | s \in D, |s| \in D\}$. For simplicity, we assume that D contains numbers and strings, and that the elements of D reside in a file.

Two versions of a program to compute S are given in Figure 8. The first uses `dset`s and the second uses a specific set representation.[4] In order to properly handle equality between numbers and strings, we must make a set for each type of element in the "vanilla" example, which results in more code. It also requires the user to specify two pieces of information about each set created: the kind of structure to use and how to test for equality.[5]

Under SBCL 1.0, `find-s-dset` takes approximately 45% longer than `find-s-vanilla` when the stability threshold values is set to 50 and $|D| = 100000$. This is reasonably close to what we might expect. Although slower, `find-s-dset` is considerably shorter to write and arguably more "natural" in terms of computing with sets due to the presence of a `filter` function.

It should be noted that `filter` is not used in `find-s-vanilla` because it does not exist. `filter` is a DADT operation that does not call an actual interface operation; instead, it implements the algorithm directly (see Figure 6). In order for a "vanilla" `filter` function to exist, it would have to combine sets properly and the implementations that the DADT set uses do not provide these operations.

5. DISCUSSION

The approach of dynamic abstract data types is not meant to optimize the run time of production code — as the data clearly indicates — so much as it is a foray into optimizing developer time. The example program in Figure 8 shows that taking advantage of familiar and common situations means that some programs can be very simple to write and have acceptable performance.

Gabriel and Goldman [7] recently proposed the idea of conscientious software, asserting that software should aid in its installation and customization, as well as adapting to new situations. Dynamic ADTs provide a way to achieve part of this idea by allowing a program to observe the use of its data objects and employ the information gathered. This is done by introducing more unknowns to the program initially and specializing over time.

Filling in the unknowns means addressing them directly by using errors as a source of information. In our implementation, we have made a conscious effort to provide execution paths that can continue the running of the program using the Lisp restart system. This, in effect, means the program/run time can learn from its mistakes. Our current implementation is not very robust — it only learns from rudimentary situations — but we are working on detecting and handling more complex conditions in an intuitive manner.

Rinard [17] has asserted that flawed software has enormous value. The `dset` DADT adds further support to this claim by permitting a program to work in a familiar way even in

[4]The representation is taken from Gary King's CL-containers package [9].

[5]In fact, the equality test can affect the choice of structure. In this example, the user must know that the `associative-array` class uses a standard hash table as the underlying representation, so only one of the standard equality functions will work.

the absence of complete information. Looking at the way `dset` works, one may notice that it allows for the incremental creation of a data structure suitable for some task without devising the particulars ahead of time. As we know more about the problem, the software "knows" more about the problem and helps the user deal with the complexity in a pragmatic and usable fashion. It doesn't necessarily solve every aspect of the problem, but it mitigates some of the labour involved in doing so by eschewing the notion of building one part before moving on to another.

Some of the techniques behind changing data representations for a single object can be found in the programming language SETL [19]. SETL is a programming language that focuses on the use of sets and uses a form of automatic data structure selection, but it does not change representations at run time. Instead, it uses static analysis by way of the compiler [18]. Development of SETL focused on determining efficient representations for sets based on what was stated in the program rather than what happened at run time. Low [12] also did work on techniques for choosing data representations automatically, but also did not focus much on dynamic aspects. SETL and Low's thesis provided some ideas for managing multiple data representations in the DADT protocol. Richardson [16] has investigated automatic changes in functional programs by program transformation. An analysis of multiple data representations within a computation can be found in [22]. A theoretical approach to extending data types with dynamic features can be found in [24].

To a limited extent, the "tables" of Symbolics Common Lisp are similar to DADTs.[6] Tables would change internal representation based on the options given when the table was created [1]. For example, a table with few elements that are tested with `eq` may result in faster lookups using an alist than a hash table. Changing representations primarily for performance reasons is more a function of the program than the data. Tables in Symbolics Common Lisp encompass some of the abilities of DADTs, but lack the dynamism afforded by extensible monitoring capabilities. Furthermore, Tables are an attempt to work across all programs on a specific data type whereas DADTs can be used to on multiple data types, at different stages of development, and offer the option to be disabled.

The KIDS [20] and DTRE [4] systems for semi-automatic program development and transformation contain some of the same ideas as the DADT protocol, although again they are more focused on writing static program specifications. The tuple space of Linda implementations provides for an opportunity to determine what data layout should be used for a given data set. [15] shows that this is possible, also through static analysis.

More recently, work has been done on adjusting programs while they are running. [10] looks at delaying code generation until load time. The results show that this must be done with care, but in some circumstances it can be beneficial. The ideas of using monitoring information to guide decisions are used to decide what parts of the code to optimize, similar to a just-in-time compiler. Acar's work on self-adjusting computation [2] looks at remembering parts of a computation such that changes to the input propagate through the computation without having to recompute everything. In both cases, past experiences are remembered in order to make use of them later.

Dynamic algorithm selection is demonstrated by the GNU MP library. Different algorithms for the same operation are used depending on the number of machine words the operands occupy and the nature of the operands [8]. Additionally, the parameters dictating when one algorithm is chosen over another are user-configurable at compile time. Each algorithm for a given operation performs the same task, but the efficiency is different for each one depending on the input. A prototyping tool for dynamic algorithm selection using reinforcement learning that works on pre-existing binaries is presented in [3] and demonstrates some of the same principles as dynamic ADTs.

In the future, it would be interesting to explore a dynamic ADT implementation of the sequences interface in Common Lisp. Using the profiling features of dynamic ADTs, it would be possible to help determine specialized representations for some sequences, such as specific dimensions for vectors. It may also be beneficial to combine unit tests with the profiling information to deduce some traits of the expected input. This would likely require the test suite to interact with the dynamic ADT in some fashion to prevent the accumulation of misleading data. (For example, the DADT may need to know if a test is supposed to pass or fail.)

We find that the dynamic abstract data type is a way of blurring the boundary between the programmer and the program in interesting ways. The tenets behind the DADT protocol were meant to mirror the basic workflow of program development: start with an idea and get something working so that the problems can be found and dealt with. In this sense, we took inspiration from the idea of *flow* put forth by Csikszentmihalyi [6], specifically that flow tends to be characterized by immediate feedback to one's actions. Looking for common occurrences in the use of certain constructs by the developer allows for some aspects of the program to be left unspecified meaning that fewer details are needed for such feedback to be given. This may end up simply delaying the inevitable, but the path taken may be more enjoyable.

6. CONCLUSION

We have outlined dynamic abstract data types, a construct and protocol for monitoring and altering data objects at run time that can make some programming tasks simpler by alleviating the need to specify certain details about a data object's structure; the overhead involved can be mitigated by reasonable heuristics. This was demonstrated by the implementation of an abstract data type for sets that restructures its instances based on how they are used. Additionally, it allows for the profiling of data in a manner analogous to profiling of code, which can be used to specialize the program. With the profiling data accessible at run time, it presents opportunities for dynamic optimization.

[6] According to Barry Margolin [14], the original design of tables involved changing the class of the table object, but this was found to be problematic. Instead, it was redesigned so that a slot in a hash table object contained the "actual" representation of the hash table.

7. REFERENCES

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

[2] U. A. Acar, G. E. Blelloch, M. Blume, and K. 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] W. Armstrong, P. Christen, E. McCreath, and A. P. Rendell. Dynamic algorithm selection using reinforcement learning. *International Workshop on Integrating AI and Data Mining (AIDM)*, 0:18–25, 2006.

[4] L. Blaine and A. 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.

[5] T. H. Corman, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.

[6] M. Csikszentmihalyi. *Creativity: Flow and the Psychology of Discovery and Invention*. HarperCollins, New York, 1996.

[7] R. P. Gabriel and R. 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.

[8] T. Granlund. The GNU MP manual. `http://www.swox.com/gmp/` (valid 20 Feb 2007).

[9] G. King. CL-containers. `http://common-lisp.net/project/cl-containers/` (valid 20 Feb 2007).

[10] T. Kistler and M. Franz. Continuous program optimization: A case study. *ACM Transactions on Programming Languages and Systems*, 25(4):500–548, 2003.

[11] D. Knuth. *The Art of Computer Programming: Searching and Sorting*, volume 3. Addison-Wesley Professional, 1998.

[12] J. Low and P. Rovner. Techniques for the automatic selection of data structures. In *POPL '76: Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 58–67, New York, NY, USA, 1976. ACM Press.

[13] N. Magaud. *Theorem Proving in Higher Order Logics*, volume 2758/2003, chapter Changing Data Representation within the Coq System, pages 87–102. Springer, 2003.

[14] B. Margolin. CLOS and C++. Article b06c0a64.0311201123.4302e51a@posting.google.com in Usenet group comp.lang.lisp (valid 20 Feb 2007), 20 Nov 2003.

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

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

[17] M. Rinard. Acceptability-oriented computing. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 221–239, New York, NY, USA, 2003. ACM Press.

[18] E. Schonberg, J. T. Schwartz, and M. Sharir. An automatic technique for selection of data representations in setl programs. *ACM Transactions on Programming Languages and Systems*, 3(2):126–143, 1981.

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

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

[21] P. 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.

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

[23] G. Wozniak. Dynamic abstract data types. `http://www.csd.uwo.ca/~wozniak/dadt` (valid 20 Feb 2007).

[24] E. Zucca. From static to dynamic abstract data-types: an institution transformation. *Theoretical Computer Science*, 216(1-2):109–157, March 1999.