Structures for Manipulating Proposed Updates In Object-Oriented Databases*

Michael Doherty
Computer Science Department
University of Colorado
Boulder, CO 80309-0430
doherty@cs.colorado.edu

Richard Hull
Computer Science Department
University of Colorado
Boulder, CO 80309-0430
hull@cs.colorado.edu

Mohammed Rupawalla
Computer Science Department
University of Colorado
Boulder, CO 80309-0430
mohammed@cs.colorado.edu

Abstract

Support for virtual states and deltas between them is useful for a variety of database applications, including hypothetical database access, version management, simulation, and active databases. The Heraclitus paradigm elevates delta values to be "first-class citizens" in database programming languages, so that they can be explicitly created, accessed and manipulated.

A fundamental issue concerns the trade-off between the "accuracy" or "robustness" of a form of delta representation, and the ease of access and manipulation of that form. At one end of the spectrum, code-blocks could be used to represent delta values, resulting in a more accurate capture of the intended meaning of a proposed update, at the cost of more expensive access and manipulation. In the context of object-oriented databases, another point on the spectrum is "attribute-granularity" deltas which store the net changes to each modified attribute value of modified objects.

This paper introduces a comprehensive framework for specifying a broad array of forms for representing deltas for complex value types (tuple, set, bag, list, o-set and dictionary). In general, the granularity of such deltas can be arbitrarily deep within the complex value structure. Applications of this framework in connection with hypothetical access to, and "merging" of, proposed updates are discussed.

1 Introduction

Support for virtual database states and deltas between them is useful for a variety of database applications, including hypothetical access to proposed updates, version management, active databases, and simulation. The Heraclitus paradigm [HJ91, JH91, GHJ96] elevates delta values to be "first-class citizens" in database programming languages (DBPLs), so that programmers can explicitly create, access and manipulate them. A relational version of Heralictus called Heraclitus[Alg,C] has been implemented [GHJ+93, GHJ96]. We are currently implementing a version of Heraclitus for object-oriented databases (OODBs); this is called the Heraclitus[OO] DBPL, abbreviated as "H2O DBPL" [BDD+95, DHDD95].

To illustrate, we consider how Heraclitus provides hypothetical access to proposed updates. Suppose that the current database state is DB, and that ϵ is an expression denoting an update. The *deltafication* of ϵ is the expression '`[< ε >]`'. Consider the expression '`d = [< ε >]`', where `d` is a variable ranging over deltas. Evaluating this expression in state DB makes no changes to DB, and assigns to `d` a delta that corresponds to the net effect on DB that executing ϵ would have. The `when` operator in Heraclitus permits hypothetical access to a database. For example, if γ is a side-effect free expression (e.g., a query), then the expression 'γ `when d`' yields the value that γ *would* take in the state that would result from applying the value of `d` to DB (i.e., the result of executing ϵ on DB). This permits relatively efficient hypothetical access to the effect of update ϵ, without modifying the underlying database state.

As an example of the application of Heraclitus functionality, consider how a telephone company might support requests from customers to change their telephone service. Such *service orders* are typically made on one date, to become effective at some future date. Several pending service orders might be related in some way. In current systems it can be quite expensive to detect conflicts between related service orders, because the net effect of a service order is buried in code and *ad hoc* representations. As detailed in Section 2 below, if the net effect of each service order is represented as a delta, then such conflicts can be detected in a relatively efficient manner. Furthermore, in some cases there may be ways to "resolve" the conflicts by using special merge operators on deltas.

A fundamental issue in the Heraclitus paradigm concerns how delta values associated with a proposed update are represented. Suppose that Δ is the value

*This research was supported in part by NSF grant IRI-931832, and ARPA grants BAA-92-1092 and 33825-RT-AAS.

SIGMOD '96 6/96 Montreal, Canada
© 1996 ACM 0-89791-794-4/96/0006...$3.50

associated with [< ϵ >] when evaluated in state DB. As detailed in Sections 2 and 4 below, Δ may become "out of date", or "lose its intended meaning", if the underlying state is changed. That is, if state DB is updated to state DB', then applying Δ to DB' may not have the same effect as executing ϵ in state DB'. There is a trade-off between the expressive power (i.e., the ability to retain intended meaning) of a form of delta representation and the ease of access and manipulation of that form. One end of the spectrum is to use code-blocks to represent delta values; in this case the delta value accurately captures the intended meaning of a proposed update even if the underlying database state is changed. However, access to and manipulation of code-block delta values can be prohibitively expensive. In the context of object-oriented databases (OODBs) another point on the spectrum is "attribute-granularity" deltas. Speaking loosely, in an attribute-granularity delta, changes to an object are represented by giving the new values for modified attributes. It is much cheaper to access and manipulate attribute-granularity deltas than code-block deltas, but they may lose their intended meaning if the underlying database state is changed.

This paper introduces a comprehensive framework for specifying a broad array of forms of delta representations in the context of OODBs. The focus is on delta representations that are data-like (i.e., sets of concrete structured values) rather than code-like (e.g., code-blocks or expressions). Primary emphasis is given to complex value types, for which the granularity of delta representations can be at an arbitrarily deep level within the complex value.

Section 2 gives more detail concerning how the Heraclitus paradigm can be used in connection with the access and manipulation of proposed updates and informally presents several forms of delta representation. Section 3 gives background about the Heraclitus paradigm and briefly considers the prototype H2O DBPL currently being implemented. Section 4 presents the framework for specifying a broad array of forms of delta representation for complex value types.

2 Motivating Examples

This section illustrates certain aspects of the Heraclitus framework, to provide background and motivation for the research presented here. We extend the discussion of the Introduction by considering four scenarios based on the detection of conflicts between telephone service orders (see also [DDD+96]). Scenarios 1 and 2 show how deltas systematically provide hypothetical access to future states of the system and how such access is helpful in detecting conflicts between service orders as early as possible. These scenarios use a relatively coarse but efficient form of delta. Scenarios 3 and 4 introduce delta forms that are more semantically

```
Line (o17 : ⟨ cust : 'Molly', number : '555',
              features : {'vm'} ⟩),
Line (o23 : ⟨ cust : 'Tom', number : '123',
              features : {'vm'} ⟩),
Line (o48 : ⟨ cust : 'Sam', number : '987',
              features : {'vm'} ⟩),
Line (o52 : ⟨ cust : 'Gwen', number : '432',
              features : { } ⟩)
```

Figure 1: Four objects in database state DB_a

expressive. These forms provide a natural mechanism for computing the net effect of two proposed updates through a mechanical "merging" operation on deltas, thus yielding more efficiency An additional short example illustrates how arithmetic functions can be incorporated into a delta form

For the scenarios, we suppose that the database schema[1] consists of a single class **Line**, instances of which represent phone lines, with the following specification[2]:

```
Line: tuple (cust: string,
             number: string,
             features: set(string) )
```

where **cust** holds a customer name and **features** holds the set of features that the line currently supports (e.g., custom-ringing (cr), call-screening (cs), call-forwarding (cf) and voice-mail (vm)).

A portion of the database state (DB_a) representing the situation at the beginning of the scenarios (Day 0) is given in Figure 1. In this depiction, we list some of the objects of class **Line**, along with their values.

We now consider the four scenarios in turn. The first scenario illustrates the usefulness of deltas to support hypothetical queries against proposed updates.

Scenario 1: [Conflicts Due to Future Commitments] Suppose that the features custom ringing and call screening are incompatible Consider the following sequence of events regarding Molly's line (number 555):

Day 0:	Molly has voice mail
Day 10:	She requests custom ringing, effective Day 30
Day 20:	She requests call screening, effective Day 40
Day 30:	Custom ringing installed
Day 40:	Attempt to install call screening fails

When Day 40 arrives, the conflict between custom ringing and call screening is detected and the installation of call screening fails. At this point the phone company

[1] This schema and the corresponding database state are obviously quite simplified for the purpose of illustration. Much larger and more intricate examples arise in telecommunications and other applications where proposed updates do not take effect immediately (e.g , scheduling of transportation or flow of inventory).

[2] In this paper we use a pidgin form of the O_2 language to specify types.

must inform Molly that she cannot have call screening after all.

The outcome of this scenario is unfortunate, because the phone company had all of the relevant information on Day 20, when Molly placed the second request. However, in many application environments the impacts of proposed updates are buried in code and/or *ad hoc* representations, and not easily accessible. □

We now describe one approach by which the conflict between the two requests can be detected on Day 20, when the second request is made. As mentioned in the Introduction, an arbitrary expression ϵ can be *deltafied* as '[< ϵ >]'. Evaluation of this expression in a given database state DB yields a *delta value* that captures the intent of ϵ (at least in the context of state DB), but does not change the state DB.

Scenario 1 (continued): [Using Look-ahead to Prevent Conflicts] Let ϵ_1 be the expression that modifies the database state to reflect the installation of custom ringing to Molly's line, and ϵ_2 be analogous for call screening. On Day 10, when Molly makes her first request, the system can compute Δ_1^{og} = [< ϵ_1 >] in the current state, namely in state DB_a (see Figure 1). Under a very simple and inexpensive form of delta representation, called "object-granularity" (indicated by the *og* superscript), this yields the first delta value shown in Figure 2. This delta indicates how the value of o_{17} needs to be changed in order to reflect the proposed update ϵ_1.

Suppose that q is a query that checks whether a customer request r applied to the current database state will lead to a constraint violation. On Day 20, if q is evaluated with r corresponding to the request for call screening, the answer will be "no conflict", because ϵ_1 has not yet been executed against the state. On the other hand, the query q **when** Δ_1^{og}, which corresponds to asking q against the hypothetical future state of Day 30 or later, will yield "conflict". Furthermore, because Δ_1^{og} is stored essentially as a data value, the answer to q **when** Δ_1^{og} is typically much cheaper to compute than performing some computation involving q and ϵ_1. □

The preceding scenario shows in principle the usefulness of providing hypothetical access to the future states of the system, based on proposed updates (i.e., service orders) that have been made to date. We now illustrate how deltas can be used to provide systematic support for hypothetical access against more than one proposed update.

Scenario 2: [Look-ahead with Multiple Deltas] Consider the following sequence of events with respect to Tom's line (number 123):

Day 0:	Tom has voice mail
Day 10:	He requests call screening, effective Day 30
Day 20:	He requests to change his number (to 999), effective Day 40
Day 30:	Call screening installed
Day 40:	Number changed to 999

Let ϵ_3 be the expression that calls for adding call screening to Tom's line, and ϵ_4 be the expression that changes his phone number to 999. We can compute

$$\Delta_3^{og} = [< \epsilon_3 >]$$
$$\Delta_4^{og} = [< \epsilon_4 >] \textbf{ when } \Delta_3^{og}$$

We use [< ϵ_4 >] **when** Δ_3^{og} rather than the simpler [< ϵ_4 >], because ϵ_4 will be executed on Day 40, in the context of all updates that occurred before then, i.e., in the context of the update ϵ_3, the net effect of which is captured by Δ_3^{og}.

Continuing with object-granularity deltas, the values of Δ_3^{og} and Δ_4^{og} are shown in Figure 2. These two deltas can be used to ask hypothetical queries against the future as reflected by proposed updates in the system as follows:

On Days 10-20;	For Days 30+:	**when** Δ_3^{og}
On Days 20-30;	For Days 30-40:	**when** Δ_3^{og}
	For Days 40+:	**when** $\Delta_3^{og}!\Delta_4^{og}$
After Day 30;	For Days 40+:	**when** Δ_4^{og}

(The binary operator smash, denoted '!', corresponds to a form of composition of deltas, and is discussed in Sections 3 and 4. In this very trivial example, $\Delta_3^{og}!\Delta_4^{og} = \Delta_4^{og}$.) □

The advantage of object-granularity deltas is their conceptual simplicity and computational efficiency. However, they are quite limited in terms of expressive power. The next scenario illustrates this and shows how this limitation can be overcome by using more "precise" delta forms.

Scenario 3: [Loss of Intended Meaning] For this scenario, consider the following sequence of events concerning Sam's line (number 987):

Day 0:	Sam has voice mail
Day 10:	He requests call screening, effective Day 40
Day 20:	He requests a new number (111), effective Day 30
Day 30:	Number changed to 111
Day 40:	Custom screening installed

Let us first attempt a naive application of the approach used in the previous example. Let ϵ_5 be an expression that will add call screening to Sam's line, and ϵ_6 an expression that changes his phone number to 111. Following the previous example, we might compute

$$\Delta_5^{og} = [< \epsilon_5 >]$$
$$\Delta_6^{og} = [< \epsilon_6 >]$$

(See Figure 2.) Here [< ϵ_6 >] is not evaluated under a

Scenario 1:

$$\Delta_1^{og} = \{\ mod\ \texttt{Line} : \{\ mod\ o_{17} : \langle\ \texttt{cust} : \text{'Molly'},\ \texttt{number} : \text{'555'},\ \texttt{features} : \{\text{'vm'}, \text{'cr'}\}\rangle\ \}\ \}$$

Scenario 2:

$$\Delta_3^{og} = \{\ mod\ \texttt{Line} : \{\ mod\ o_{23} : \langle\ \texttt{cust} : \text{'Tom'},\ \texttt{number} : \text{'123'},\ \texttt{features} : \{\text{'vm'}, \text{'cs'}\}\rangle\ \}\ \}$$

$$\Delta_4^{og} = \{\ mod\ \texttt{Line} : \{\ mod\ o_{23} : \langle\ \texttt{cust} : \text{'Tom'},\ \texttt{number} : \text{'999'},\ \texttt{features} : \{\text{'vm'}, \text{'cs'}\}\rangle\ \}\ \}$$

Scenario 3:

$$\Delta_5^{og} = \{\ mod\ \texttt{Line} : \{\ mod\ o_{48} : \langle\ \texttt{cust} : \text{'Sam'},\ \texttt{number} : \text{'987'},\ \texttt{features} : \{\text{'vm'}, \text{'cs'}\}\rangle\ \}\ \}$$

$$\Delta_6^{og} = \{\ mod\ \texttt{Line} : \{\ mod\ o_{48} : \langle\ \texttt{cust} : \text{'Sam'},\ \texttt{number} : \text{'111'},\ \texttt{features} : \{\text{'vm'}\}\rangle\ \}\ \}$$

$$\Delta_5^{ag} = \{\ mod\ \texttt{Line}\ \{\ mod\ o_{48} : \{\ mod\ \texttt{features} : \{\text{'vm'}, \text{'cs'}\})\ \}\ \}\ \}$$

$$\Delta_6^{ag} = \{\ mod\ \texttt{Line} \cdot \{\ mod\ o_{48} : \{\ mod\ \texttt{number} : \text{'111'}\ \}\ \}\ \}$$

$$\Delta_5^{ag}\ !\ \Delta_6^{ag} = \Delta_6^{ag}\ !\ \Delta_5^{ag} = \left\{\ mod\ \texttt{Line} : \left\{\ mod\ o_{48} : \left\{ \begin{array}{l} mod\ \texttt{number} : \text{'111'} \\ mod\ \texttt{features} : \{\text{'vm'}, \text{'cs'}\} \end{array} \right\}\right\}\right\}$$

Scenario 4:

$$\Delta_7^{ag} = \{\ mod\ \texttt{Line} : \{\ mod\ o_{52} : \{\ mod\ \texttt{features} : \{\text{'vm'}\}\ \}\ \}\ \}$$

$$\Delta_8^{ag} = \{\ mod\ \texttt{Line} : \{\ mod\ o_{52} : \{\ mod\ \texttt{features} : \{\text{'cs'}\}\ \}\ \}\ \}$$

$$\Delta_7^{ag}\ !\ \Delta_8^{ag} = \Delta_8^{ag} \neq \Delta_7^{ag} = \Delta_8^{ag}\ !\ \Delta_7^{ag}$$

$$\Delta_7^{sag} = \{\ mod\ \texttt{Line} : \{\ mod\ o_{52} : \{\ mod\ \texttt{features} : \{ins\ \text{'vm'}\}\ \}\ \}\ \}$$

$$\Delta_8^{sag} = \{\ mod\ \texttt{Line} : \{\ mod\ o_{52} : \{\ mod\ \texttt{features} : \{ins\ \text{'cs'}\}\ \}\ \}\ \}$$

$$\Delta_7^{sag}\ !\ \Delta_8^{sag} = \Delta_8^{sag}\ !\ \Delta_7^{sag} = \{\ mod\ \texttt{Line} : \{\ mod\ o_{52} : \{\ mod\ \texttt{features} : \{ins\ \text{'vm'}, ins\ \text{'cs'}\}\ \}\ \}\ \}$$

Figure 2: Deltas associated with the four scenarios[3]

when, because ϵ_6 is to be applied on Day 30, before Day 40 when ϵ_5 is to be evaluated.

Under this approach, these object-granularity deltas will not always give the correct hypothetical view of the future. Suppose that on Day 35 the query "what is Sam's phone number on Day 45?" is asked. Following the approach of the previous example, we would ask the query "what is Sam's phone number?" **when** $\Delta_6^{og}\ !\ \Delta_5^{og}$. The value of the **number** attribute of o_{48} under the clause **when** $\Delta_6\ !\ \Delta_5^{og}$ is 987 (see Figure 2), and so the answer to this query is 987. However, if we trace through the sequence of steps in the actual scenario, we see that Sam's phone number on Day 45 will actually be 111.

The problem is that Δ_5^{og} is the value of $[<\epsilon_5>]$ on state DB_a, but it is not the value of $[<\epsilon_5>]$ on the state of the database on Day 39, just before ϵ_5 is to be applied. This is the price of using a value-based delta value such as Δ_5^{og} rather than simply storing the expression ϵ_5: a value-based delta value may "become out of date", or "lose its intended meaning" if the underlying database changes. To correct this, Δ_5^{og} should be replaced by

$$\Delta_5^{og\prime} = [<\epsilon_5>]\ \texttt{when}\ \Delta_6^{og}$$

after Δ_6^{og} is created on Day 20. □

In this scenario, $[<\epsilon_5>]$ had to be recomputed because it became "out of date" due to the changes to the database state represented by Δ_6^{og}. In this case, we say that Δ_5^{og} *conflicts with* Δ_6^{og}. Although the issue of conflict detection is beyond the scope of this paper (see [DH95, DHDD95]), a conservative test for conflict is provided by smash commutativity (as will be seen in Section 3). In particular, if $\Delta_a\ !\ \Delta_b \neq \Delta_b\ !\ \Delta_a$, then Δ_a and Δ_b conflict. Thus, Δ_5^{og} and Δ_6^{og} are viewed as conflicting.

Intuitively, the updates represented by ϵ_5 and ϵ_6 do not conflict, since they affect independent pieces of data. However, Δ_5^{og} and Δ_6^{og} do interfere, because they carry superfluous information that is not relevant to the corresponding update. We continue this example by introducing a more "precise" delta form called *attribute-granularity* that carries less superfluous data, and is therefore less likely to introduce spurious update conflicts.

Scenario 3 (continued): [Attribute-granularity Deltas] The values of the attribute-granularity deltas

$$\Delta_5^{ag} = [<\epsilon_5>]$$
$$\Delta_6^{ag} = [<\epsilon_6>]$$

are shown in Figure 2. In this case, $\Delta_5^{ag}\ !\ \Delta_6^{ag} = \Delta_6^{ag}\ !\ \Delta_5^{ag}$, and so they do not interfere with each other. (The value of this smash is also shown in Figure 2.) Thus, if using attribute-granularity, the expression $[<\epsilon_5>]$ does not need to be re-evaluated on Day 20 of Scenario 3. □

In the final scenario, we illustrate the usefulness of an

[3]The deltas shown here modify a single object from a single class. In general, a delta may modify several classes, and include insertions, deletions, and modifications to more than one object in those classes.

even more refined form for deltas.

Scenario 4: [Structured Attribute-granularity] Consider the following sequence of events with respect to Gwen's line (number 432):

Day 0: Gwen has basic service
Day 10: She requests voice mail, effective Day 40
Day 20. She requests call screening, effective Day 30
Day 30: Call screening installed
Day 40: Voice mail installed

This is similar to Scenario 3, except that in this case both requested updates affect the set of features associated with Gwen's phone. Let ϵ_7 (respectively ϵ_8) be an expression that adds voice mail (call screening) to Gwen's line. If using attribute-granularity deltas, we would compute $\Delta_7^{ag} = [< \epsilon_7 >]$ on Day 10 and $\Delta_8^{ag} = [< \epsilon_8 >]$ on Day 20. These values are shown in Figure 2. Their smash does not commute, and they conflict with each other. To maintain up-to-date deltas, the value $\Delta_7^{ag\prime} = [< \epsilon_7 >]$ `when` Δ_8 would need to be computed on Day 20.

Similar to Scenario 3, the basic operations involved (adding voice mail and adding call screening) are independent. This suggests a further refinement of the form of delta values that is called, generically, "structured-attribute-granularity" In this form, modifications to the `features` attribute can be specified in terms of individual insertions and deletions to the set. Deltas Δ_7^{sag} and Δ_8^{sag} with this form corresponding to $[< \epsilon_7 >]$ and $[< \epsilon_8 >]$ are shown in Figure 2. These deltas do not conflict, and so Δ_7^{sag} does not need to be recomputed. □

The following example illustrates that delta forms might make use of various modification functions, as appropriate for a particular data type.

Example 2.1: [Attribute-granularity with Arithmetic] Assume for this example that a Terminal Box (Tbox) is a router that can route up to 100 lines. Suppose now that the class `Tbox` has the following specification:

```
Tbox: tuple (id: string,
             slots_reserved: int,
             slots_active: int,
             slots_total: int)
```

Here, a slot is reserved if it will be used to support a line that has been promised for some future date. For this example, an appropriate form for deltas is to permit arithmetic expressions of the form $+n$ or $-n$ for the three integer fields of `Tbox`. The following delta corresponds to making three previously reserved lines active:

$$\left\{ \begin{array}{l} mod\ \texttt{Tbox} : \\ \left\{ \begin{array}{l} mod\ o_{84} : \left\{ \begin{array}{l} mod\ \texttt{slots\_active} : +3 \\ mod\ \texttt{slots\_reserved} : -3 \end{array} \right\} \end{array} \right\} \end{array} \right\}$$

Operation	Algebra	Language
creation		
deltafication	$trace(\epsilon, PS, DB)$	$[< \epsilon >]$
access		
application	$apply(DB, \Delta)$	`apply` δ;
hypothetical		ϵ `when` δ
combination		
smash	Δ_1 ! Δ_2	δ_1 `smash` δ_2
merge	Δ_1 & Δ_2	δ_1 `merge` δ_2

Figure 3: Some of the Core Algebraic and Language Operators of the H2O DBPL

In general, if $+n$ occurs in a delta of this form, it indicates that the function $f(x) = x + n$ is to be applied to the appropriate field of the specified object.

As with the use of structured attribute-granularity in connection with set-valued attributes, the use of arithmetic expressions increases the ability of deltas to preserve intended meaning. In this example, the deltas will commute with other deltas that add or subtract active and/or reserved lines (subject to the constraint that `slots_active + slots_reserved` $\leq$ `slots_total`). □

3 The Framework of the H2O DBPL

A prototype implementation of the H2O DBPL is currently being developed [BDD+95, DHDD95]. To provide a context for the discussion of the current paper, this section provides an overview of the syntax and semantics of the language (in Subsections 3.1 and 3.2, respectively). Subsection 3.3 formally defines the notion of "delta form", and lists several requirements and desires in connection with choosing delta forms. Subsection 3.4 illustrates the definition of a delta form by specifying an object-granularity delta form.

3.1 The syntax of the H2O DBPL

The H2O DBPL is defined by incorporating a set of algebraic operators for manipulating delta values and database states into an existing DBPL. In the current prototype we extend the standard ODMG [Cat93] languages. Figure 3 shows some of the core algebraic and language-based operators present in the H2O DBPL (see [DHDD95] for other H2O DBPL operators).

As shown in the figure, there are three basic groups of delta operators, for creating, accessing, and combining delta values. The language expression $[< \epsilon >]$, if evaluated in program state PS and database state[3]

[3] In this paper we distinguish between the program state, which holds non-persistent data, and the database state, which holds persistent data. Deltas range exclusively over the database state. This distinction is motivated by our interest in database applications; other variations are possible.

DB, yields $trace(\epsilon, PS, DB)$, in the form of a delta value which can be manipulated as a first-class object. The value of $trace(\epsilon, PS, DB)$ captures the sequence of changes to the objects in the database that would occur if the expression were actually executed. The resulting delta value is dependent on the types of the affected objects and the delta form being used (as explained in Section 4). Although not considered in this paper, the H2O DBPL also supports a *reverse deltafication* operator; this executes an expression ϵ against the database state, and returns a delta that would get back to the original database state.

The first way to access a delta value is to apply it (realize the change it represents to the database state). The second way is hypothetically, via the **when** operator in the language. The **when** operator has no equivalent in the algebra, essentially because the algebra does not support arbitrary expressions. The H2O DBPL also supports an operator **peek**, that permits explicit access to the internal value of a delta; this is not addressed here.

The **smash** and **merge** operators for combining deltas are discussed in the next two subsections.

3.2 The semantics of the H2O DBPL

A complete formal semantics of the H2O DBPL with object-granularity deltas is given in [DHR96]; see also [DHDD95]. We indicate here the elements of that semantics that are needed for the discussion of this paper. In particular, we introduce the valuation functions that specify the meaning of the language operators in terms of the underlying algebraic operators. As will be shown below, the choice of the algebraic operators is variable, allowing the same core language constructs to be used in connection with a variety of delta forms.

The semantics of some of the core H2O DBPL operators are given by the valuation functions in Figure 4. The expression valuation function, $\mathcal{E}[\![\cdot]\!]$, maps expressions from the *syntactic domain* of the H2O DBPL to the *semantic domain* of algebraic objects. States are viewed to be pairs (PS, DB) where PS denotes the *program state* and DB denotes the *database state*. Since the H2O DBPL is based on the ODMG languages that extend C++, we follow the convention that expressions can have side-effects on the state. Thus,

$$\mathcal{E} : Expression \rightarrow (ps \times db) \rightarrow (val \times (ps \times db))$$

Under $\mathcal{E}$ each expression yields a mapping from a state to a value (the result of the expression) and a new state. Similarly, statements are treated as expressions that yield the null value. We use $\mathcal{E}_{val}[\![\cdot]\!]$, $\mathcal{E}_{ps}[\![\cdot]\!]$, and $\mathcal{E}_{db}[\![\cdot]\!]$ to give the value, program state, and database state components of the output of $\mathcal{E}[\![\cdot]\!]$, respectively.

$$\mathcal{E}[\![\texttt{[<}\ \epsilon\ \texttt{>]}]\!](PS, DB) =$$
$$\quad \text{let } (V, (PS', DB')) = \mathcal{E}[\![\epsilon]\!](PS, DB) \text{ in}$$
$$\quad\quad (trace(\epsilon, PS, DB), (PS', DB))$$

$$\mathcal{E}[\![\texttt{apply}\ \delta\ \texttt{;}]\!](PS, DB) =$$
$$\quad \text{let } (\Delta, (PS', DB')) = \mathcal{E}[\![\delta]\!](PS, DB) \text{ in}$$
$$\quad\quad (\texttt{NULL}, (PS', apply(DB', \Delta)))$$

$$\mathcal{E}[\![\epsilon\ \texttt{when}\ \delta]\!](PS, DB) =$$
$$\quad \text{let } (\Delta, (PS', DB')) = \mathcal{E}[\![\delta]\!](PS, DB) \text{ in}$$
$$\quad\quad \text{let } (V, (PS'', DB'')) = \mathcal{E}[\![\epsilon]\!](PS', apply(DB', \Delta)) \text{ in}$$
$$\quad\quad\quad (V, (PS'', DB'))$$

$$\mathcal{E}[\![\delta_1\ \texttt{smash}\ \delta_2]\!](PS, DB) =$$
$$\quad \text{let } (\Delta_1, (PS', DB')) = \mathcal{E}[\![\delta_1]\!](PS, DB) \text{ in}$$
$$\quad\quad \text{let } (\Delta_2, (PS'', DB'')) = \mathcal{E}[\![\delta_2]\!](PS', DB') \text{ in}$$
$$\quad\quad\quad (\Delta_1 ! \Delta_2, (PS'', DB''))$$

Figure 4: Semantics of some H2O DBPL operators

Consider now the equation in Figure 4 giving the semantics of deltafication. First, the value of evaluating **[<** ϵ **>]** is a delta value capturing the effect the expression ϵ would have on the database state if executed The program state PS will change, exactly as if ϵ had been executed. The database state DB remains unchanged, except possibly for the consumption of "new" OIDs. The consumption of OIDs does not materially affect a database state, and so it is ignored in this paper.

The semantics of **apply**, **when** and **smash** are also shown in Figure 4. (For **when**, the database state might change because the evaluation of a delta expression may consume OIDs, and because of other technical issues beyond the scope of this paper; see [DHR96]).

3.3 Delta forms, requirements, and desires

In this subsection, we give a formal definition for *delta forms*, which gives the basic structure for the framework for specifying deltas for different data types. We then state three Requirements that a delta form must satisfy, as well as several informal "desires" concerning properties of delta forms.

A *delta form* over a type τ is a five tuple

$$\mathcal{D} = (D, \texttt{[<}\ \cdot\ \texttt{>]}^{\mathcal{D}}, apply^{\mathcal{D}}, smash^{\mathcal{D}}, merge^{\mathcal{D}})$$

where

- D is a family of syntactic objects which includes *fail*. (This is the set of *delta values* of the form.)
- $\texttt{[<}\ \cdot\ \texttt{>]}^{\mathcal{D}} : expressions \rightarrow (ps \times db) \rightarrow D$ maps expressions in the host language, in the context of a program state and database state, into deltas.

- $apply^{\mathcal{D}} : db \times D \rightarrow (db \cup error)$, such that for all states db, $apply^{\mathcal{D}}(db, fail) = error$. (This allows each element Δ of D to be viewed as a function from database states to database states.)

- $smash^{\mathcal{D}} : D \times D \rightarrow D$ such that $smash^{\mathcal{D}}(fail, \Delta) = smash^{\mathcal{D}}(\Delta, fail) = fail$ for each $\Delta \in D$.

- $merge^{\mathcal{D}} : D \times D \rightarrow D$ such that $merge^{\mathcal{D}}(fail, \Delta) = merge^{\mathcal{D}}(\Delta, fail) = fail$ for each $\Delta \in D$.

The above definition is focused on structure rather than implicit semantics. We now present some fundamental requirements placed on delta forms by the Heraclitus paradigm (see also [GHJ96]). The first focuses on the relationship between deltafication and apply:

Requirement 1 *For each expression ϵ,*

$$\mathcal{E}_{db}[\![\texttt{apply([< } \epsilon \texttt{ >])}]\!] \;=\; \mathcal{E}_{db}[\![\epsilon]\!].$$

Intuitively, this states that evaluating the delta expression [< ϵ >] to obtain a delta value Δ and then applying Δ has the same effect on the database state as simply executing ϵ. Importantly, this is weaker than insisting that for all pairs (PS, DB), (PS', DB') of states

$$apply(DB', \mathcal{E}_{val}[\![\texttt{[< } \epsilon \texttt{ >]}]\!](PS, DB)) = \mathcal{E}_{db}[\![\epsilon]\!](PS, DB')$$

which would require that the delta value [< ϵ >] preserve the meaning of the expression ϵ in all contexts, not only the context in which [< ϵ >] was computed. In order to support this stronger condition, the delta values would essentially have to be code-blocks, which would violate Desire 1 below.

The next requirement states that combining deltas using the smash (!) operator is, in a sense, equivalent to sequential application:

Requirement 2 *For each database state DB and pair Δ_1, Δ_2 of delta values,*

$$apply(DB, \Delta_1 ! \Delta_2) = apply(apply(DB, \Delta_1), \Delta_2).$$

The merge operator is intented to be used to combine pairs of "non-conflicting" or "non-interfering" deltas. In general, a variety of application-dependent semantic issues might be involved in choosing a merge operator for a given context. Here, we focus on defining a natural and simple merge operator defined by the following requirement:

Requirement 3

$$\Delta_1 \;\&\; \Delta_2 \;=\; \begin{cases} \Delta_1 ! \Delta_2 & \text{if } \Delta_1 ! \Delta_2 \;=\; \Delta_2 ! \Delta_1 \\ \text{fail} & \text{otherwise} \end{cases}.$$

We now informally present three "desires" which describe desirable, but not required, properties for delta forms. These desires can be used to design and select "good" delta forms. The first desire states that deltas should permit relatively efficient hypothetical access to proposed updates.

Desire 1 *Suppose that variable* `d` *holds a delta variable. Then the expense of evaluating an expression ϵ* `when d` *in a state DB should be comparable to the expense of evaluating ϵ in DB.*

The next desire expresses the intuition that because our deltas are primarily value-based, distinct deltas should correspond to different mappings from state to state.

Desire 2 *if $\forall$ database states DB $apply(DB, \Delta_1) = apply(DB, \Delta_2)$, then $\Delta_1 = \Delta_2$.*

The third desire attempts to enforce a strong semantic correspondence between an expression ϵ and the delta associated with [< ϵ >].

Desire 3 *If ϵ is a command that does not involve any reads to the database, then the value of* [< ϵ >] *is independent of the database state.*

3.4 Delta forms for object-granularity

To illustrate the notion of delta form, we now define the delta form for object-granularity deltas.

Let C be a class associated with a type τ in some OODB schema. We will define a delta form

$$\mathcal{D}^{og} \;=\; (D^{og}, [< >]^{og}, apply^{og}, smash^{og}, merge^{og})$$

to support the object-granularity semantics over C. (This can be generalized to create deltas over the full OODB schema, by viewing the schema as a tuple of classes)

Each element Δ of D^{og} is either *fail* or it is a possibly empty finite set containing: *modification atoms* of the form *mod* $o : v$ where $v \in dom(\sigma)$; *insertion atoms* of the form *ins* $o : v$ where $v \in dom(\sigma)$; and *deletion atoms* of the form *del* o Object-granularity deltas are subject to the following consistency condition: an OID o cannot appear in two distinct atoms.

The value of the deltafication operator [< ϵ >]^{og} in state (PS, DB) is defined as $trace^{og}(\epsilon, PS, DB)$. An informal description of this trace function is as follows: ϵ is executed in hypothetical mode. For each OID o of type τ mentioned by ϵ, atomic commands to insert, delete, or modify the value of o are recorded. The net effect of ϵ on o is incorporated into the final delta value.

The semantics of the algebraic apply and smash operators for the object-granularity form of deltas are given in tabular form in Figure 5. The information in

presence of o in extent of C in DB	occurrence of o in Δ: absent	$mod(o : v_2)$	$ins(o : v_2)$	$del(o)$
absent	absent	absent	$(o : v_2)$	absent
$(o : v_1)$	$(o : v_1)$	$(o : v_2)$	$(o : v_2)$	absent

(a) Semantics of *apply*

occurrence of o in Δ_1	occurrence of o in Δ_2: absent	$mod(o : v_2)$	$ins(o : v_2)$	$del(o)$
absent	absent	$mod(o : v_2)$	$ins(o : v_2)$	$del(o)$
$mod(o : v_1)$	$mod(o : v_1)$	$mod(o : v_2)$	$ins(o : v_2)$	$del(o)$
$ins(o : v_1)$	$ins(o : v_1)$	$ins(o : v_2)$	$ins(o : v_2)$	$del(o)$
$del(o)$	$del(o)$	$del(o)$	$ins(o : v_2)$	$del(o)$

(b) Semantics of smash ('!')

Figure 5: Semantics of apply and smash for object-granularity deltas

the three tables is essentially "pointwise", describing what to do for each individual object or pair of objects in the operands. This is possible because any particular OID can be referenced by at most one element in a valid delta.

Consider Figure 5(a), which defines $apply^{og}(DB, \Delta)$. A particular OID o is either absent from the extent of class C in state DB, or present with value v_1. In the former case, o is present in the resulting state $apply(DB, \Delta)$ only if $ins(o : v_2) \in \Delta$ for some v_2. The case where $(o : v_1)$ is present and $mod(o : v_2) \in \Delta$ is self-explanatory. Suppose now that $(o : v_1)$ is in the extent of C in DB and $ins(o : v_2) \in \Delta$. Intuitively, in this case we view the *ins* operator as having two components: first insert the object if necessary, and then modify its value.

The smash operator $(\Delta_1 ! \Delta_2)$ for object-granularity deltas is given in Figure 5(b). Smash is computed by combining the elements of each delta which refer to the same OID. If a particular OID is referenced in only one delta, then the element from that delta appears in the result. If the same OID is referenced by elements in both deltas, the element which will appear in the result is computed according to the table, which essential gives precedence to the second delta in cases where there are conflicting operations.

The merge operator $(\Delta_1 \& \Delta_2)$ for object-granularity deltas is defined using a table analogous to Figure 5(b), and satisfies Requirement 3 (see [DHR96]).

It is easy to verify that this delta form satisfies the three requirements of Subsection 3.3. It is also relatively straightforward to verify that it satisfies Desires 2 and 3. Current experiments with the prototype H2O DBPL indicate that Desire 1 is also satisfied by this delta form.

4 Deltas for Complex Value Types

This section presents a systematic and recursive framework for specifying forms of deltas over complex value types, which mirrors the internal, nested structure of those types. These complex value types can be defined using the usual type constructors, *tuple*, *set*, *bag*, and *list*, and two additional constructors, *o-set* and *dictionary*. Due to space limitations, *set* and *list* are omitted from this paper; see [DHR96] for the complete specification including these constructors.

In our framework, there are two dimensions involved in the specification of a delta form for a given complex value type. The first involves the depth into the internal structure of the type at which the delta value is computed. The second involves the family of functions that can be used in the delta value to specify how values are modified (as illustrated by Example 2.1).

For each type, including the base types, we permit the user to specify a family (or several families) of functions that can be used on values of that type. A *proper family of functions* (PFF) over type τ is a family $\mathcal{F}$ of functions from $dom(\tau)$ to $dom(\tau)$ that is closed under composition and includes all constant functions. We write $f \circ g$ to denote the composition of applying f followed by applying g. For any type τ, the set of all constant functions over τ is a PFF over τ. As another example, a PFF over type `integer` is all constant functions *and* all functions of the form $f(x) = c_1 \times x + c_2$, where c_1, c_2 are integers. In some delta forms, a PFF might be defined over a non-leaf subtype of a type.

Subsection 4.1 describes the individual constructors used in our complex value types. Subsection 4.2 describes, for each constructor a canonical "template" for constructing delta forms. Subsection 4.3 describes how the templates can be combined to create a variety of nested delta forms for arbitrary complex value types.

Theoretical results concerning these delta forms are also presented.

4.1 A family of complex value types

To illustrate our framework for deltas we will use a family of complex value types that captures many of the data structures that arise frequently in OODBs. In general, we assume that an OODB schema (i.e., class hierarchy) is defined, where the type associated with each class is a complex value type. (See, e.g., [AHV95] for a discussion of how complex values are incorporated into OODB schemas.)

A *complex value type* (which we will generally refer to simply as *type*) is a tree $\tau = (V, E, \lambda)$ where V is the set of nodes, E the set of edges, and λ maps the nodes in V to base types or type constructors. Each leaf node is mapped by λ to a *base type* (e.g., integer, float or string) or to a class in the OODB schema. Each non-leaf node is mapped by λ to a complex-value type *constructor*, which may be one of {tuple, set, bag, list, o-set, dict[b]}, where b ranges over base types. All nodes mapped to set, bag, list, o-set, or dict[b] have exactly one child. For a type τ, $dom(\tau)$ denotes the *domain* of τ, i.e., the set of values having type τ. If τ is a type and n a node of τ, then $subtype(\tau, n)$ denotes the subtype of τ with root n.

Each base type is considered to be a complex-value type. We now consider four of the constructors for complex-value types, and then indicate how OODB schemas are represented in our framework.

tuple: A type with tuple as the root type has the form $\tau = tuple(a_1 : \sigma_1, \ldots, a_n : \sigma_n)$ where σ_i is a complex value type for each i. Instances of this type have the form $\langle a_1 : val_1, \ldots, a_n : val_n \rangle$. The operators supported for tuple types include the ability to change the value of a coordinate by applying a function f to it (which might be a constant function).

bag: Instances of types with form $\tau = bag(\sigma)$ have the form $\{\!\{v_1 : cnt_1, \ldots, v_n : cnt_n\}\!\}$ where $n \geq 0$, each v_i is a distinct element of $dom(\sigma)$, and each cnt_i is a positive integer. This bag has cnt_i copies of value v_i in it. We write $cnt(T, v_i) = cnt_i$ for each i, and $cnt(T, v) = 0$ for each v such that $v \notin \{v_1, \ldots, v_n\}$. Supported operations on bags are to add or delete elements. Deleting an element which is not present in a bag is a no-op.

o-set: We use an "o-set" type to model the structure of OODB classes. We assume an infinite set of *object identifiers* (OIDs), and the ability to consume distinct, unused OIDs on demand. If an OODB class C has associated type σ, then the extent of C is an o-set of type $o\text{-}set(\sigma)$.

An instance of type $\tau = o\text{-}set(\sigma)$ is a set S of pairs having the form $(o : v)$ where o is a distinct OID and $v \in dom(\sigma)$. The operators supported on o-sets of type $o\text{-}set(\sigma)$ are (a) to insert a new element $(o : v)$ where o is a newly consumed OID, (b) to delete an element (indicated simply by naming the OID to be deleted), or (c) to modify the value of an element (o, v) by applying some function f, i.e., to replace (o, v) by $(o, f(v))$ for some function $f : \sigma \to \sigma$. Deletion of o for some o not in the o-set is a no-op, as is modification of (o, v) for o not in the o-set.

dictionary: A *dictionary* (also called "map" or "associative array") is a binary relation whose first coordinate has some base type and is a key. Instances of types with form $\tau = dict[b](\sigma)$ are sets of pairs $(s : v)$ where $s \in dom(b)$ and $v \in dom(\sigma)$. The dictionary can be modified by inserting or deleting pairs (subject to the functional dependency), or by modifying the second coordinate of a pair. (An o-set can be viewed as a special kind of dictionary, where the elements of the first coordinate are system generated.)

representing OODB schemas: Suppose now that S is an OODB schema, consisting of a set of classes $C_1, \ldots, C_n$, each with associated types $T_1, \ldots, T_n$. (For the discussion here we ignore subtyping and inheritance relationships between the classes.) An instance I of S is defined to be a mapping that associates to each class C_i an o-set of type $o\text{-}set(T_i)$. Thus, we view the topmost structure of instances of S to be of the form $tuple(C_1 : o\text{-}set(T_1), \ldots, C_n : o\text{-}set(T_n))$.

4.2 Templates for delta forms

This subsection specifies delta form "templates" for the four complex value type constructors discussed here. (Due to space limitations, some of these specifications are incomplete, see [DHR96] for complete specifications, as well as the templates for sets and lists). These templates are combined to build nested delta forms in the next subsection.

A *template* for type constructor γ is a five-tuple

$$\mathcal{C} = (C, [\!< >\!]^{\mathcal{C}}, apply^{\mathcal{C}}, smash^{\mathcal{C}}, merge^{\mathcal{C}})$$

If γ is one of o-set, list, or dict[b], then such a template takes as input a type σ and a PFF $\mathcal{F}$ over σ. The output, denoted $\mathcal{C}(\sigma, \mathcal{F})$, is a delta form $\mathcal{D}$ for type $\gamma(\sigma)$. For tuple, such a template takes as input a vector $(\sigma_1, \ldots, \sigma_n)$ of types and vector $(\mathcal{F}_1, \ldots, \mathcal{F}_n)$ of PFFs for these types. The output, denoted $\mathcal{C}((\sigma_1, \ldots, \sigma_n), (\mathcal{F}_1, \ldots, \mathcal{F}_n))$, is a delta form for type $tuple(a_1 : \sigma_1, \ldots, a_n : \sigma_n)$. Finally, if γ is set or bag, such a template takes as input a type σ. The output, denoted $\mathcal{C}(\sigma)$ is a delta form for type $\gamma(\sigma)$.

For each constructor γ under consideration we now describe its *canonical template*, denoted as $\mathcal{C}_\gamma$.

o-set: Let $\tau = o\text{-}set(\sigma)$ for some σ, and let $\mathcal{F}$ be a PFF over σ. In $\mathcal{C}_{o\text{-}set}(\sigma, \mathcal{F})$, a delta value over τ is either

		occurrence of o in Δ			
		absent	$mod(o:f)$	$ins(o:v_2)$	$del(o)$
presence of o in instance T of τ	absent	absent	absent	$(o:v_2)$	absent
	$(o:v_1)$	$(o:v_1)$	$(o:f(v_1))$	$(o:v_2)$	absent

(a) Semantics of *apply*

		occurrence of o in Δ_2			
		absent	$mod(o:g)$	$ins(o:v_2)$	$del(o)$
occurrence of o in Δ_1	absent	absent	$mod(o:g)$	$ins(o:v_2)$	$del(o)$
	$mod(o:f)$	$mod(o:f)$	$mod(o:f \circ g)$	$ins(o:v_2)$	$del(o)$
	$ins(o:v_1)$	$ins(o:v_1)$	$ins(o:g(v_1))$	$ins(o:v_2)$	$del(o)$
	$del(o)$	$del(o)$	$del(o)$	$ins(o:v_2)$	$del(o)$

(b) Semantics of smash ('!')

Figure 6: Semantics of apply and smash for canonical template for o-sets

fail or a finite set containing: *modification atoms* of the form $mod\ o : f$ where $f \in \mathcal{F}$; *insertion atoms* of the form $ins\ o : v$ where $v \in dom(\sigma)$; and *deletion atoms* of the form $del\ o$, subject to the following consistency condition: an OID o cannot appear in two distinct atoms.

In order to compute the delta associated with $[< \epsilon >]$ on an o-set T of type $\tau = o\text{-}set(\sigma)$, ϵ is executed in hypothetical mode. For each OID o of type τ, ϵ can be viewed as a sequence of atomic operations, $\epsilon_1, \ldots, \epsilon_n$. Let $c_1, \ldots, c_n$ be the sequence of values of the object with OID o, where c_i is the value after the (hypothetical) execution of ϵ_i. We can than compute a sequence of atomic deltas, $\alpha_1, \ldots, \alpha_n$, as follows:

if ϵ_i is an insertion, then $\alpha_i = ins\ o : c_i$
else if ϵ_i is a deletion, then $\alpha_i = del\ o$
else $\alpha_i = mod\ o : f_i$, where
  if ϵ_i is expressible as some $f \in \mathcal{F}$, then $f_i = f$
  otherwise $f_i =$ the constant function c_i.

The single atom which will appear in the resulting delta for OID o is computed by smashing the atomic deltas $\alpha_1 \ldots \alpha_n$ in sequence. For *modification* atoms, this corresponds to taking the composition of the functions f_i. If any f_i is a constant function, then the resulting function will also be a constant function.

The semantics for apply and smash for delta form $\mathcal{C}_{o\text{-}set}(\sigma, \mathcal{F})$ are shown in Figure 6. In that figure, symbols 'f' and 'g' range over elements of $\mathcal{F}$. The semantics for merge can be derived from the semantics of smash and Requirement 3.

Note that the operators associated with object-granularity deltas as described in Section 2 (see Figure 5) correspond to a special case of the algebraic operators for o-sets, in which the family $\mathcal{F}$ is chosen to be the family of constant functions.

dictionary: Let $\tau = dict[b](\sigma)$ and let $\mathcal{F}$ be a PFF over σ. In $\mathcal{C}_{dict[b]}(\sigma, \mathcal{F})$, a delta over τ is either *fail* or a set of *modification atoms*, *insertion atoms*, and *deletion atoms*, analogous to deltas for o-sets. The semantics for apply, smash, merge and deltafication are essentially identical to the semantics for o-sets, except that the variable o ranging over OIDs is replaced by a variable s ranging over b.

As an aside, we note that the semantics of delta operators for dictionaries can also be used to provide a delta form for conventional relations constrained by one key dependency.

bag: Let $\tau = bag(\sigma)$. In $\mathcal{C}_{bag}(\sigma)$, a delta is either the special value *fail* or a set of pairs

$$\Delta = \{\langle del(v_1) : cnt_1^d,\ ins(v_1) : cnt_1^i\rangle, \ldots, \langle del(v_k) : cnt_k^d,\ ins(v_k) : cnt_k^i\rangle\}$$

where each v_j is distinct, each cnt_j^d and cnt_j^i is a non-negative integer, and $cnt_j^d + cnt_j^i > 0$ for each j.

When applying Δ to value T, for each v_j we first perform $min\{cnt_j^d, cnt(T, v_j)\}$ deletions of v_j, followed by cnt_j^i insertions of v_j.

The semantics for smash for bag deltas is given in Figure 7. The semantics for merge has been omitted for space considerations, it is given in [DHR96].

In order to compute the delta for $[< \epsilon >]$ on bag T for delta from $\mathcal{C}_{bag}(\sigma)$, ϵ is executed in hypothetical mode and for each v of type σ, each atomic command to insert or delete v into T is recorded. This yields, for each v, a sequence of commands $c_1; c_2; \ldots; c_k;$. This sequence is rewritten using the rule $ins(v); del(v) \mapsto$ empty-string until there are no consecutive pairs of this form. This yields a sequence $((del(v);)^n\ (ins(v);)^m)$ with $n, m \geq 0$. If $n+m \neq 0$ then $\langle del(v) : n, ins(v) : m\rangle$ is included in the delta, otherwise no pair for v is included in the delta.

		occurrence of v in Δ_2: absent	occurrence of v in Δ_2: $\langle del(v):d_2, ins(v):i_2\rangle$
occurrence of v in Δ_1	absent	absent	$\langle del(v):d_2, ins(v):i_2\rangle$
	$\langle del(v):d_1, ins(v):i_1\rangle$	$\langle del(v):d_1, ins(v):i_1\rangle$	$i_1 > d_2 : \langle del(v):d_1, ins(v):i_2 + (i_1 - d_2)\rangle$ $i_1 = d_2 : \langle del(v):d_1, ins(v):i_2\rangle$ $i_1 < d_2 : \langle del(v):d_1 + (d_2 - i_1), ins(v):i_2\rangle$

(b) Semantics of smash ('!')

Figure 7: Semantics of smash for canonical template for bags

tuple: Let $\tau = tuple(a_1 : \sigma_1, \ldots, a_n : \sigma_n)$. Let $\mathcal{F}_i$ be a PFF over σ_i for $i \in [1, n]$. Deltas for $\mathcal{C}_{tuple}((\sigma_1, \ldots, \sigma_n), (\mathcal{F}_1, \ldots, \mathcal{F}_n))$ have the form

$$\Delta = \{a_{i_1} : f_{i_1}, \ldots, a_{i_k} : f_{i_k}\}$$

where $i_1, \ldots, i_k$ is a subsequence of $1, \ldots, n$ and $f_{i_j} \in \mathcal{F}_{i_j}$ for each j. The semantics for apply, smash, and merge are defined in the natural manner.

4.3 Nested delta forms

This subsection describes how the templates for individual constructors can be combined to create nested delta forms. Also, several theoretical results are presented that demonstrate the soundness of our framework.

Let τ be a complex value type. A *frontier* of τ is a set F of nodes such that each path from root to leaf contains exactly one node in F. A frontier is *proper* if for each node n in F, each node in the path from the root to n, except possibly for n itself, has one of the following labels: o-set, tuple, dict[b], or list (Sets and bags are not permitted as internal nodes of the path, because the canonical templates for these do not permit modifications to individual elements of sets or bags.)

An *annotated type* is a triple

$$\mathcal{A} = (\tau, F, \{\mathcal{F}^n \mid n \in F'\})$$

where F is a proper frontier of type τ, and F' is the set of nodes in F not labeled by set or bag, and for each node $n \in F'$, $\mathcal{F}^n$ is a PFF for the type $subtype(\tau, n)$.

We now give a recursive definition of the *canonical delta form*

$$\mathcal{D}[\mathcal{A}] = (D[\mathcal{A}], [< >]^{\mathcal{A}}, apply^{\mathcal{A}}, smash^{\mathcal{A}}, merge^{\mathcal{A}})$$

of an annotated type $\mathcal{A} = (\tau, F, \{\mathcal{F}^n \mid n \in F'\})$

Base cases: Suppose that $F = \{root(\tau)\}$. There are three subcases:
$\tau = set(\sigma)$: In this case $\mathcal{D}[\mathcal{A}] = \mathcal{C}_{set}(\sigma)$.
$\tau = bag(\sigma)$: In this case $\mathcal{D}[\mathcal{A}] = \mathcal{C}_{bag}(\sigma)$.
Otherwise: In this case a PFF $\mathcal{F}$ is specified for the root r of τ, and we set

$$\mathcal{D}[\mathcal{A}] = (\mathcal{F}, [< >]^{\mathcal{A}}, apply^{\mathcal{A}}, smash^{\mathcal{A}}, merge^{\mathcal{A}})$$

where in the first coordinate $\mathcal{F}$ is viewed as a set of syntactic objects, $apply^{\mathcal{A}}(v, f) = f(v)$, $smash^{\mathcal{A}}(f, g) = f \circ g$, and $merge^{\mathcal{A}}(f, g) = f \circ g$ if $f \circ g = g \circ f$ and is *fail* otherwise. Finally, $[< \epsilon >]^{\mathcal{A}}$ is defined by taking the trace of ϵ hypothetically executed in a state (PS, DB), and expressing the net effect as a member of $\mathcal{F}$, in a manner analogous to the semantics for $[< \epsilon >]$ in the canonical template for o-sets.

We now consider the cases where $F \neq \{root(\tau)\}$.

o-set, dictionary, list: Suppose $\tau = o\text{-}set(\sigma)$. Let F' be the set of nodes in F that are not labeled by set or bag. Then $\mathcal{D}[\mathcal{A}] = \mathcal{C}_{o\text{-}set}(\sigma, D[(\sigma, F, \{\mathcal{F}^n \mid n \in F'\})])$. The cases of dictionary and list are analogous.

tuple: Suppose $\tau = tuple(a_1 : \sigma_1, \ldots, a_m : \sigma_m)$. Let F_i be the portion of F that is below $(a_i : \sigma_i)$ in τ, and F_i' the set of nodes in F_i that are not labeled by set or bag. Then

$$\begin{aligned}\mathcal{D}[\mathcal{A}] = \mathcal{C}_{tuple}((\sigma_1, \ldots, \sigma_m),\\ (D[(\sigma_1, F_1, \{\mathcal{F}^n \mid n \in F_1'\})], \ldots,\\ D[(\sigma_m, F_m, \{\mathcal{F}^n \mid n \in F_m'\})]))\end{aligned}$$

We now present four results concerning canonical delta forms (see [DHR96] for detailed proofs). The first result ensures the validity of the recursive construction just given.

Proposition 4.1: Suppose that $\tau = o\text{-}set(\sigma)$, and let $\mathcal{F}$ be a PFF over σ. Let n be the root of σ. Let $\mathcal{A} = (\tau, \{n\}, \{\mathcal{F}^n\})$ and $\mathcal{D}[\mathcal{A}] = (D[\mathcal{A}], [< >]^{\mathcal{A}}, \ldots)$. Let $\mathcal{F}'$ be the set $D[\mathcal{A}]$, where each $\Delta \in D[\mathcal{A}]$ is viewed as a function over τ according to the semantics specified by $apply^{\mathcal{A}}$ Then $\mathcal{F}'$ is a PFF over τ. The analogous results hold for types with root set, bag, list, tuple, and dict[b].

Proof: This follows because the smash operator is defined for the canonical template for each constructor, and satisfies Requirement 2. □

We now state:

Theorem 4.2: If $\mathcal{D}$ is a canonical nested delta form, then $\mathcal{D}$ satisfies Requirements 1, 2, and 3.

The remaining results concern the ability of different delta forms to retain their intended meaning as the underlying database state changes, and relate this to structural characteristics.

Definition: Delta form $\mathcal{D}_1$ *semantically dominates* delta form $\mathcal{D}_2$, denoted $\mathcal{D}_1 \geq \mathcal{D}_2$, if for each expression ϵ, program state PS, and pair DB_a, DB_b of database states,

$$apply(DB_b, \mathcal{E}_{val}^{\mathcal{D}_2}[\![[< \epsilon >]]\!](PS, DB_a)) = \mathcal{E}_{db}[\![\epsilon]\!](PS, DB_b)$$

implies

$$apply(DB_b, \mathcal{E}_{val}^{\mathcal{D}_1}[\![[< \epsilon >]]\!](PS, DB_a)) = \mathcal{E}_{db}[\![\epsilon]\!](PS, DB_b)$$

Intuitively, if $\mathcal{D}_1 \geq \mathcal{D}_2$, then deltas from $\mathcal{D}_1$ preserve their intended meaning whenever deltas from $\mathcal{D}_2$ do.

Definition: Let τ be a type, and let $\mathcal{D}_i = \mathcal{D}[(\tau, F_i, \{\mathcal{F}_i^n \mid n \in F_i'\})]$ be a nested delta form for $i \in [1,2]$. Then $\mathcal{D}_1$ *frontier dominates* $\mathcal{D}_2$ if F_1 is "below" F_2, in the sense that no node of F_2 is a descendant of a node of F_1 in τ.

The following presents a characterization of semantic dominance, for a restricted family of delta forms.

Theorem 4.3: Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be canonical nested delta forms over type τ. Suppose that for each non-set, non-bag frontier node n, the PFF used is the family of all constant functions over $subtype(\tau, n)$. Then $\mathcal{D}_1 \geq \mathcal{D}_2$ if and only if $\mathcal{D}_1$ frontier dominates $\mathcal{D}_2$

This result can be strengthened by permitting arbitrary PFFs at leaf nodes of τ, under the assumption that if the frontiers of both annotated types include a leaf node n, then the PFF at n of $\mathcal{D}_1$ contains the PFF at n of $\mathcal{D}_2$. The result cannot be extended to permit arbitrary PFFs at frontier nodes. The next result gives a sufficient condition for semantic dominance, when arbitrary PFFs are used.

Theorem 4.4: Let $\mathcal{D}_i = \mathcal{D}[(\tau, F_i, \{\mathcal{F}_i^n \mid n \in F_i'\})]$ be a canonical nested delta form for $i \in [1,2]$. Then $\mathcal{D}_1 \geq \mathcal{D}_2$ if (a) $\mathcal{D}_1$ frontier dominates $\mathcal{D}_2$, and (b) for each node $p \in F_2$ the following holds: If $\sigma = subtype(\tau, p)$ and F_1^p is the set of nodes of F_1 occurring in σ, then the family $\mathcal{F}_2^p$ is contained in $D[(\sigma, F_1^p, \{\mathcal{F}_1^q \mid q \in F_1^{p\prime}\})]$.

Acknowledgements

We are grateful to Dean Jacobs, Omar Boucelma, Jean-Claude Franchitti, Roger King, and Gang Zhou, for many interesting discussions on Heraclitus for object-oriented databases and related topics. Also, we are very thankful to Marcia Derr and Jacques Durand for numerous discussions concerning the Heraclitus paradigm and practical examples of its usefulness.

References

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, Reading, MA, 1995.

[BDD+95] O. Boucelma, J. Dalrymple, M. Doherty, J. C. Franchitti, R. Hull, R. King, and G. Zhou. Incorporating Active and Multi-database-state Services into an OSA-Compliant Interoperability Framework. In *The Collected Arcadia Papers, Second Edition.* University of California, Irvine, May 1995.

[Cat93] R.G.G. Cattell. *The Object Database Standard: ODMG-93.* Morgan Kaufmann Publishers, San Mateo, California, 1993.

[DDD+96] M. Derr, J. Durand, M. Doherty, R. Hull, and M. Rupawalla. Applications of Heraclitus in telecommunications information processing. Technical report, University of Colorado, Boulder, 1996.

[DH95] M. Doherty and R. Hull. Towards a framework for efficient management of potentially conflicting database updates. In *Proc. IFIP WG2.6 Sixth Working Conference on Database Semantics (DS-6)*, 1995. to appear.

[DHDD95] M. Doherty, R. Hull, M. Derr, and J. Durand. On detecting conflict between proposed updates. In *Proc. of Intl. Workshop on Database Programming Languages*, September 1995. To appear.

[DHR96] M. Doherty, R. Hull, and M. Rupawalla. A framework for manipulating proposed updates in object-oriented databases, 1996. Technical report in preparation.

[GHJ+93] S. Ghandeharizadeh, R. Hull, D. Jacobs, et al. On implementing a language for specifying active database execution models. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 441–454, 1993.

[GHJ96] S. Ghandeharizadeh, R. Hull, and D. Jacobs. Heraclitus: Elevating deltas to be first-class citizens in a database programming language. *ACM Trans. on Database Systems*, 1996. To appear.

[HJ91] R. Hull and D. Jacobs. Language constructs for programming active databases. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 455–468, 1991.

[JH91] D. Jacobs and R. Hull. Database programming with delayed updates. In *Intl. Workshop on Database Programming Languages*, pages 416–428, San Mateo, Calif., 1991. Morgan-Kaufmann, Inc.