IceDust 2

Derived Bidirectional Relations and Calculation Strategy Composition

Harkes, Daco; Visser, Eelco

DOI
10.4230/LIPIcs.ECOOP.2017.14

Publication date
2017

Document Version
Final published version

Published in
31st European Conference on Object-Oecoopriented Programming, ECOOP 2017

IceDust 2: Derived Bidirectional Relations and Calculation Strategy Composition*

Daco C. Harkes[1] and Eelco Visser[2]

1 **Delft University of Technology, Delft, The Netherlands**
`d.c.harkes@tudelft.nl`
2 **Delft University of Technology, Delft, The Netherlands**
`e.visser@tudelft.nl`

Abstract

Derived values are values calculated from base values. They can be expressed with views in relational databases, or with expressions in incremental or reactive programming. However, relational views do not provide multiplicity bounds, and incremental and reactive programming require significant boilerplate code in order to encode bidirectional derived values. Moreover, the composition of various strategies for calculating derived values is either disallowed, or not checked for producing derived values which will be consistent with the derived values they depend upon.

In this paper we present IceDust2, an extension of the declarative data modeling language IceDust with derived bidirectional relations with multiplicity bounds and support for statically checked composition of calculation strategies. Derived bidirectional relations, multiplicity bounds, and calculation strategies all influence runtime behavior of changes to data, leading to hundreds of possible behavior definitions. IceDust2 uses a product-line based code generator to avoid explicitly defining all possible combinations, making it easier to reason about correctness. The type system allows only sound composition of strategies and guarantees multiplicity bounds. Finally, our case studies validate the usability of IceDust2 in applications.

1998 ACM Subject Classification D.3.2 Data-flow languages

Keywords and phrases Incremental Computing, Data Modeling, Domain Specific Language

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.14

Supplementary Material ECOOP Artifact Evaluation approved artifact available at `http://dx.doi.org/10.4230/DARTS.3.2.1`

1 Introduction

Derived values are values computed from base values. Base values are provided by the users of an application. When base values change, derived values should change accordingly. A key concern in *implementing* systems with derived values is minimizing the *computational* effort that is spent to re-compute derived values after updates to base values. A key concern in *modeling* systems with derived values is minimizing the *programming* effort to realize such minimal computations. Ideally, one *declaratively* specifies how values are derived from base values; from such a specification an efficient update strategy is generated automatically. Declarative programming with derived values is an old idea, going back at least to incremental computation of views in relational databases [12]. More recently it has seen much attention in new fields. Incremental programming [13, 14, 15, 24, 31] uses previously calculated values

* This research was funded by the NWO VICI *Language Designer's Workbench* project (639.023.206).

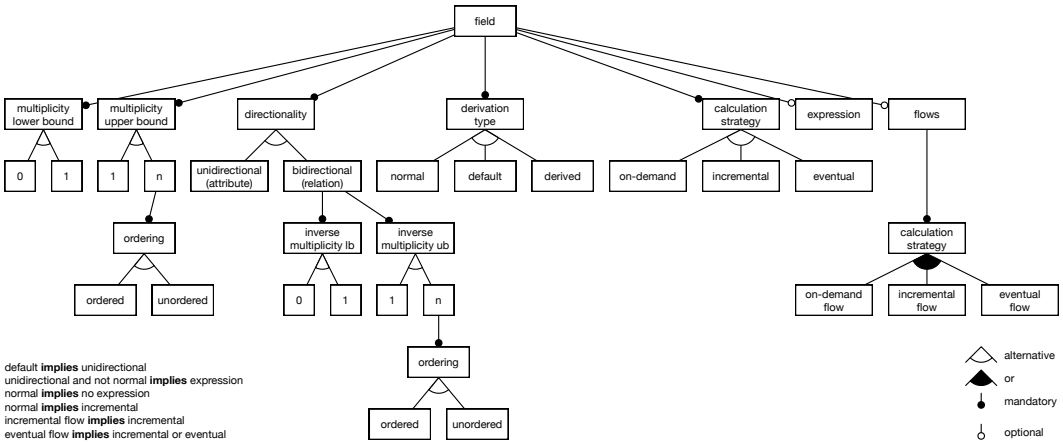

Figure 1 Feature model for configuration of a field in IceDust and IceDust2.

to efficiently compute new ones. In (functional) reactive programming [7, 22, 23, 28] base values are modeled as time-varying signals, and derived values are modeled as signals that are automatically updated when the values of dependent signals change.

These techniques vary in expressiveness and in static guarantees for consistency. Derived bidirectional relations can be expressed directly in the relational paradigm, but the relational paradigm provides no guarantees on multiplicity bounds for derived values. On the other hand, multiplicity bounds can be directly expressed with `Option` and `Collection` types in incremental and reactive programming, but only unidirectional relations can be expressed without encoding. Moreover, the composition of strategies for calculating derived values is either disallowed [15], or composition is not statically checked to guarantee that derived values will be consistent with the values they depend upon [23, 28]. For example, the (accidental) dependency of incremental computations on on-demand computations can lead to inconsistencies in incrementally computed values.

The IceDust data modeling language [15] supports declarative specification of derived value attributes through separation of concerns. An IceDust data model definition consists of *entities* with *attributes* and *bidirectional relations* between entities. *Fields* of entities comprise attributes and the ends of bidirectional relations. IceDust fields vary independently in *multiplicity* lower-bound and upper-bound, *directionality* (unidirectional or bidirectional), *derivation type* (user value, default value, or calculated value), and *calculation strategy*. A bidirectional field also defines a multiplicity bound for its inverse. This variability is captured by the feature model[1] in Figure 1. IceDust is a configuration language for this feature model. Each field in a data model is a selection of features complying with this feature model. However, the language does not support full orthogonality of feature selection. First, the choice of calculation strategy is global, i.e. the chosen calculation strategy applies to all fields in a data model; choosing different strategies for different fields is not supported. Second, only attribute values can be derived; derivation of relation values is not supported.

In this paper we present IceDust2, an extension of IceDust with fully orthogonal configuration selection supporting the following features:

[1] A feature model is a compact representation of all the products of a software product line (SPL)[18]. A product configuration is determined by a selection of features satisfying the constraints of the feature model.

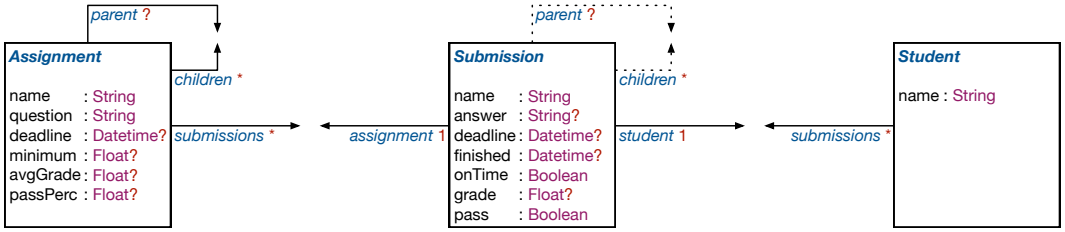

Figure 2 Running example class diagram. Bidirectional relations are denoted by →←, and dotted lines express derived relations.

- In addition to derived value attributes, IceDust2 supports derived bidirectional relations. Derived relations are computed incrementally or eventually, which requires incremental maintenance of bidirectional relations.
- Derived relations have multiplicity bounds. The type system statically checks that derived relation computations are guaranteed to satisfy these bounds.
- While IceDust only supports *global selection* of calculation strategies, IceDust2 supports *local selection* or *composition* of calculation strategies, which allows tuning the re-calculation behavior of individual fields.
- Not all combinations of strategies yield consistent re-calculation of derived values. The IceDust2 type system checks that selected strategy compositions are sound.
- While the *selection* of features in a data model specification is orthogonal, each combination of features requires a *specialized implementation* in order to produce consistent results. We address the combinatorial explosion of specializations using a product-line approach to reduce the size of the compiler and make reasoning about its correctness feasible.

The paper is structured as follows. In the next section we examine IceDust and its limitations and introduce IceDust2 for specifying derived bidirectional relations with multiplicity bounds and composition of calculation strategies. In Section 3 we analyze the run-time interaction between derived values, bidirectional relations, multiplicity bounds, and various calculation strategies. In Section 4 we define the operational semantics covering all possible feature combinations. In Section 5 we describe the type system guaranteeing sound composition of calculation strategies. In Section 6 we discuss two implementations of IceDust2. In Section 7 we evaluate the expressiveness of the language with case studies. In Section 8 we analyze the limitations entailed by static multiplicity checks on derived relations. In Section 9 we compare IceDust2 to other approaches to declarative data modeling.

2 Declarative Data Modeling by Feature Selection

In this section we summarize the features of the IceDust data modeling language, analyze its variability limitations, and introduce IceDust2, an extension of IceDust with orthogonal feature selection.

2.1 Running Example.

To illustrate data modeling in IceDust and IceDust2, we use a simplified learning management system as running example (Figures 2-4). `Assignments` are structured as a tree. For example, the `math` assignment consists of an `exam` and a `lab` (Figure 3 center). `Students` submit `Submissions` to these assignments. These submissions form trees as well, mirroring the

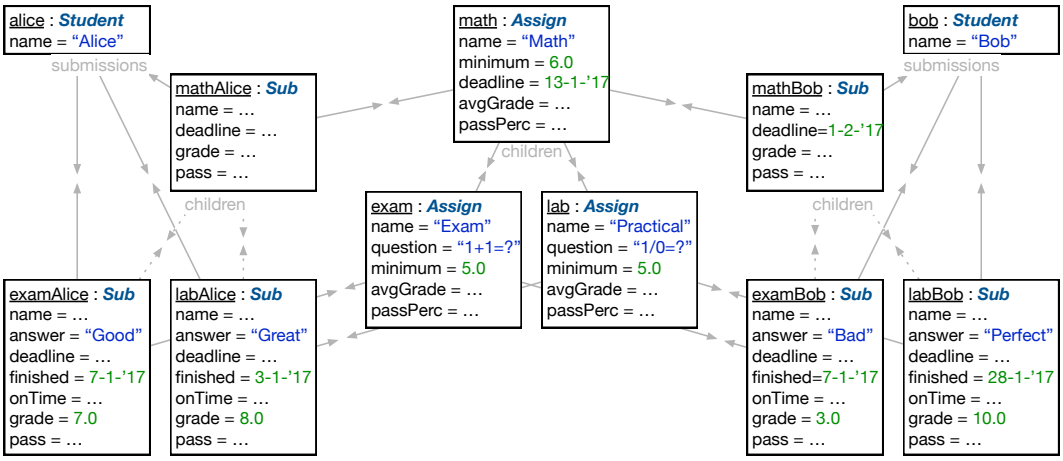

Figure 3 Running example data. References are denoted by →, bidirectional relation values are denoted by →←, derived references are dotted arrows, and derived attribute values are dots.

```
module example (incremental)
entity Assignment (eventual) {
  name      : String
  question  : String?
  deadline  : Datetime?
  minimum   : Float
  avgGrade  : Float?     = avg(submissions.grade)
  passPerc  : Float?     = count(submissions.filter(x=>x.pass)) / count(submissions)
}
entity Student {
  name      : String
}
entity Submission {
  name      : String     = assignment.name + " " + student.name          (on-demand)
  answer    : String?
  deadline  : Datetime? = assignment.deadline <+ parent.deadline        (default)
  finished  : Datetime?
  onTime    : Boolean    = finished <= deadline <+ true
  grade     : Float?     = if(conj(children.pass)) avg(children.grade) (default)
  pass      : Boolean    = grade >= assignment.minimum && onTime <+ false
}
relation Submission.student    1 <-> * Student.submissions
relation Submission.assignment 1 <-> * Assignment.submissions
relation Assignment.parent     ? <-> * Assignment.children
relation Submission.parent     ? =
  assignment.parent.submissions.find(x => x.student == student)
                               <-> * Submission.children
```

Figure 4 Running example IceDust2 specification.

assignment tree (see Alice's and Bob's submission trees in Figure 3). The tree structure of submissions is derived in order to avoid redundant data, which can lead to inconsistencies.

Assignments have optional `deadline`s. Student submissions inherit their `deadline` from the assignment or from their parent submission, unless the deadline is overridden by the instructor to provide a personal deadline for a student. For example, `mathBob`'s deadline in Figure 3 is supplied by the instructor, while `mathAlice`'s deadline is the assignment deadline. Leaf submissions are graded by assigning a grade to the `grade` attribute (overriding the default value), while the grades of non-leaf submissions depend on the grades of their child submissions. Note that students only receive a grade for a collection-submission if all of the child submissions are `pass`, and a submission is only a pass when its grade is above the minimum assignment grade and all its children pass. Finally, every assignment has an average grade and pass percentage.

Most derived values in this example are calculated `incrementally`, providing fast performance for reads. The course statistics are calculated `eventually`, providing better performance on writes to grades. Student grades need to be up-to-date, but statistics can be (temporarily) outdated. The submission name is calculated `on-demand` as it need not be cached. This example is interesting as it has a derived bidirectional relation (`Submission`'s parent-children) with a multiplicity bound on parent. Moreover, the derived relation is used in both directions in other derived values: `parent` is used in inheriting deadlines and `children` is used in calculating grades.

2.2 Orthogonality of Field Configurations in IceDust

An IceDust data model definition consists of *entities* with *fields*. Instantiations of entities are objects that assign *values* to fields. A field declaration specifies the *type* of values that can be assigned to the field and several other configuration elements. We analyze IceDust's configurability in terms of the feature model of Figure 1.

Multiplicities. A source of boilerplate code in regular programming languages are nullable values and explicit collections used to encode the cardinality of values. Instead of encoding cardinalities in (collection) types, IceDust supports the specification of *multiplicities* as a separate, orthogonal concern, following the work of Steinmann [29] and Harkes et al. [16]. Multiplicity modifiers on types express that a field has exactly one value (`1`), zero or one value (`?`), zero or more values (`*`), or one or more values (`+`). All operators are defined for all cardinalities of operands. For example, an expression calculating average grades based on children (implicit collection) and grade (implicitly nullable) is specified as:

```
mathAlice                      // : Submission  ~ 1
mathAlice.children             // : Submission  ~ *
mathAlice.children.grade       // : Float       ~ *
mathAlice.children.grade.avg() // : Float       ~ ?
```

Directionality. There are two kinds of fields. *Attributes* such as `grade` refer to a (collection of) primitive value(s). *Reference* fields refer to a (collection of) object(s). In object-oriented languages bidirectional relations between entities are modeled by a reference field on each side of the relation. Keeping such a relation consistent requires work. That is, when assigning to a field on one side of the relation, the other side should be made consistent with that assignment (as we will discuss in more detail in the next section). To avoid the associated boilerplate code, IceDust provides 'native' bidirectional relations between entities. For example, the following relation defines a tree structure for submissions:

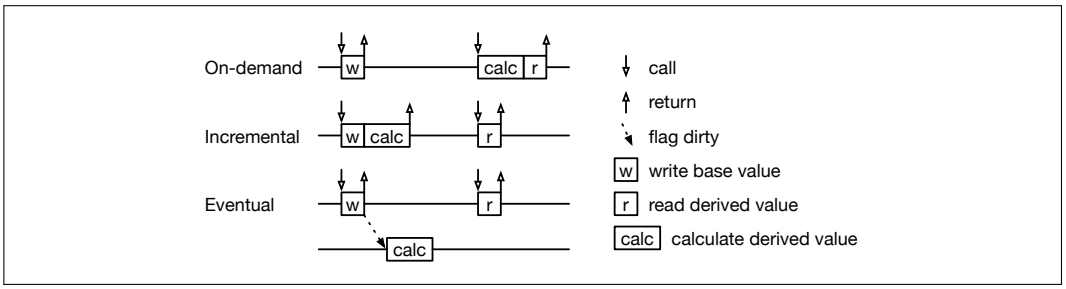

Figure 5 Thread activation diagrams for different calculation strategies.

```
entity Submission { }
relation Submission.children * <-> ? Submission.parent
```

IceDust guarantees that the reference fields that implement a relation are kept consistent at run time. Thus, IceDust supports unidirectional primitive valued attributes and bidirectional relations between entities. Note that multiplicities apply equally to attributes and the endpoints of relations.

Derivation Type. The values of *normal* attributes are directly assigned by (the users of) an application. Similarly, *normal* relations are constructed by an application. A *derived* value attribute specifies an expression that calculates the attribute's value from the values of other attributes and relations. For example, the `grade` attribute is defined as the average of the grades of the children's grades:

```
entity Submission {
  grade : Float? = children.grade.avg()
}
relation Submission.children * <-> ? Submission.parent
```

Derived and user-defined attributes can be combined in a `default`-valued attribute. If a value is explicitly assigned to such an attribute, that value is returned. Otherwise, the calculated (default) value is returned. For example, a submission grade can be calculated from its children's grades, but it can also be set by the instructor:

```
grade : Float? = children.grade.avg() (default)
```

Calculation Strategies. In object-oriented languages, calculated values can be specified with getter methods, encoding an on-demand calculation strategy; the value is calculated each time it is read. Switching to a cached implementation strategy requires invasive code changes. Derived value attributes in IceDust can be configured with different calculation strategies orthogonally to the expression of the calculation. The difference between the different calculation strategies is the point in time at which derived values are calculated. Figure 5 shows the differences by means of thread activation diagrams in response to incoming reads and writes. The `on-demand` strategy calculates derived values when they are read. This means that writes to base values, on which derived values can depend, will be fast, but reads of derived values will be slow. The `incremental` strategy recalculates all derived values that transitively depend on base value directly after an update to a base value. Writes will be slow, but reads will be fast. Finally, the `eventual` strategy schedules recalculating on a separate thread. Writes and reads will be fast, but consistency is not guaranteed: possibly outdated derived values might be read.

2.3 Generalizing Data Modeling with IceDust

IceDust limits the possible configurations of the feature model. First, only unidirectional fields (attributes) can be derived, not bidirectional relations. Second, all fields in an IceDust program are required to have the same calculation strategy. In this paper we relax these constraints to enable a more general combination of features.

Derived Relations. In the relational model, derived bidirectional relations can be expressed directly in relational terms. For example, the derived relation in Figure 2 is expressed in Datalog as follows:

```
submissionParent(?s1, ?s2) :-
  submissionAssignment(?s1, ?a1),
  submissionAssignment(?s2, ?a2),
  assignmentParent    (?a1, ?a2),
  submissionStudent   (?s1, ?st),
  submissionStudent   (?s2, ?st).
```

However, the relational paradigm specifies no multiplicity bounds: a `Submission` can have $[0, n)$ `parents`. (Which is a problem if a submission should inherit its parent deadline, and there might be multiple parents.) On the other hand, in reactive or incremental programming, for example with REScala [28], a multiplicity bound of $[0, 1]$ can be specified (the type is `Option[Submission]`):

```
class Submission {
  val parent: DependentSignal[Option[Submission]] = Signal {
    assignment().flatMap(_.parent()).map(_.submissions()).getOrElse(Nil)
      .find(_.student() == student())
  }
}
```

However, this only specifies a unidirectional relation. Making this relation bidirectional in REScala requires defining a children signal, keeping track of the previous parent, and updating the children signal on parent change events:

```
val children     : VarSynt[List[Submission]] = Var(Nil)
var oldParent    : Option[Submission]        = None
val parentChanged: Event[Option[Submission]] = parent.changed
parentChanged += ((newParent: Option[Submission]) => {
  oldParent.foreach { o => o.children() = o.children.get.filter(_ != this) }
  newParent.foreach { n => n.children() = this :: n.children.get }
  oldParent = newParent
})
```

To avoid such boilerplate and provide multiplicity bounds we generalize IceDust's derived values to apply to relations and attributes, rather than just attributes. A derived relation is expressed in IceDust2 as

```
relation Entity1.field1 multiplicity = expr  <-> multiplicity Entity2.field2
```

where the expression defines how to compute the left-hand side of the relation. The parent-child relation of submissions in our example can be expressed as follows:

```
relation Submission.parent ? =
  assignment.parent.submissions.find(x => x.student == student)
    <-> * Submission.children
```

Figures 2-3 show the model and some example data for this derived relation respectively. The derived relation is specified on the left-hand side, but can be used inversely, from the right-hand side, as well. For example, using `children` in calculating the average grade:

```
entity Submission {
  grade : Float? = children.grade.avg()
}
```

Composition of Calculation Strategies. We extend IceDust with composition of calculation strategies. Strategy composition enables using different strategies for different parts of the program. For example, in our running example, student grades are always required to be consistent, but course statistics may be out of date (temporarily) for better performance. We can express this by calculating student grades incrementally, while calculating course statistics eventually:

```
entity Assignment {
  avgGrade : Float? = submissions.grade.avg() (eventual)
}
entity Submission {
  grade    : Float? = children.grade.avg()    (incremental)
}
relation Submission.children    * <-> ? Submission.parent
relation Assignment.submissions * <-> 1 Submission.assignment
```

The calculation strategies can be specified on modules, entities, and individual fields. If a strategy is not specified, the field inherits it from its entity or module. The default strategy is `incremental`, as all other strategies can depend on it (see Section 5 for more details).

Constraints on Feature Composition. IceDust2 allows almost all combinations of features in Figure 1, but we impose three restrictions. First, we disallow unsound composition of calculation strategies as we will discuss in Section 5.

Second, derived relations can only be used inversely if they are materialized (`incremental` and `eventual` calculation). Navigating inversely in `on-demand` would require either materializing or coming up with an inverse expression. Consider the following derived relation:

```
relation Submission.root 1 = parent.root<+this <-> * Submission.rootDescendants
```

It defines the root for each submission in the tree. Reading `root` in `on-demand` is trivial: execute the expression `parent.root <+ this` (take your parent's root, or take yourself). The inverse for this bidirectional relation is `rootDescendants`: for the root, all its descendants, and for all non-root nodes, nothing. In `incremental` and `eventual` we can use the materialized `rootDescendants` for reads. But, in `on-demand` the compiler would need to come up with an expression that computes exactly the inverse of `root` which is non-trivial:

```
relation Submission.descendants * = this ++ children.descendants
  <-> * Submission.ancestors
relation Submission.rootDescendants*= if(count(parent)==0) descendants else null
  <-> 1 Submission.root
```

In this example we need a helper relation to compute the transitive closure.

Third, we disallow `default` derived relations since their behavior is unexpected. Consider the following example:

```
entity Student { }
entity Committee { }
relation Committee.members     *                    <-> * Student.committees
relation Committee.mailingList * = members (default) <-> * Student.subscriptions
```

We have specified the `mailingList` of a `Committee` to be its `members` by default. Now, if a member is added, and there is no user-provided value, the member will be added to the mailing list. But, if some student had also subscribed, the user-provided value will be used,

which will not be updated with the new member. Better would be to get the desired behavior by combining the committee members and the mailing list in a new derived value:

```
relation Committee.members          * <-> * Student.committees
relation Committee.mailingList      * <-> * Student.subscriptions
relation Committee.fullMailingList  * = members ++ mailingList
                                      <-> * Student.allSubscriptions
```

3 Run-Time Feature Interaction

In the previous section we generalized the configurability of fields in IceDust2 data models. As a result, features can be combined independently (up to semantic soundness). While the *selection* of features in a data model specification is orthogonal, each combination of multiplicity, directionality, derivation type, and calculation strategy requires a *specialized implementation* to produce consistent results. In this section we examine the nature of this run-time feature interaction before addressing the resulting complexity in the next section.

Incrementality and Bidirectional Updates. Maintaining bidirectionality and updating incremental derived values happen on writes and are mutually recursive. In Figure 3, consider executing `lab.setParent(exam)`, moving the `lab` from `math` to `exam`. Bidirectional maintenance will update `math.children` and `exam.children`. This will trigger incremental updates for `Submission.children` fields, which will in turn update `Submission.parent` fields, which will trigger updates for `Submission.deadline` fields, etcetera. Thus, it is not possible to define incrementality behavior orthogonally to the bidirectional maintenance behavior.

Multiplicities Guide Bidirectional Updates. When maintaining bidirectionality, multiplicity bounds have to be respected. Multiplicity upper bounds are respected by implicitly removing old values if needed. For example, executing `exam.addToChildren(lab)` will implicitly remove `math` as `parent` from `lab`. The behavior is identical to executing `lab.setParent(exam)`. Figure 6 shows the result of writes to bidirectional relations while preserving bidirectionality and respecting multiplicity upper bounds. Behavior 7 is executed on `lab.setParent(exam)`, and behavior 10 on `exam.addToChildren(lab)`. Both will implicitly remove the old `parent` of `lab`. The alternative to implicitly removing old values would be to fail when calling `exam.addToChildren(lab)`. This is what the Booster language does [5]; it only updates objects referenced explicitly in the update operation. But, it would be verbose to have to call `math.removeFromChildren(lab)` first. Multiplicity lower bounds are respected by failing the operation on a violation, as implicitly adding relations with arbitrary objects is undesirable. For example, on deleting `exam`, the multiplicity lower bounds of `examAlice.assignment` and `examBob.assignment` are violated. But, implicitly setting `examAlice.assignment` to `lab` is undesirable. The behavior of bidirectional maintenance varies with multiplicity bounds. Thus, it is not possible to define the bidirectional maintenance behavior orthogonally to the behavior for respecting multiplicity bounds.

Minimizing Setter Calls for Incrementality. For incrementality it is important to minimize the (internal) calls to setters, as duplicate setter calls will duplicate dirty flagging of derived values that depend on it. If we look at Figure 6, behavior 2, then we should not first call `b2.setA(null)` and subsequently `b2.setA(a1)` during bidirectional maintenance. So, rather than first removing `a2-><-b2` and subsequently adding `a1-><-b2`, the algorithm should

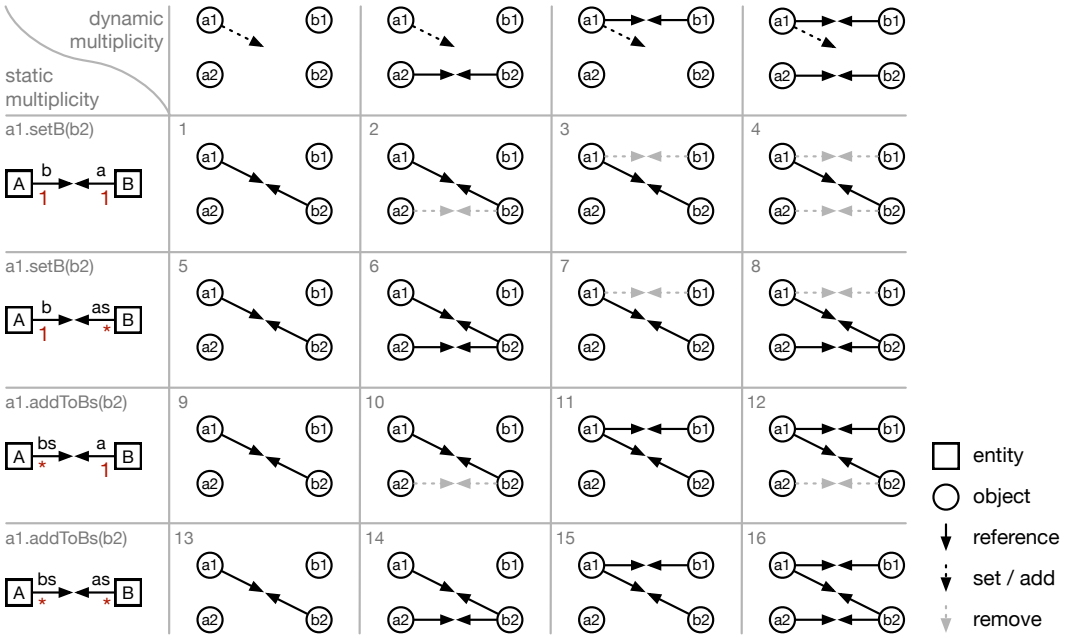

Figure 6 Update a bidirectional relation and preserve both bidirectionality and multiplicity upper bounds. Left column shows class diagram with multiplicity bounds, the top row shows starting object graph, and 1-16 show the object graph after update.

update `a1.b`, `a2.b`, and `b2.a` directly. The behavior maintaining bidirectionality needs to trigger the *minimal* number of incremental updates.

Only Trigger Updates on Observable Changes. An additional way to minimize incremental update computation is updating only on observable changes. The various derivation types influence this. If a `normal` attribute is assigned the same value as it previously had, there is no need to trigger updates. Default values have various scenarios in which updates are not observable. Suppose we would 'override' the `grade` of `mathAlice` with a 7.5 in Figure 3. This should not trigger any updates, as the default value was 7.5 already (the average of 7.0 and 8.0). If we change the `grade` of `examAlice` to a 9.0 after that, we trigger an update for `mathAlice.grade`. But we can stop propagating at that point because the new average (8.5) is not visible; we overrode the grade with 7.5. When writing to a field, an update should only be triggered when the change is observable. Thus, the incremental update behavior cannot be defined orthogonally to the derivation type behavior.

Only Trigger Updates for Incremental and Eventual. Finally, updates only need to be triggered for derived value fields that are updated on writes (`incremental` and `eventual`). Fields only referenced in `on-demand` derived value fields do not need to send update triggers (for example `Assignment.name` in Figure 4). Note that if we would change `Submission.name` to `incremental`, `Assignment.name` does need to send update triggers. Thus, the calculation strategy behavior of a field can not be defined orthogonally to the calculation strategy behaviors of the fields that reference it.

Summary. In summary, derived values, bidirectional relations, multiplicity bounds, and calculation strategies all interact with each other. These interactions are hidden from the

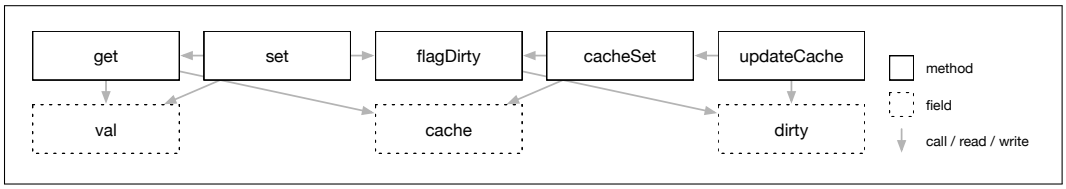

Figure 7 General overview for the semantics of a single field in IceDust2.

language users in the getters and setters of fields. Because all these features interact, they cannot be implemented separately. Creating different specialized getters and setters for all possible feature combinations is also not an option; the feature model has 384 valid configurations. (The number of configurations, without any restrictions, and ignoring flow calculation strategies, is $6 * 7 * 3 * 3 * 2 * 2 = 1512$. With the `implies` restrictions it is 384.) With about 20 to 100 lines of code generated for getters and setters, specifying all specialized getters and setters would be roughly 20000 lines of code. This amount of code would pose a serious maintenance problem, and would make it impossible to reason about correctness. Our solution is to implement this as a compact product-line for each field. We discuss this in the next section.

4 Operational Semantics

An IceDust2 data model consists of entities with fields, representing attributes and relations. The public API of such a data model consists of entity instantiation, object deletion, reading the value of a field (`get`), and changing the value of a field (`set`). The previous section showed that IceDust2's features are not compositional, leading to over 300 different configurations for fields with as many getter/setter definitions. In this section we define the operational semantics for these getters and setters by factoring out variability into mutually dependent auxiliary methods. Moreover, we argue that all these behaviors maintain bidirectionality, respect multiplicity bounds, and maintain caches for incrementality.

Figure 7 gives an overview of the semantics of a single field. A field is represented at runtime by at most three fields: a user value, a derived value cache, and a dirty flag. The `get`ter is responsible for returning the correct value on a read. The `set`ter is responsible for maintaining bidirectionality and multiplicity bounds in the `userValue`. Moreover, it calls `flagDirty` on observable changes. The `cacheSet`ter does the same for `cacheValue`s. The incremental update algorithm (not shown in Figure 7, as it is global) reads the `dirtyFlag`s, and calls `updateCache` to maintain derived value caches. How these fields and methods are implemented varies based on the configurations in the feature model.

We specify the operational semantics of IceDust2 using big-step semantics. The reduction rules modify a store. The store can contain a user value, a cached value, and a dirty flag for every field in every object (Figure 8). We omit the store in a rule when it is not directly used in the rule. When we omit the store, it is implicitly threaded from left to right. Note that in list comprehensions the store is threaded as well. For conciseness, all rules operate on lists of values, even if fields have a multiplicity upper bound of 1. In the rules, we use '$\in$' for testing whether a field has a certain configuration in the feature model. For example, '$f \in$ incremental' is true if the field uses the `incremental` calculation strategy. We use '.' for accessing related information. For example, 'f.expr' denotes the expression of field f, and 'f.inverse' denotes the inverse field of a bidirectional relation.

$$\Sigma \in Store : EntityReference \times Field \mapsto (\text{val} \mapsto [Value],\ \text{cache} \mapsto [Value],\ \text{dirty} \mapsto Boolean)$$
$$Value : EntityRerence \mid PrimitiveValue$$

Figure 8 The store maps combinations of references and field names to tuples of three: user value, cached value, and dirty flag.

$$\frac{f \in \text{normal}}{o.\text{get}(f)/\Sigma \Downarrow \Sigma[o,f].\text{val}/\Sigma}\ [\text{Get1}] \qquad \frac{}{V.\text{get*}(f) \Downarrow [v | v \in V_2,\ o.\text{get}(f) \Downarrow V_2,\ o \in V]}\ [\text{Get}*]$$

$$\frac{f \in \text{default} \quad \Sigma[o,f].\text{val} = V \neq [\,]}{o.\text{get}(f)/\Sigma \Downarrow V/\Sigma}\ [\text{Get2}] \qquad \frac{f \in \text{on-demand} \quad o.\text{calc}(f) \Downarrow V}{o.\text{getCalc}(f) \Downarrow V}\ [\text{GetCalc1}]$$

$$\frac{f \in \text{default}\ \ \Sigma[o,f].\text{val}=[\,]\ \ o.\text{getCalc}(f) \Downarrow V}{o.\text{get}(f)/\Sigma \Downarrow V/\Sigma}\ [\text{Get3}] \qquad \frac{f \in \text{incremental}}{o.\text{getCalc}(f)/\Sigma \Downarrow \Sigma[o,f].\text{cache}/\Sigma}\ [\text{GetCalc2}]$$

$$\frac{f \in \text{derived} \quad o.\text{getCalc}(f) \Downarrow V}{o.\text{get}(f) \Downarrow V}\ [\text{Get4}] \qquad \frac{o \vdash (f.\text{expr}) \Downarrow V}{o.\text{calc}(f) \Downarrow V}\ [\text{Calc}]$$

Figure 9 Getter evaluation rules.

Getter. Figure 9 defines the evaluation rules for getters. Method `get` behaves differently depending on the derivation type. The rule for `normal` just reads the user value of the field [Get1]. The rule for `default` reads the user value [Get2], but if that is not present (empty list of values), the calculated value is returned [Get3]. (It is not possible to override a calculated value with an absent user value.) The rule for `derived` returns the calculated value [Get4]. Method `get*` maps a getter over a collection of objects, which is used in the compilation of expressions. The rules for `getCalc` call `calculate` for `on-demand` [GetCalc1], but read the cached value for `incremental` [GetCalc2]. Finally, `calculate` calculates a value using the expression of the field. Note that in expression evaluation ($o \vdash$ `this` $\Downarrow [o]$) the o before the turnstyle binds `this`. We omit the rules for expression evaluation as they are standard.

The `on-demand` and `incremental` calculation strategies should return the same values on field reads. (Except for cyclic definitions, which we will discuss later.) When the getter is called, `incremental` (`default` or `derived`) fields should have a cached value equal to re-evaluating the expression, and there should be no dirty flags:

▶ Invariant 1 (Incrementality). $\forall\ E.f \in \text{incremental},\ \ \forall o \in E, \Sigma[o,f,\text{dirty}] = false \quad \Rightarrow$
$\forall\ E.f \in \text{incremental},\ \ \forall\ o : E,\ \ o.\text{calc}(f) \Downarrow \Sigma[o,f,\text{cache}]$

If the cached value contains the exact value that `calculate` would compute if executed, then the `incremental` getter will return the same value as the `on-demand` getter. The setter and update algorithm should keep the cached value up-to-date.

Setter. Figure 10 defines the evaluation rules for setters. Method `set` is responsible for maintaining bidirectionality and multiplicity upper bounds. For attributes, `set` does not have to maintain bidirectionality so it passes the call through to `setIncr` [Set1]. For relations, `set`'s behavior varies depending on multiplicity bounds [Set2]. References on $V.(f.\text{inverse})$ are removed by `addIncr` if the multiplicity upper bound is 1 [AddIncr1]. The inverses of these references are implicitly removed by `remInv` [RemInv2]. This realizes the behavior visualized in Figure 6. Method `setIncr` is responsible for dirty flagging on observable changes [SetIncr2]. Method `cacheSet` is identical to the `set` method, updating cache values rather than user values.

$$\frac{f \notin \text{bidir} \quad f \sim [\_, u] \quad |V| \leq u \quad o.\text{setIncr}(f, V) \Downarrow}{o.\text{set}(f, V) \Downarrow} \text{[Set1]}$$

$$\frac{\begin{array}{l} f \in \text{bidir} \quad f \sim [\_, u] \quad |V| \leq u \\ V_{old} = \Sigma[o, f].\text{val} \\ V_{add} = V \setminus V_{old} \\ V_{rem} = V_{old} \setminus V \\ [v_{add}.\text{remInv}(f.\text{inverse}) \Downarrow \mid v_{add} \in V_{add}] \\ o.\text{setIncr}(f, V) \Downarrow \\ [v_{rem}.\text{remIncr}(f, o) \Downarrow \mid v_{rem} \in V_{rem}] \\ [v_{add}.\text{addIncr}(f, o) \Downarrow \mid v_{add} \in V_{add}] \end{array}}{o.\text{set}(f, V) \Downarrow} \text{[Set2]}$$

$$\frac{f \sim [\_, 1] \quad \Sigma[o, f].\text{val} = [\,]}{o.\text{remInv}(f)/\Sigma \Downarrow /\Sigma} \text{[RemInv1]}$$

$$\frac{f \sim [\_, 1] \quad \Sigma[o, f].\text{val} = [v] \quad v.\text{setIncr}(f.\text{inverse}, [\,])/\Sigma \Downarrow /\Sigma_2}{o.\text{remInv}(f)/\Sigma \Downarrow /\Sigma_2} \text{[RemInv2]}$$

$$\frac{f \sim [\_, n)}{o.\text{remInv}(f) \Downarrow} \text{[RemInv3]}$$

$$\frac{f \sim [\_, 1] \quad o.\text{setIncr}(f, [v]) \Downarrow}{o.\text{addIncr}(f, v) \Downarrow} \text{[AddIncr1]}$$

$$\frac{f \sim [\_, n) \quad V = \Sigma[o, f].\text{val} +\!\!+ [v] \quad o.\text{setIncr}(f, V)/\Sigma \Downarrow /\Sigma_2}{o.\text{addIncr}(f, v)/\Sigma \Downarrow /\Sigma_2} \text{[AddIncr2]}$$

$$\frac{o.\text{setIncr}(f, \Sigma[o, f].\text{val} \setminus v)/\Sigma \Downarrow /\Sigma_2}{o.\text{remIncr}(f, v)/\Sigma \Downarrow /\Sigma_2} \text{[RemIncr]}$$

$$\frac{f \in \text{incremental} \quad o.\text{get}(f)/\Sigma \Downarrow V_2 \quad \Sigma_2 = \Sigma[o, f, \text{val} \mapsto V] \quad o.\text{get}(f)/\Sigma_2 \Downarrow V_2}{o.\text{setIncr}(f, V)/\Sigma \Downarrow /\Sigma_2} \text{[SetIncr1]}$$

$$\frac{\begin{array}{l} f \in \text{incremental} \quad o.\text{get}(f)/\Sigma \Downarrow V_2 \\ \Sigma_2 = \Sigma[o, f, \text{val} \mapsto V] \quad o.\text{get}(f)/\Sigma_2 \Downarrow V_3 \\ V_2 \neq V_3 \quad o.\text{dirtyFlows}(f)/\Sigma_2 \Downarrow /\Sigma_3 \end{array}}{o.\text{setIncr}(f, V)/\Sigma \Downarrow /\Sigma_3} \text{[SetIncr2]}$$

Figure 10 Setter evaluation rules.

$$\frac{[v.\text{flagDirty}(f_2) \Downarrow \mid v \in V, \ o \vdash expr \Downarrow V, \ f_2 \in \text{incremental}, \ expr.f_2 \in f.\text{flows}]}{o.\text{dirtyFlows}(f) \Downarrow} \text{[DirtyFlows]}$$

$$\frac{\Sigma_2 = \Sigma[o, f, \text{dirty} \mapsto \text{true}]}{o.\text{flagDirty}(f)/\Sigma \Downarrow /\Sigma_2} \text{[FlagDirty]}$$

Figure 11 Flag dirty evaluation rules.

$$\frac{o.\text{calc}(f) \Downarrow V \quad o.\text{cacheSet}(f, V) \Downarrow}{o.\text{update}(f) \Downarrow} \text{[Update]}$$

$$\frac{[o.\text{update}(f) \Downarrow \mid o \in V]}{V.\text{update*}(f) \Downarrow} \text{[Update*]}$$

$$\frac{V = [o \mid \Sigma[o, f, \text{dirty}] = \text{true}] \quad V.\text{clean*}(f)/\Sigma \Downarrow /\Sigma_2 \quad V.\text{update*}(f)/\Sigma_2 \Downarrow /\Sigma_3}{\text{updateCache*}(f)/\Sigma \Downarrow /\Sigma_3} \text{[UpdateCache*]}$$

$$\frac{\Sigma_2 = \Sigma[o, f, \text{dirty} \mapsto \text{false}]}{v.\text{clean}(f)/\Sigma \Downarrow /\Sigma_2} \text{[Clean]}$$

$$\frac{[v.\text{clean}(f) \Downarrow \mid v \in V]}{V.\text{clean*}(f) \Downarrow} \text{[Clean*]}$$

$$\frac{[o \mid \Sigma[o, f, \text{dirty}] = \text{true}] \neq [\,]}{\text{hasDirty*}(f)/\Sigma \Downarrow \text{true}/\Sigma} \text{[HasDirty*1]}$$

$$\frac{[o \mid \Sigma[o, f, \text{dirty}] = \text{true}] = [\,]}{\text{hasDirty*}(f)/\Sigma \Downarrow \text{false}/\Sigma} \text{[HasDirty*2]}$$

Figure 12 Update evaluation rules.

$$\frac{[\text{maintGroup}^*(g) \mid g \in p.\text{topo}]}{\text{maintCache}^*(p) \Downarrow}\ [\text{MaintCache}*]$$

$$\frac{[\text{updateCache}^*(f) \mid f \in g] \quad \forall f \in g,\ \neg\text{hasDirty}^*(f)}{\text{maintGroup}^*(g) \Downarrow}\ [\text{MaintGroup}*1]$$

$$\frac{[\text{updateCache}^*(f) \mid f \in g] \quad \exists f \in g,\ \text{hasDirty}^*(f) \quad \text{maintGroup}^*(g) \Downarrow}{\text{maintGroup}^*(g) \Downarrow}\ [\text{MaintGroup}*2]$$

Figure 13 Update algorithm evaluation rules.

For each object, for each field that is bidirectional, it should hold that if the field refers to another object, the other object also refers back to this object from the inverse field:

▶ Invariant 2 (Bidirectionality). $\forall\ E.f \in \text{bidir},\ \forall\ o_1 : E,\ o_2 \in o_1.f_1\ \Rightarrow\ o_1 \in o_2.(f.\text{inverse})$

Moreover, a read from a field should always return a list of values the size of which is smaller than or equal to the multiplicity upper bound:

▶ Invariant 3 (Multiplicity Upper Bound). $\forall\ E.f \sim [\_, u],\ \ \forall\ o : E,\ \ |o.f| \leq u$

The rules for `set` satisfy these two properties by construction; they generalize Figure 6 to work on collections of values. The setter is also partially responsible for Invariant 1. Whenever `get` of a field returns a different value, `setIncr` will call `dirtyFlows`. If `dirtyFlows` sets all dependent values dirty, and all dirty values are updated, Invariant 1 holds.

Flag Dirty. Whenever a value is observably changed, all `incremental` derived values that depend on it are flagged dirty. Figure 11 defines the evaluation rules for dirty flagging. Method `dirtyFlows` traverses the data-flow expressions, and calls `flagDirty` to flag the appropriate field dirty. Note that `dirtyFlows` only calls `flagDirty` for flows that end in a field that is `incremental`, as `on-demand` does not require dirty flagging. The data flows are obtained by path-based abstract interpretation. The basic idea is that all fields referenced in an expression are dependencies, and that the inversion of these dependencies determines the data flow. (For more details on data flow, see the IceDust paper [15].)

The `flagDirty` method is also partially responsible for Invariant 1. Method `dirtyFlows` flags all derived values dirty that depend on the changed value. If the incremental update algorithm updates all cached values that are dirty, Invariant 1 holds.

Update Cache. After changes, the caches have to maintained, so that reads return up-to-date values. Figure 12 defines the evaluation rules for cache updates. Method `update` is responsible for updating the cache of a single field for a single object. Method `updateCache*` updates the field in all objects that have this field dirty. Together with `updateCache*`, `hasDirty` is the API for the cache maintenance algorithm.

These methods are partially responsible for Invariant 1 as well. Method `cacheUpdate` ensures that Invariant 1 hold for a single field of a single object after its execution. However, updating the cache of a field might invalidate the cache of another. So, the incremental update algorithm calls `updateCache*` until `hasDirty*` evaluates to `false` for all fields.

Incremental Update Algorithm. The update algorithm is responsible for cleaning all caches. The evaluation rules for the update algorithm are defined in Figure 13. The data-flow analysis provides a topological ordering which can be used for scheduling updates [15]. Method `maintCache*` invokes `maintGroup*` for each connected component in topological order. Method `maintGroup*` invokes itself recursively as long as the group `hasDirty*`.

Invariant 1 is now satisfied by the fact that groups can only dirty flag fields in their own group or later groups, and each group is updated until no more dirty flags remain.

Note that in this operational semantics, transactions have to be managed manually. First constructors, `set` and `delete` are invoked, then `maintainCache*` has to be invoked, and only then `get` and `get*` are guaranteed to return values that are up-to-date. Transactions can be made implicit by invoking `maintainCache*` directly from `set`.

Object Creation and Deletion. On object creation all `incremental` fields of that object are dirty flagged. Before object deletion, all fields are set to `null` (or empty collections) to ensure bidirectionality and incrementality are maintained for the fields of other objects. Creation and deletion behavior do not vary based on different field features.

Multiplicity Lower Bounds. So far we have ignored multiplicity lower bounds:

▶ Invariant 4 (Multiplicity Lower Bound). $\forall\ E.f \sim [l, \_],\quad \forall\ o : E,\quad |o.f| \geq l$

These are checked at the end of transactions. (We have omitted transactions from the evaluation rules for conciseness.) If any of the multiplicity lower bounds is violated, the whole transaction is reverted.

Eventual Calculation Strategy. We have also omitted the eventual calculation strategy in the semantics. The eventual calculation strategy is implemented by taking the incremental update algorithm, but running this in a separate thread, and updating a single field of a single object at the time. To keep track of the dirty flags for eventual calculation, a fourth element in the store tuples is required: `dirtyEventual`. (In the implementation `dirtyEventual` flags are shared across all threads while `dirty` flags are thread-local.) The dirty flags for eventual calculation do not have to be cleaned before ending a transaction. But, when all dirty flags are cleaned, then all eventually calculated values are up-to-date:

▶ Invariant 5 (Eventuality).
$$\begin{aligned} &\forall\ E.f \in \text{incremental},\quad \forall o \in E, \Sigma[o, f, \text{dirty}] = false && \wedge \\ &\forall\ E.f \in \text{eventual},\quad \forall o \in E, \Sigma[o, f, \text{dirtyEventual}] = false && \Rightarrow \\ &\forall\ E.f \in \text{eventual},\quad \forall\ o : E,\quad o \vdash f.\text{expr} \Downarrow \Sigma[o, f, \text{cache}] \end{aligned}$$

Discussion: Computation Cycles. The `on-demand` and `incremental` calculation strategy produce the same values locally. But, in cyclic data flow their behavior is different. Consider the following program:

```
entity Foo {
  a : Int
  b : Int = a <+ c    // if(count(a) > 0) a else c
  c : Int = b
}
```

If `a` is not set, and `c` is read, `on-demand` will not terminate, but `incremental` will return `null`. If `a` is set, and `c` is read, both strategies will return the same value. If after that, `a` is set to `null` and `c` is read again, `incremental` will still return the previous value of `c` as it is cached in both `b` and `c`, while `on-demand` will not terminate again.

The `incremental` calculation strategy satisfies Invariant 1, as all derived values are consistent with each other. Invariant 1 is the same as the property guaranteed by synchronous reactive programming [22, 28]. In incremental computing with Adapton, a stronger property is guaranteed: incremental computation returns identical results to from-scratch computation [13, 14]. Note that in Adapton cyclic programs cannot be expressed, as cyclic computations

cannot be constructed. For acyclic data flows, IceDust2 satisfies the same property as Adapton: `incremental` calculation returns the same value as `on-demand` calculation.

5 Sound Composition of Calculation Strategies

In this section we examine how different calculation strategies can be composed. In composition the strategies need to evaluate to the right answers, and do so within their time constraints. Moreover, we introduce a type system that statically checks the safety of the composition of calculation strategies in an IceDust2 program.

Some systems for computing derived values allow composing various calculation strategies. However, the composition is not always checked for correctly calculating derived values. Derived values should be consistent with the values they depend on. On-demand values are not aware of changes to their dependencies, and they do not notify the derived values depending on them of changes. For example, in REScala `on-demand` values can be accidentally referenced in `reactive` values, causing reactive values not to be updated on changes to their dependencies. Take the following example:

```
class Student {
  val name   :VarSynt[String]         = Var("")                        //reactive
  val city   :VarSynt[String]         = Var("")                        //reactive
  val street :VarSynt[String]         = Var("")                        //reactive
  def address:String                  = street.get + " " + city.get  //on-demand
  val summary:DependentSignal[String]= Signal{name() + " " + address}//reactive
}
```

A change to `name` will trigger an update to `summary`, so summary will be consistent with name. Accessing `address` will read the latest values from `city` and `street`, so it will be consistent with its dependencies as well. But, `summary` is not updated after a change to `city` or `street`, so `summary` is not consistent with all its dependencies.

In IceDust, letting an `incremental` field depend on an `on-demand` field would have the same problem. Changing the `incremental` strategy to reevaluate `on-demand` referenced fields would make reads of `incremental` fields slower. (A cache read is O(1), reevaluating might be expensive.) We designed IceDust2 to have predictable performance, so we chose to prevent the above situation by a type system.

Type Checking Strategy Composition. IceDust2 features three calculation strategies: `on-demand`, `incremental`, and `eventual` (Figure 5). The `on-demand` strategy is pull-based, while the `incremental` and `eventual` strategies are push-based. Push-based derived values are recalculated on changes to base values, while pull-based derived values are calculated when they are read. Pull-based derived values can depend on push-based derived values, but not the other way around, as pull-based values would not notify the push-based values of changes. Within the push-based strategies, `eventual` can depend on `incremental`, but not the other way around. An `incremental` derived value depending on an `eventual` derived value would be eventually calculated rather than be up-to-date. An `on-demand` derived value depending on an `eventual` derived value is not always up-to-date, so we create a new strategy, `on-demand eventual`, to reflect this. Finally, any calculation strategy can depend on values entered by users, so we also create a new strategy `base-value` for that. We combine these five strategies in a lattice such that strategies in the lattice can depend on strategies below them (Figure 14, right).

This lattice is used to check the composition of calculation strategies in IceDust2 programs. The general idea is to check what strategy is used for each sub-expression of derived values,

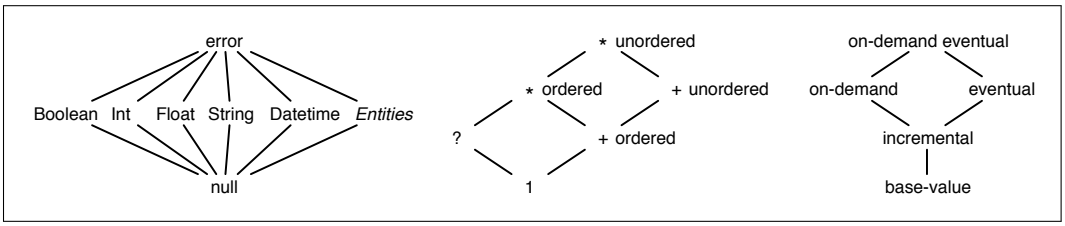

Figure 14 IceDust2's type lattice (left), multiplicity and ordering lattice (middle), and composition of calculation strategies lattice (right).

Expression Strategy Composition $\boxed{\Gamma \vdash Expr \Uparrow S}$

$$\frac{c \text{ is constant}}{c \Uparrow \text{base-value}}\ [\text{Const}] \qquad \frac{\oplus \in UnOp \quad e \Uparrow s}{\oplus\, e \ \Uparrow\ s}\ [\text{UnOp}]$$

$$\frac{}{\text{this} \Uparrow \text{base-value}}\ [\text{This}] \qquad \frac{\oplus \in BinOp \quad e_1 \Uparrow s_1 \quad e_2 \Uparrow s_2}{e_1 \oplus e_2 \ \Uparrow\ s_1 \sqcup s_2}\ [\text{BinOp}]$$

$$\frac{\neg\Gamma(m) \quad f.\text{stratComp} = s}{\Gamma \vdash f \Uparrow s}\ [\text{NavStart}] \qquad \frac{e_1 \Uparrow s_1 \quad e_2 \Uparrow s_2 \quad e_3 \Uparrow s_3}{e_1\,?\,e_2 : e_3 \ \Uparrow\ s_1 \sqcup s_2 \sqcup s_3}\ [\text{TenOp}]$$

$$\frac{e \Uparrow s_1 \quad f.\text{stratComp} = s_2}{e\,.\,f \Uparrow s_1 \sqcup s_2}\ [\text{Nav}] \qquad \frac{\Gamma \vdash e_1 \Uparrow s_1 \quad \Gamma[x \mapsto s_1] \vdash e_2 \Uparrow s_2}{\Gamma \vdash e_1.\text{filter}(x => e_2) \ \Uparrow\ s_1 \sqcup s_2}\ [\text{Filter}]$$

$$\frac{}{\Gamma \vdash x \Uparrow \Gamma(x)}\ [\text{Var}]$$

Field and Program Strategy Composition $\boxed{Field|Prog \Uparrow}$

$$\frac{f.\text{stratComp} = s_{def} \quad \emptyset \vdash f.\text{expr} \Uparrow s_{expr} \quad s_{def} \sqsupseteq s_{expr}}{f \in Field \Uparrow}\ [\text{Field}]$$

$$\frac{\forall e \in p.\text{entities},\ \forall f \in \{f \mid f.\text{expr},\ f \in e.\text{fields}\},\ f \Uparrow}{p \in Prog \Uparrow}\ [\text{Prog}]$$

Figure 15 Strategy composition rules.

and whether these are lower in the lattice than the definition of the derived value specifies. The reduction rules for the strategy composition type system are defined in Figure 15. The environment (Γ) maps variable names to strategies.

Constants [Const] and `this` [This] are base values. Field dereference on `this` has the strategy of the field definition [NavStart]. If the field has derivation type normal, it is a base value. The strategy of a field dereference on an object is the least-upper-bound of the strategy of the sub-expression and strategy of the field definition [Nav]. Unary operators pass on their strategy [UnOp], and both binary and ternary operators take the least-upper-bound of their sub-expression strategies [BinOp, TenOp]. The `filter` stores the strategy of the variable in the environment [Filter], and variables read their strategy from the enviroment [Var]. A field is sound if its expression calculation strategy is less than or equal to its defined calculation strategy [Field], and finally, a program is sound if all entity fields with expressions are sound [Prog].

Example. Lets apply these rules to an example. Suppose we extend `Submission` with:

```
summary : String =
  name + (if(pass) " pass" else " fail") + " grade = " + (grade <+ "none") +
  " (average = " + (assignment.avgGrade <+ "none") + ")"
```

Type checking sub-expressions yields the following:

```
name                                          // on-demand
pass                                          // incremental
" pass"                                       // base-value, idem all literals
(if(pass) " pass" else " fail")               // incremental
name + (if(pass) " pass" else " fail")        // on-demand
grade                                         // incremental
assignment                                    // incremental
assignment.avgGrade                           // eventual
assignment.avgGrade <+ "none"                 // eventual
name + ... + (assignment.avgGrade <+ "none") // on-demand eventual
```

The sub-expression `name` is `on-demand`, and the sub-expression `assignment.avgGrade` is eventual. These two strategies are propagated through the operators until they meet in a `+` operator. The `+` operator takes the least-upper-bound of both strategies, which is `on-demand eventual`. So the definition of `summary` needs to be annotated with `(on-demand eventual)`.

It is possible to perform strategy inference instead of checking consistency of annotations. However, it is not clear whether that would improve usability or not. In our example, the programmer might not notice that the inferred strategy is `on-demand eventual`, and assume that the summary would always be up-to-date. So, we require annotating derived value fields with their calculation strategy, or inheriting the strategy from the entity or module.

6 Implementations

We discuss two IceDust2 compilers. The first compiler closely matches the operational semantics in Section 4. It compiles to single threaded, in-memory, plain old Java objects. The second compiler serves a more complicated context. It compiles to an object-relational mapper with transaction semantics.

Compilation to Java. The compilation to Java closely matches the semantics in Section 4. It does not feature transactions (no multiplicity lower-bound runtime checks), and does not feature eventual calculation (it is single threaded). The translation from semantics to a code generator for Java code is straightforward. The store (fields, caches, and dirty flags) are compiled to fields in classes, and the arrows to methods. However, the compiler is not a literal translation of the operational semantics: the compiler makes multiplicity, calculation strategy and derivation-type choices at compile time, and leaves the remaining behavior to run time. Moreover, the compiler specializes types for various multiplicities.

An example of this compile-time/run-time split is the code generation for `get` (Figure 16). The semantics has two rules for the default-value behavior [Get2, Get3], but the compiler defers this decision to run time by compiling to an `if` statement. Another example is the code generator for the `set` method. The compiler makes bidirectionality and multiplicity upper bound choices, so it has six implementations. For these six implementations, it inlines rule [RemInv], or omits it if it has no effect. Figure 17 shows two of the implementations. The first variation is specialized to multiplicities with an upper bound of 1, so it has to deal with `null` values. The second variation is a literal translation of [Set2] without the [RemInv] calls. (The multiplicity upper-bounds of n never force implicit removals of references.)

```
fieldname-to-java-classbodydec: x_name -> get
  x_get           := $[get[<ucfirst>x_name]];
  x_getCalculated := $[getCalculated[<ucfirst>x_name]];
  t               := <type-and-mult-to-java-type>x_name;
  switch id
    case is-normal:  get := cbd |[
      public ~type:t x_get(){ return x_name; }
    ]|
    case is-default: get := cbd |[
      public ~type:t x_get(){
        if(x_name!=null && !x_name.equals(new HashSet<~type:t>())) return x_name;
        return x_getCalculated();
      }
    ]|
    case is-derived: get := cbd |[
      public ~type:type x_get(){ return x_getCalculated();}
    ]|
  end
```

Figure 16 Java code generation for `get()`. The `cbd|[ ]|` parses a Java class body declaration with meta-variables for types (`~type:...`) and identifiers (`x_...`). For normal fields, the getter returns the user value. For default fields, it returns the user value if it is set, and the calculated value otherwise. For derived fields, it always returns the calculated value.

```
case is-normal-default; is-bidirectional; is-to-one; inverse-is-to-one: set := |[
  public void x_set(x_type other){
    if(x_name != null) x_name.x_inverseSetIncr(null);
    if(other != null){
      x_inverseType v = other.x_inverseName;
      if(v != null) v.x_setIncr(null);
      other.x_inverseSetIncr(this);
    }
    this.x_setIncr(other);
  }
]|
case is-normal-default; is-bidirectional; is-to-many; inverse-is-to-many: set:= |[
  public void x_set(Collection<x_type> others){
    Collection<x_type> toAdd = new HashSet<x_type>();
    toAdd.addAll(others); toAdd.removeAll(x_name);
    Collection<x_type> toRem = new HashSet<x_type>();
    toRem.addAll(x_name); toRem.removeAll(others);
    for(x_type n : toRem) n.x_inverseRemoveIncr(this);
    for(x_type n : toAdd) n.x_inverseAddIncr(this);
    x_setIncr(others);
  }
]|
```

Figure 17 Two cases from the `set()` Java code generation. The case for 1 to 1 relations removes previous references to both objects (`this` and `other`) and sets the references of both objects to each other. The case for n to n relations removes the references from previously related objects `toRem` to `this`, adds new references from `toAdd` to `this`, and updates the references of `this`.

```
case (is-left; is-normal-default; is-zeroormore-unordered)
   + (is-left; is-default; is-oneormore-unordered): ebd_field* := ebd* ⟦
  x_name : Set<srt_type> (inverse=x_inverseEntityName.x_inverseName)
⟧
case is-left; is-normal; is-oneormore-unordered: ebd_field* := ebd* ⟦
  x_name : Set<srt_type> (inverse=x_inverseEntityName.x_inverseName,
                    validate(x_get().length != 0, "" + e_name + " is required."))
⟧
```

Figure 18 Two of the twelve cases for `userField` WebDSL code generation. Types are specialized for $[\_, 1]$ to single values, for $[\_, n)$ *ordered* to `Lists`, and for $[\_, n)$ *unordered* to `Sets`. The left-hand side of relations specify `inverse`s. A validator checks the multiplicity lower-bound of 1 at runtime for `normal`-valued (not `default`-valued) fields.

```
fieldname-to-webdsl-entitybodydeclarations: x_name -> ebd_setIncr*
  x_set         := $[set[<ucfirst>x_name]];
  x_flagFlows   := $[flagFlows[<ucfirst>x_name]];
  srt_multType  := <type-and-mult-to-webdsl-srt>x_name;
  stat_flows*   := <flows;filter(where(expr-last;is-incr-even);to-webdsl)>x_name;
  switch id
    case is-normal-default; where(not([] := stat_flows*)): ebd_setIncr* := ebd*  ⟦
      extend function x_set(newValue : srt_multType){
        if(x_name != newValue){ x_flagFlows(); }
      }
    ⟧
    otherwise: ebd_setIncr* := []
  end
```

Figure 19 WebDSL `set`ter-hook code generation. If the field has any data-flow to an `incremental` or `eventual` field, generate a setter-hook that flags the cache dirty if the value changed.

The to-Java compiler supports specifying test data, and expressions for execution. This enables us to use IceDust2 as a glorified spreadsheet, and to write automated tests for IceDust2 specifications.

Compilation to WebDSL. The second compiler compiles IceDust2 to WebDSL, a domain-specific language for building web applications [32]. The to-WebDSL compiler features all IceDust2 features, including multiplicity lower-bound runtime checks, and the `eventual` calculation strategy. WebDSL differs from Java. WebDSL persists its data in a relational database and maps it to memory with an object relational mapper. The object-relational mapper provides transaction semantics. WebDSL already has a language feature for bidirectional relations, including the interaction with 'multiplicities' (single values or lists). This means the to-WebDSL compiler need not generate any code for that. However, this built-in support complicates the interaction with IceDust2 incrementality.

Figure 18 shows two cases of the code generator for fields. The WebDSL field code generation touches many IceDust2 features. Bidirectionality in WebDSL is defined by `inverse` annotations, which should be specified on one field of the relation. For a quality object-relational mapping, ordered fields are compiled to `Lists`, unordered fields are compiled to `Sets`, and single values to single values. Finally, the checks for multiplicity bounds should be specified on the field definitions as well. Together, three possible types, an optional inverse, and an optional validator make twelve possible field definitions.

For incremental updates, the to-WebDSL compiler generates incremental setters. To escape the bidirectionality abstraction, and get access to updates on both sides of the relation,

```
entity Conference {
  name             : String
  rootName         : String = root.name
  numComittees     : Int    = count(committees)
}
relation Conference.parent ? <-> * Conference.children
relation Conference.root 1 = parent.root <+ this <-> * Conference.rootDescendants

entity Person {
  name          : String
}
entity Profile {
  name          : String = person.name + " in " + conference.name
  numComittees  : Int    = count(committees)
}
relation Profile.conference 1 <-> * Conference.profiles
relation Profile.person     1 <-> * Person.profiles

entity Committee {
  name        : String
  fullName    : String = conference.name + " " + name
}
relation Committee.conference 1 <-> * Conference.committees
relation Committee.members    * <-> * Person.committees
relation Profile.committees   * =
  person.committees.filter(x => x.conference == this.conference)
                              <-> * Committee.profiles
```

Figure 20 Mini conference management system IceDust2 specification. A `Conference` can be a sub-conference of a `parent` conference. A `Person` has a separate `Profile` for each conference (s)he participates in. A conference is organized by multiple `Committee`s. A person can be `member` of committees in various conferences.

WebDSL provides `setter` hooks, similar to aspect-oriented pointcuts [19]. Figure 19 shows the implementation of the setter hook. These hooks only intercept calls, they do not update the fields. Thus, it cannot test for observable changes (by calling `get` before and after changing the field [SetIncr]). It approximates this by checking whether the value changes.

The to-WebDSL compiler is used in web applications. It enables specifying the business logic in derived values, and enables changing the calculation strategy of fields without much effort to tune the performance of web applications.

7 Case Studies

We discuss the application of IceDust2 to two representative applications, a conference management system, and an online learning management system (the running example).

Conference Management System. Figure 20 shows a mini version of a conference website management system. In this system multiple `Conference`s can be managed. A `Person` can be part of multiple conferences, and has a `Profile` for each. The conference system contains various derived values. For this paper, the most interesting ones are derived relations.

The mini system contains two derived relations. The first derived relation is the `root` of a conference tree (Figure 20, line 7). Conferences can have sub-conferences, and these can have sub-conferences again. For presentation purposes it is important to display the context of a sub-conference: the root conference. The inverse of the `root` field, `rootDescendants`,

```
entity Assignment { }
entity Submission {
  grade : Float? = groupSubmission.grade <+ children.grade.avg() (default)
}
entity Group { }
entity GroupSubmission {
  grade : Float?
}
relation Group.members               * <-> * Student.groups
relation Submission.assignment       1 <-> * Assignment.submissions
relation GroupSubmission.assignment  1 <-> * Assignment.groupSubmissions
relation GroupSubmission.group       1 <-> * Group.submissions
relation Submission.groupSubmission ? =
  assignment.groupSubmissions.find(x => x.group.members.contains(student))
                                       <-> * GroupSubmission.individualSubmissions
```

Figure 21 Learning management system specification for group submissions. If a student is part of a group that has submitted to a certain assignment, his individual grade will be taken from the group grade by default. The individual grade of a student can still be overridden by the instructor.

```
relation Submission.children * (ordered) =
  assignment.children.submissions.filter(x => x.student == student)
                                       <-> ? Submission.parent
relation Submission.next ? =
  parent.children.elemAt(parent.children.indexOf(this) + 1)
                                       <-> ? Submission.previous
```

Figure 22 Bidirectional relation `next` and `previous` is derived from the `children`'s ordering.

does not have a practical use in the application specification. However, it is used by the compiler to incrementally maintain `rootName` on name changes to the root conference. It is possible to omit the name `rootDescendants`. The IceDust2 compiler will then invent a name for the field itself (`rootInverse` in this case).

The second derived relation is the committees a person is a member of in a specific conference: `Profile.committees` (Figure 20, bottom). It is similar in structure to the submission parent-children relation in Figure 4. Both navigate the object graph to a collection of objects, and subsequently filter this collection. The committee membership derived relation is used bidirectionally: a committee page links to the profile pages of its members.

Learning Management System. Our running example (Figure 4) is a partial model of a learning management system, which we have specified in IceDust2. The production system is much more complicated. We will cover some interesting aspects of its specification.

Figure 21 shows a part of the specification that deals with group submissions. In some courses students get graded in groups. Moreover, in some labs the groups change during the semester. To calculate correct grades for individual students, their individual submissions are connected to the group submissions (`Submission.groupSubmission`). The student grade for a single assignment (`Submission.grade`) is the group grade, if it exists, and otherwise the normal individual student grade.

Figure 22 revisits the submission parent-child relation. We use the ordering of children to define `next` and `previous` for submissions, which are used for navigation in the user interface. Note that both of the derived bidirectional relations in Figure 22 have a multiplicity bound $[0, 1]$ on the right-hand side. This is disallowed by the IceDust2 compiler, as these bounds cannot be statically guaranteed. We will discuss this in the next section.

In our running example (Figure 4) we have used composition of calculation strategies to get good performance on changes to data, while always reading up-to-date student grades. In the full learning management system we have used the same approach: `incremental` for individual student data, and `eventual` for statistics. This approach works great with our to-WebDSL compiler. Often multiple students send changes to their submissions concurrently. These changes influence just their own grades. Incrementally updating the grades for single students is fine, as the cache updates will not overlap. However, course statistics cannot be updated incrementally in a concurrent setting, as the aggregated values would get update conflicts when multiple students concurrently get a new grade. In future work it might be worth investigating whether the calculation strategies can be automatically determined based on the partitioning of data between application users (students in this case).

In both case studies the orthogonal nature of the features for fields in IceDust2 turned out to be advantageous. Changing the derivation type, for example from a user value to a derived value, only requires adding or removing an expression. Changing the calculation strategy is a matter of changing a single keyword, and if any changes of calculation strategies in other fields are required for consistency, the type system will tell. Changing a multiplicity, for example making a field optional (`?`), rather than required, is a matter of changing a single character. Here as well, the type system will signal any places where semantic changes are required (for example the read of that field where a value with multiplicity of `1` is required). If these changes were to be made to a program expressed in a general purpose language, they would require all kinds of boilerplate changes, on top of the semantic changes. This has been argued before for multiplicities [29], bidirectional relation maintenance [16], and calculation strategy switching [15] individually. But combined, it is certainly true as well.

8 Multiplicity Bounds for the Right-Hand Side of Derived Relations

Derived bidirectional relations in IceDust2 specify multiplicity bounds both for the left-hand and right-hand side. The multiplicity bound on the left-hand side is checked by checking the multiplicity of the expression. The multiplicity bound on the right-hand side is only allowed to be $[0, n)$, as IceDust2 features no static checks for the right-hand side multiplicity bound.

We can view a bidirectional relation as a function, where the left-hand side is the domain and the right-hand side is the codomain. A derived relation is a total function (the expression can be executed for all objects in the domain), and each element in the domain maps to zero or more elements in the codomain (restricted to the multiplicity bound of the expression). To get guarantees for the right-hand side multiplicity bound, this function needs to satisfy certain properties. For a multiplicity upper-bound of 1, the function needs to be injective: at most one element in the domain will refer to to each element in the codomain. For a multiplicity lower-bound of 1, the function needs to be surjective: at least one element in the domain will refer to each element in the codomain. IceDust2's type system does not include reasoning about this. We can only safely assume the function is not injective and not surjective, and give the right-hand side a multiplicity bound of $[0, n)$.

However, our case studies revealed two useful derived bidirectional relations that would benefit from a more strict multiplicity bound on the right-hand side. Figure 22 shows them. If the inverses are actually within the specified multiplicity bound, the runtime works fine for these derived relations. Our type system rejects these derived relations, but the programmer can disable the error if he is confident the inverse is within the multiplicity bound.

Disabling the error is not sound, the programmer might be mistaken. If the programmer makes an error, IceDust2 cannot statically guarantee one of the following three properties:

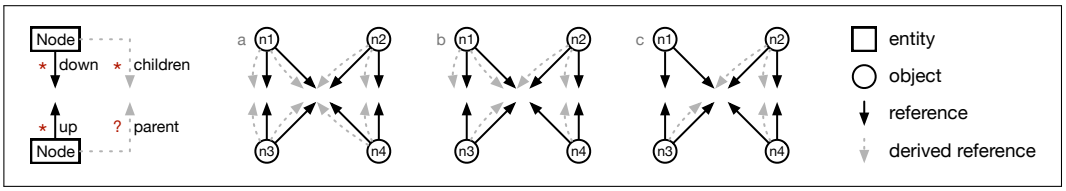

Figure 23 Contradictory specification solutions: (a) give up multiplicity bounds, (b) give up bidirectionality, or (3) give up derivation semantics.

multiplicity bounds, bidirectionality, or derivation semantics. Consider the following program:

```
entity Node { }
relation Node.down * <-> * Node.up
relation Node.children * = down <-> ? Node.parent
```

If some object refers to two other objects in `up`, it should have two `parents` as well, violating the multiplicity bound (Invariant 3). To satisfy the multiplicity bound, either bidirectionality (Invariant 2) or derivation semantics (Invariant 1) has to be given up. Figure 23 shows the three solutions by giving up one of the three invariants. This example implemented in REScala (in the same way we implemented submission parent-children in Section 2) does not preserve bidirectionality (Figure 23b). The parents of `n3` and `n4` would first be set to one of the objects `n1` and `n2`, and then to the other. The IceDust2 implementation gives up derivation semantics in this situation (Figure 23c). Either object `n1` or `n2` will not have any children, even though evaluating the derivation expression would yield children. We do not argue one is better than the other, both violate an invariant. In future work we will investigate creating a type system that rejects the above example, but accepts Figure 22.

In conclusion, in our case studies we only encountered this one example where a non-$[0, n)$ multiplicity on the right-hand side of a relation was required. The rest of the case studies could all be specified in a way that guarantees Invariants 1-3. If the programmer correctly specifies a right-hand side multiplicity, Invariants 1-3 are still guaranteed. Nonetheless, it is still worth to move the responsibility of checking the right-hand side multiplicities for derived relations from the programmer to the type system, in future work.

9 Related Work

The related work is organized along the lines of the various language features. We cover bidirectional relations, incremental and eventual computation, and the use of product lines in language engineering.

Derived Bidirectional Relations. Various languages feature bidirectional relations as a language feature. Rumer [3], RelJ [4], Relations [16], and IceDust [15] all feature bidirectional relations as language feature, but do not support derived bidirectional relations. They vary in multiplicity bound behavior: Rumer and RelJ enforce multiplicities at runtime, while Relations and IceDust feature multiplicities in the type system. IceDust2's behavior for maintaining multiplicity upper bounds is similar to RelJ's: it implicitly removes references.

Derived bidirectional relations can be described as views in relational and logic databases. They can be incrementalized by materializing the views [11]. Traditional algorithms for materialized views limit recursive aggregation [12]. Some forms of recursive aggregation can be incrementalized [26, 27], but until now the community has not converged to a recursive

aggregation technique [10]. LogiQL [9] has rudimentary support for recursive aggregation (behind a compiler flag). Most databases that feature materialized views also feature non-materialized views, enabling composition of incremental and on-demand calculation strategies. Database languages do not allow specification of multiplicity bounds, thus all derived values have a multiplicity of $[0, n)$. IceDust2 does feature multiplicity constraints, includes an eventual calculation strategy, and admits recursive aggregation.

i3QL [24], Materialized Object Query Language (OQL) [8], and MOVIE [2] support materialized views in object-oriented languages. The data is in memory, rather than persisted on disk. Strategy composition can be done by using the framework for incremental derived values, and the host language for on-demand derived values. As these systems are relational, they have the same limitations as databases: no multiplicity bounds, no eventual calculation strategy, and limited support for recursion (except for i3QL, it features fixpoint recursion).

IncQuery [31] features incremental graph queries. These can be scheduled by a query planner, but provide no multiplicity bounds. In IceDust2 derived relations are specified as expressions, which provides a multiplicity bound for the left-hand side of the derived relation. For derived primitive values IncQuery has an escape hatch to Java. This makes it Turing complete, but only the dependencies and results are cached, not the internal computation. On the other hand, IceDust2 is not Turing complete (its memory footprint is bounded by the total number of fields of all objects), but the full computation is incrementalized.

Alloy [17] (with operational semantics Alchemy [20]) and Booster [5] feature bidirectional derived relations as well. These systems use constraints for describing derived values and multiplicity bounds. On changes to fields, other fields are updated to maintain the constraints. In constraints, all field references can function as inputs and outputs, so for predictability, only values mentioned in update operations are updated. In contrast, IceDust2 can predictably update any value, as it uses expressions for derived values, not constrains. The fields referenced in an expression are input, the field the expression is for, is output.

Incremental Computation without Bidirectional Relations. Various programming styles and languages that can be used for incremental computation do not support derived bidirectional relations. These can only be used for derived unidirectional relations.

Functional reactive programming (FRP) [7], with for example REScala [28], or Scala.React [22] can be used for incremental computation. Wrapping expressions in signal macros realizes incremental behavior, reevaluating the expression when one of its dependencies is changed. FRP maintains dependencies at runtime, causing memory overhead. In contrast, IceDust2 uses static dependency information. However, FRP frameworks do support any language feature as long as it is pure, while IceDust2 restricts its expression language to be able to statically analyze its dependencies. FRP allows strategy composition by modeling incremental derived values in FRP, and using the host language for on-demand derived values. However, the safety of compositions is not checked, and can result in inconsistencies.

Self-adjusting computation (SAC) [1] and Adapton [14] also use dependency tracking for incremental computation. Adapton features a demand-driven incremental calculation strategy: dirty flag transitively on writes, and recompute transitively on reads if dirty. IceDust2 features on-demand, incremental, and eventual calculation strategies. We might add Adapton's calculation strategy to IceDust2 in future work, it would fit in the general IceDust2 approach without requiring invasive changes to the architecture. Adapton works only on algebraic data types, but Nominal Adapton [13] is better suited for object graphs, it allows identifying caches. In Nominal Adapton's terms, the derived value caches in IceDust2's runtime can be identified 'objectIdentifier+fieldName'. Adapton allows strategy composition

by modeling incremental derived values in Adapton, and the on-demand derived values in the host language. The safetey of strategy composition of is checked in Adapton. Adapton does not feature eventual calculation, bidirectional relations, or data persistence.

Incremental Java Query Language (JQL) [34], and Demand-Driven Incremental Object Queries (DDIOQ) [21] enable specifying derived values as queries in Java. They transform imperative code to a relational calculus, and use the relational model to generate code that incrementally maintains caches. In contrast, IceDust2 uses path-based abstract interpretation instead of a relational calculus to generate maintenance code.

Attribute grammars (AGs) feature a declarative style of specifying derived primitive values similar to IceDust. Attribute values can also be incrementally computed [6]. Reference attribute grammars (RAGs) support derived relations [30]. RAGs only support trees as input (graphs can only be derived values), while IceDust2 works with graphs. As AGs and RAGs are designed for use in compilers they do not feature an eventual calculation strategy.

Eventual Calculation without Bidirectional Relations. Reactive programming (RP), with for example RX [23], features a programming model similar to FRP. However, RP provides an eventual instead of an incremental calculation strategy by asynchronously processing updates. RP enables composition with eventual and on-demand calculation strategies by using the host language for on-demand calculation. Note that on-demand calculation is `eventual on-demand` if it depends on eventual calculation, as in our approach (see Figure 14).

Software Product Lines and Language Engineering. Völter and Visser have investigated the combination of software product lines (SPLs) and domain-specific languages (DSLs) [33]. In their taxonomy, IceDust2 falls in the category *'Variations in the Transformation or Execution'*. The IceDust2 operational semantics vary in execution, and the IceDust2 compilers vary in transformation based on the field properties. Behavior is chosen based on presence conditions. IceDust2 falls in the sub category *'Negative Variablility via Removal'* by only retaining the behavior satisfying the presence conditions out of all possible behaviors.

The Dana language [25] enables switching features at run time. In order to be able to switch at run time, the various options for a feature need to have the same public API, and they need to share a set of transfer fields. Unfortunately, this is not possible with the IceDust2 runtime, as the public API varies based on the features selected. We would like to investigate switching calculation strategies at runtime in future work.

10 Summary and Future Work

In this paper we have presented IceDust2, a declarative data modeling language that supports composition of derivation calculation strategies and bidirectional derived relations with multiplicity bounds. Because updating derived values with various strategies, maintaining bidirectionality, and keeping multiplicity bounds all interact, the IceDust2 semantics for individual fields is structured as a product line, which can be instantiated in two compilers. One that compiles to plain old Java objects, and one that compiles to an object-relational mapper. Finally, our case studies validated the usability of IceDust2 in applications: derived values can be specified declaratively and concisely, independent of their complex runtime.

This work also raises open research questions. First, is it possible to provide static guarantees for multiplicity bounds for the right-hand side of derived bidirectional relations? Second, what calculation strategies can be added to IceDust2, and (more importantly) how can these strategies be composed in a sound way? Finally, is it possible to automatically assign

calculation strategies to derived values based on high level directives, such as partitioning data between application users?

References

1. Umut A. Acar. Self-adjusting computation: (an overview). In Germán Puebla and Germán Vidal, editors, *Proceedings of the 2009 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2009, Savannah, GA, USA, January 19-20, 2009*, pages 1–6. ACM, 2009. doi:10.1145/1480945.1480946.
2. M. Akhtar Ali, Alvaro A. A. Fernandes, and Norman W. Paton. Movie: An incremental maintenance system for materialized object views. *Data & Knowledge Engineering*, 47(2):131–166, 2003. doi:10.1016/S0169-023X(03)00048-X.
3. Stephanie Balzer. *Rumer: a Programming Language and Modular Verification Technique Based on Relationships*. PhD thesis, ETH, Zürich, 2011. doi:10.3929/ethz-a-007086593.
4. Gavin M. Bierman and Alisdair Wren. First-class relationships in an object-oriented language. In Andrew P. Black, editor, *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*, volume 3586 of *Lecture Notes in Computer Science*, pages 262–286. Springer, 2005. doi:10.1007/11531142_12.
5. Jim Davies, James Welch, Alessandra Cavarra, and Edward Crichton. On the generation of object databases using booster. In *11th International Conference on Engineering of Complex Computer Systems (ICECCS 2006), 15-17 August 2006, Stanford, California, USA*, pages 249–258. IEEE Computer Society, 2006. doi:10.1109/ICECCS.2006.1690374.
6. Alan J. Demers, Thomas W. Reps, and Tim Teitelbaum. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Williamsburg, Virginia, January 26-28, 1981*, pages 105–116, 1981. doi:10.1145/567532.567544.
7. Conal M. Elliott. Push-pull functional reactive programming. In Stephanie Weirich, editor, *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, Haskell 2009, Edinburgh, Scotland, UK, 3 September 2009*, pages 25–36. ACM, 2009. doi:10.1145/1596638.1596643.
8. Dieter Gluche, Torsten Grust, Christof Mainberger, and Marc H. Scholl. Incremental updates for materialized oql views. In François Bry, Raghu Ramakrishnan, and Kotagiri Ramamohanarao, editors, *Deductive and Object-Oriented Databases, 5th International Conference, DOOD 97, Montreux, Switzerland, December 8-12, 1997, Proceedings*, volume 1341 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 1997. doi:10.1007/3-540-63792-3_8.
9. Todd J. Green. Logiql: A declarative language for enterprise applications. In Tova Milo and Diego Calvanese, editors, *Proceedings of the 34th ACM Symposium on Principles of Database Systems, PODS 2015, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 59–64. ACM, 2015. doi:10.1145/2745754.2745780.
10. Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. Datalog and recursive query processing. *Foundations and Trends in Databases*, 5(2):105–195, 2013. doi:10.1561/1900000017.
11. Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.
12. Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, pages 157–166. ACM Press, 1993. doi:10.1145/170035.170066.
13. Matthew A. Hammer, Joshua Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael W. Hicks, and David Van Horn. Incremental computation with names. In Jonathan

Aldrich and Patrick Eugster, editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 748–766. ACM, 2015. doi:10.1145/2814270.2814305.

14 Matthew A. Hammer, Yit Phang Khoo, Michael Hicks, and Jeffrey S. Foster. Adapton: composable, demand-driven incremental computation. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 18. ACM, 2014. doi:10.1145/2594291.2594324.

15 Daco Harkes, Danny M. Groenewegen, and Eelco Visser. Icedust: Incremental and eventual computation of derived values in persistent object graphs. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.ECOOP.2016.11.

16 Daco Harkes and Eelco Visser. Unifying and generalizing relations in role-based data modeling and navigation. In Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju, editors, *Software Language Engineering - 7th International Conference, SLE 2014, Västeras, Sweden, September 15-16, 2014. Proceedings*, volume 8706 of *Lecture Notes in Computer Science*, pages 241–260. Springer, 2014. doi:10.1007/978-3-319-11245-9_14.

17 Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering Methodology*, 11(2):256–290, 2002. doi:10.1145/505145.505149.

18 Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, DTIC Document, 1990.

19 Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP 97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9-13, 1997, Proceedings*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997. doi:10.1007/BFb0053381.

20 Shriram Krishnamurthi, Kathi Fisler, Daniel J. Dougherty, and Daniel Yoo. Alchemy: transmuting base alloy specifications into implementations. In Mary Jean Harrold and Gail C. Murphy, editors, *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008*, pages 158–169. ACM, 2008. doi:10.1145/1453101.1453123.

21 Yanhong A. Liu, Jon Brandvein, Scott D. Stoller, and Bo Lin. Demand-driven incremental object queries. In James Cheney and Germán Vidal, editors, *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016*, pages 228–241. ACM, 2016. doi:10.1145/2967973.2968610.

22 Ingo Maier and Martin Odersky. Higher-order reactive programming with incremental lists. In Giuseppe Castagna, editor, *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*, volume 7920 of *Lecture Notes in Computer Science*, pages 707–731. Springer, 2013. doi:10.1007/978-3-642-39038-8_29.

23 Erik Meijer. Reactive extensions (rx): curing your asynchronous programming blues. In *ACM SIGPLAN Commercial Users of Functional Programming*, page 11. ACM, 2010. doi:10.1145/1900160.1900173.

24 Ralf Mitschke, Sebastian Erdweg, Mirko Köhler, Mira Mezini, and Guido Salvaneschi. i3ql: language-integrated live data views. In Andrew P. Black and Todd D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming*

Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014, pages 417–432. ACM, 2014. doi:10.1145/2660193.2660242.

25 Barry Porter, Matthew Grieves, Roberto Vito Rodrigues Filho, and David Leslie. Rex: A development platform and online learning approach for runtime emergent software systems. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 333–348. USENIX Association, 2016.

26 Raghu Ramakrishnan, Kenneth A. Ross, Divesh Srivastava, and S. Sudarshan. Efficient incremental evaluation of queries with aggregation. In *Workshop on Design and Impl. of Parallel Logic Programming Systems*, pages 204–218, 1994.

27 Kenneth A. Ross and Yehoshua Sagiv. Monotonic aggregation in deductive databases. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 2-4, 1992, San Diego, California*, pages 114–126. ACM Press, 1992. doi:10.1145/137097.137852.

28 Guido Salvaneschi, Gerold Hintz, and Mira Mezini. Rescala: bridging between object-oriented and functional style in reactive applications. In Walter Binder, Erik Ernst, Achille Peternier, and Robert Hirschfeld, editors, *13th International Conference on Modularity, MODULARITY '14, Lugano, Switzerland, April 22-26, 2014*, pages 25–36. ACM, 2014. doi:10.1145/2577080.2577083.

29 Friedrich Steimann. Content over container: object-oriented programming with multiplicities. In Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld, editors, *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, pages 173–186. ACM, 2013. doi:10.1145/2509578.2509582.

30 Emma Söderberg and Görel Hedin. Incremental evaluation of reference attribute grammars using dynamic dependency tracking. Technical Report 98, Department of Computer Science, Lund University, 2012.

31 Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. Emf-incquery: An integrated development environment for live model queries. *Science of Computer Programming*, 98:80–99, 2015. doi:10.1016/j.scico.2014.01.004.

32 Eelco Visser. WebDSL: A case study in domain-specific language engineering. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007*, volume 5235 of *Lecture Notes in Computer Science*, pages 291–373, Braga, Portugal, 2007. Springer. doi:10.1007/978-3-540-88643-3_7.

33 Markus Völter and Eelco Visser. Product line engineering using domain-specific languages. In Eduardo Santana de Almeida, Tomoji Kishi, Christa Schwanninger, Isabel John, and Klaus Schmid, editors, *Software Product Lines - 15th International Conference, SPLC 2011, Munich, Germany, August 22-26, 2011*, pages 70–79. IEEE, 2011. doi:10.1109/SPLC.2011.25.

34 Darren Willis, David J. Pearce, and James Noble. Caching and incrementalisation in the java query language. In Gail E. Harris, editor, *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, pages 1–18. ACM, 2008. doi:10.1145/1449764.1449766.