Declarative Specification of Information System Data Models and Business Logic

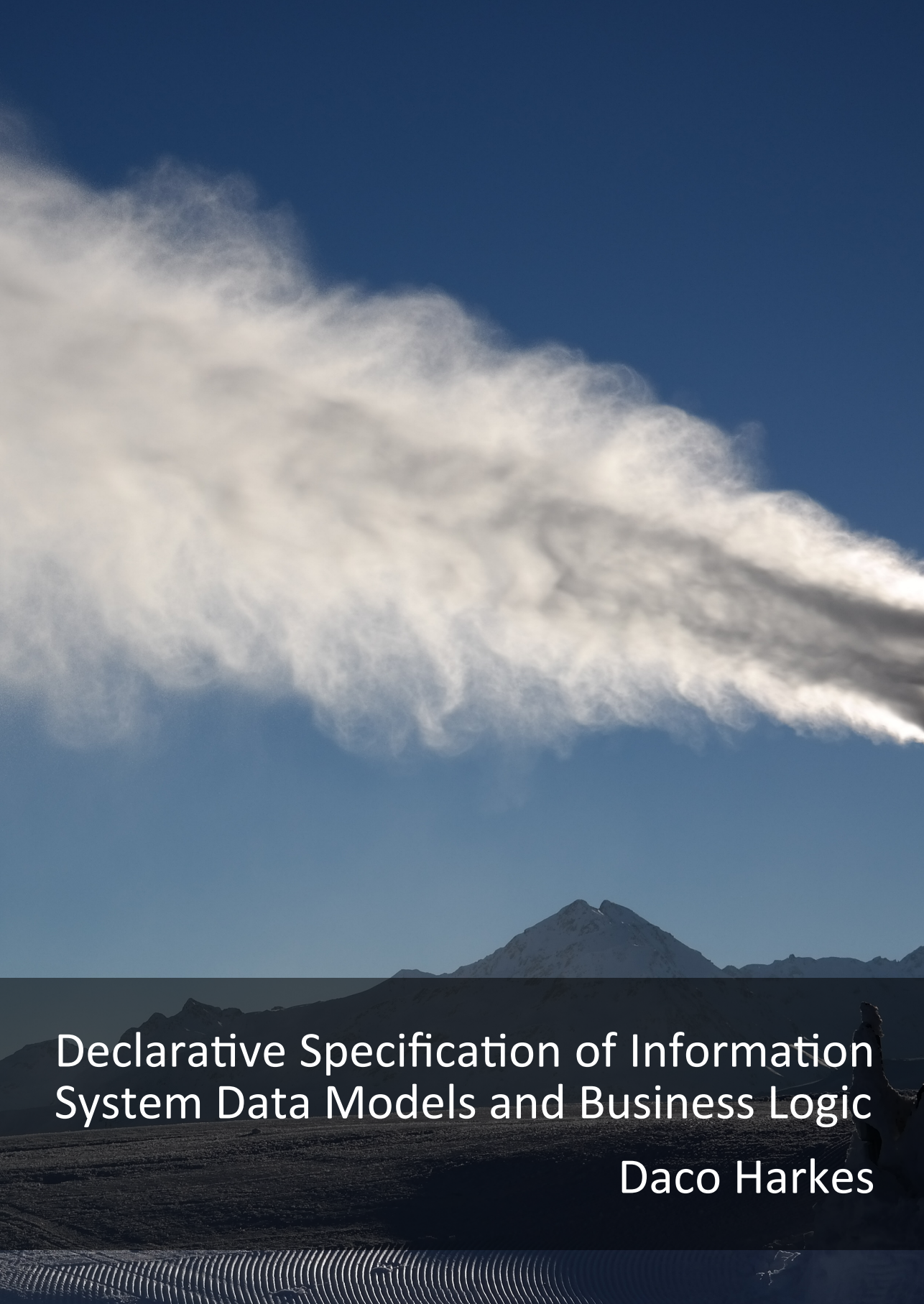
Harkes, Daco

DOI
10.4233/uuid:5e9805ca-95d0-451e-a8f0-55decb26c94a

Publication date
2019

Document Version
Final published version

Citation (APA)
Harkes, D. (2019). *Declarative Specification of Information System Data Models and Business Logic*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:5e9805ca-95d0-451e-a8f0-55decb26c94a



Declarative Specification of Information System Data Models and Business Logic

Daco Harkes

Declarative Specification of Information System Data Models and Business Logic

DISSERTATION

for the purpose of obtaining the degree of doctor
at Delft University of Technology
by the authority of the Rector Magnificus prof. dr. ir. T.H.J.J. van der Hagen
chair of the Board for Doctorates
to be defended publicly on
Tuesday 26 March 2019 at 15:00 o'clock
by

Daniël Corstiaan HARKES

Master of Science in Computer Science,
Delft University of Technology, the Netherlands
born in Waddinxveen, the Netherlands

This dissertation has been approved by the promotor.

Composition of the doctoral committee:

Rector Magnificus	chairperson
Prof. dr. E. Visser	Delft University of Technology, promotor

Independent members:

Prof. dr. A. van Deursen	Delft University of Technology
Prof. dr. ir. G.J.P.M. Houben	Delft University of Technology
Dr. ir. F.F.J. Hermans	Leiden University
Prof. dr. G. Hedin	Lund University
Prof. dr. F. Steimann	FernUni Hagen
Prof. dr. G. Salvaneschi	Technische Universität Darmstadt

The work in this thesis has been carried out at the Delft University of Technology. This research was funded by the NWO VICI *Language Designer's Workbench project* (639.023.206).

Printed by ProefschriftMaken (www.proefschriftmaken.nl)

ISBN 978-94-6366-146-1

DOI 10.4233/uuid:5e9805ca-95d0-451e-a8f0-55decb26c94a

Contents

Samenvatting

Informatiesystemen zijn systemen voor het verzamelen, organiseren, opslaan en communiceren van informatie. Deze systemen zijn gericht op het ondersteunen van activiteiten, management en besluitvorming. Voor deze ondersteuning, filteren en verwerken deze systemen gegevens, hetgeen resulteert in nieuwe gegevens. Typisch bevatten deze informatiesystemen grote hoeveelheden gegevens en worden deze gegevens frequent gewijzigd. In de loop van de tijd veranderen de eisen voor informatiesystemen, van de verwerkingslogica tot het aantal gebruikers dat met het systeem communiceert. Kortom, wanneer organisaties veranderen, moeten informatiesystemen mee veranderen.

Onze afhankelijkheid van informatiesystemen om beslissingen te nemen en de steeds veranderende eisen creëren de volgende uitdagingen voor het ontwikkelen van informatiesystemen. *Valideerbaarheid:* hoe gemakkelijk is het voor ontwikkelaars van informatiesystemen om vast te stellen dat een systeem 'doet wat het moet doen'? *Traceerbaarheid:* kan de oorzaak van door het systeem genomen beslissingen worden gecontroleerd? *Betrouwbaarheid:* kunnen we erop vertrouwen dat het systeem consequent beslissingen neemt en onze gegevens niet verliest? *Prestaties:* kan het systeem prompt reageren op gebruikers? *Beschikbaarheid:* kunnen we erop vertrouwen dat het systeem de functionaliteit altijd uitvoert? En tot slot, *veranderbaarheid:* hoe gemakkelijk is het om de systeemspecificatie te veranderen wanneer de eisen veranderen?

In dit proefschrift tonen we de haalbaarheid en het nut van declaratief programmeren voor informatiesystemen aan in het licht van deze uitdagingen.

Onze onderzoeksmethode is ontwerponderzoek. Deze iteratieve methode heeft vier fasen: analyse, ontwerp, evaluatie en verspreiding. We *analyseren* de uitdagingen van het ontwikkelen van informatiesystemen, *ontwerpen* een nieuwe programmeertaal om deze uitdagingen aan te pakken, *evalueren* onze nieuwe programmeertaal in de praktijk, en *verspreiden* onze kennis door het publiceren van wetenschappelijke artikelen. Dit heeft geresulteerd in vier nieuwe declaratieve talen: de Relaties taal, IceDust, IceDust2 en PixieDust.

Onze contributies kunnen worden samengevat door de nieuwe onderdelen van deze talen. *Taaleigen multipliciteiten, bidirectionele relaties en beknopte navigatie* verbeteren de valideerbaarheid en modificeerbaarheid van informatiesystemen ten opzichte van objectgeoriënteerde en relationele benaderingen. *Afgeleide attribuut waarden* verbeteren de traceerbaarheid. *Incrementele en uiteindelijke berekeningen* op basis van paden analyse en *het omschakelen van berekeningsstrategieën* verbetert de modificeerbaarheid van informatiesystemen zonder in te boeten op prestaties en beschikbaarheid ten opzichte van objectgeoriënteerde en relationele benaderingen. *Compositie van berekeningsstrategieën* verbetert de valideerbaarheid, aanpasbaarheid en betrouwbaarheid ten opzichte van reactieve programmeertechnieken. En ten slotte verbeteren *af-*

geleide bidirectionele relaties de valideerbaarheid van informatiesystemen ten opzichte van relationele benaderingen.

De resultaten van dit proefschrift kunnen in de praktijk worden toegepast. We hebben IceDust2 toegepast op het e-learning informatiesysteem WebLab. Dit heeft de valideerbaarheid, traceerbaarheid, betrouwbaarheid en modificeerbaarheid aanzienlijk verbeterd terwijl de prestaties en beschikbaarheid behouden zijn gebleven. Bovendien suggereert het feit dat IceDust en PixieDust in verschillende domeinen gebruikt worden, verwerkingslogica en gebruikersinterfaces respectievelijk, dat onze nieuwe taal onderdelen op meer domeinen kunnen worden toegepast.

Summary

Information systems are systems for the collection, organization, storage, and communication of information. Information systems aim to support operations, management and decision-making. In order to do this, these systems filter and process data according to business logic to create new data. Typically these information systems contain large amounts of data and receive frequent updates to this data. Over time requirements for information systems change, from the decision making logic to the number of users interacting with the system. As organizations evolve, so must their information systems.

Our reliance on information systems to make decisions and the ever changing requirements poses the following challenges for information system engineering. *Validatability:* how easy is it for information system developers to establish that a system 'does the right thing'? *Traceability:* can the origin of decisions made by the system be verified? *Reliability:* can we trust the system to consistently make decisions and not lose our data? *Performance:* can the system keep responding promptly to the load of its users? *Availability:* can we trust that the system performs its functionality all of the time? And finally, *modifiability:* how easy is it to change the system specification when requirements change?

In this dissertation we show the feasibility and usefulness of declarative programming for information systems in light of these challenges.

Our research method is *design research*. This iterative method repeats four phases: analysis, design, evaluation, and diffusion. We *analyze* the challenges of information system engineering, *design* a new programming language to address these, *evaluate* our new programming language in practice, and *diffuse* our knowledge through scholarly articles. This resulted in four new declarative languages: the Relations language, IceDust, IceDust2, and PixieDust.

Our contributions can be summarized by the new features of these languages. *Native multiplicities, bidirectional relations, and concise navigation* improve information system validatability and modifiability over object-oriented and relational approaches. *Derived attribute values* improve traceability. *Incremental and eventual computing* based on path analysis and *calculation strategy switching* improve information system modifiability without sacrificing performance and availability over object-oriented and relational approaches. *Calculation strategy composition* improves validatability, modifiability, and reliability over reactive programming approaches. And finally, *Bidirectional derived relations* improve information system validatability over relational approaches.

The results of this dissertation can be applied in practice. We applied IceDust2 to the learning management information system WebLab. We found that validatability, traceability, reliability, and modifiability were considerably improved while retaining similar performance and availability. Moreover, the fact that IceDust and PixieDust work in different domains, business logic and

user interfaces respectively, suggests that our language features could be applied to more domains.

Preface

When I started my PhD, I thought research was about adding knowledge to the world. I wanted to explore and solve programming language issues which fascinate me, kind of in the same way a Rubik's cube fascinates me. During my PhD, I quickly learned that research is about *relevant* knowledge, and that this knowledge needs to be sold. Luckily, I turned out to be good at communicating my research.

Dear reader, what you hold in your hands is a balance between that what fascinates me, and that what can be pitched to the scientific community. I hope that you enjoy it, and that you will be fascinated as well.

ACKNOWLEDGEMENTS

It is the glory of God to conceal things, but the glory of kings is to search things out.[1] Rubik (maybe unintentionally) concealed many mathematical properties in his cube, and people have come up with many algorithms, invariants, and proofs for these properties. Likewise, God concealed many mathematical properties in this universe, and I consider it an honor to figure these out. In our scientific community it gives us glory (prestige) when we figure things out, but I want to give glory to God for concealing these things in the first place.

I would like to thank my promoter and adviser Eelco Visser. He gave me the possibility to pursue a research direction which fascinates me. Moreover, he tirelessly showed me how to write good scientific literature.

I would like to thank all my colleagues for all good conversations about my research, their research, the process of research, and the emotions one has to deal with during research. I want to thank a few colleagues in particular. I would like to thank Guido Wachsmuth for getting me interested in compilers. I would like to thank Danny Groenewegen for his help with targeting IceDust to WebDSL, his help with setting up benchmarks for IceDust applications, and his advice on the social aspects of the supervisor-student relation. I would like to thank Gabriël Konat for keeping Spoofax running, especially for setting up continuous integration for early feedback on regressions, which saved me a lot of time. I would like to thank Hendrik van Antwerpen for being very helpful with keeping the IceDust static analysis up to date while NaBL2 evolved, the interesting discussions about incremental computing, and the discussions about the social and political aspects of the scientific community. I would like to thank Eduardo Souza for gradually improving the usability of the IceDust editor by developing SDF3, and the many positive conversations. I would like to thank Sebastian Erdweg for the discussions about incremental computing in order to try to get to its essence. I would like to thank Robbert

[1] Proverbs 25:2, Bible

Krebbers for teaching me how to prove properties about my language in a proof assistant. Finally, I would like to thank Elmer van Chastelet for his help with the WebLab case study, and Nick ten Veen for his implementation of PixieDust.

I would like to thank my brothers and sisters in church. I deeply value our joint journey and our conversations about life, the universe, and everything. Especially the elder people in our community, thank you for your wisdom and kind words. Thank you for caring about me. Your advice helped me to get through emotional hurdles that came as part of the PhD.

Also, I would like to thank my friends and family. Especially you helped me to continue my research in the face of frustration and insecurity. I would like to thank a few friends and my family in particular. Richard, thank you for your kinds words and your always interesting industry view on programming. Robin, thank you for our joint analysis of emotions and behavior, and for all the Belgian beers we shared. Paul, thank you for your enthusiasm about my research topic and your challenging questions. Gerben, thank you for your enthusiasm and hours of discussion. Klaas, I always enjoy your joy, and your perspectives on our society. Lianne, thank you for your listening ear. Marijke, thank you for the small presents. Joel, thank you for being such a joyful and sociable roommate. My brothers Joel and Theo, thank you for all our conversations about computer science, and the countless hours spent with video games. My sister Edith, thank you for being proud of me. And my parents, thank you for the insightful conversations about life and supporting me in the practical matters. Without you all, I would not have gotten a PhD.

Daco Harkes
May 14, 2018
Delft

Introduction

1

My thesis is that declarative specification of information system data models and business logic is feasible and useful.

In this introductory chapter, we explain this thesis. We cover information systems, and outline challenges in information system engineering. We describe the research context: declarative programming with domain-specific languages and incremental computing. We summarize our contributions, which show the feasibility and usefulness of declarative programming for information systems, and we finish with our research method, explaining why our results can be trusted.

1.1 INFORMATION SYSTEM ENGINEERING

Information systems are systems for the collection, organization, storage, and communication of information. Information systems aim to support operations, management and decision-making. In order to do this, these systems filter and process data according to business logic to create new data: derived data. Typically these information systems contain large amounts of data and receive frequent updates to this data.

Information systems are sociotechnical in nature: they are comprised of people, information and communications technology, organizational concepts (structures, processes), and the interrelationships between them [Österle et al., 2011]. As organizations evolve, so must their information systems. Thus, over time, requirements for information systems change, from the decision making logic to the number of users interacting with the system.

Our society's reliance on information systems to make decisions and the ever changing requirements poses several challenges for information system engineering. The people and organizations involved in creating and using information systems require a variety of properties of these systems. These properties concern both the running system that users interact with, as well as the specification that developers work on. Here we outline these concerns and the challenges to attain them.

Validatability

The task of information system developers is to translate user requirements to code. Bridging the gap between domain concepts and the encoding of these concepts in a programming language is one of the core challenges of software engineering [Visser, 2015]. We define the validatability of a program as a measure of the size of this gap. If one can express intent with relatively little encoding, it is straightforward to establish that a program 'does the right

```
CREATE VIEW assignmentPassAll AS
SELECT assignmentId AS id, BIT_AND(pass) AS passAll
FROM answer GROUP BY assignmentId

UNION

SELECT id, TRUE AS passAll
FROM assignment WHERE NOT EXISTS(
  SELECT NULL
  FROM answer
  WHERE answer.assignmentId = assignment.id
);
```

Figure 1.1 Bad validatability in SQL code due to a pattern for dealing with default values in aggregations. The first three lines of code express the desired intent: checking whether all answers to a question are correct. The remaining code deals with the edge case: no answers yet to a question.

```
class Assignment {
  def passAll(): Boolean = answers.forall(a => a.pass)
}
```

Figure 1.2 Better validatability in Scala code due to having a default value for forall.

thing'. If one needs to encode intent in patterns, then these patterns are an obstacle to understanding of programs by human readers [Felleisen, 1990] and make it harder to establish that a program does the right thing. Validatability decreases with increasing encoding. Information system developers want to assure the users that the system does the right thing, thus the information system specification should have good validatability.

An example of bad validatability is the pattern in SQL for dealing with default values for aggregations over empty lists (Figure 1.1). The source of this pattern is the gap between the user domain, in which all assignments should have a calculated `passAll`, and SQLs `join` and `group by` semantics, which omits assignments when they do not have answers. The first three lines of code in Figure 1.1 express the desired intent, and the rest has to deal with the edge case. On the other hand, expressing the desired functionality in a functional or object-oriented language does not have this issue (Figure 1.2).

Traceability

People make decisions based on information systems, or even let information systems make decisions for them. Thus it is important that users have the ability to verify the origin of decisions made by the system. This traceability concerns both the business logic making decisions, as well as the data that is used by this business logic. When details of specifications become scattered, traceability tends to suffer [Walker and Viggers, 2004]. Only when users can trust the decisions in the system, it is a useful tool in their organization.

An example of bad traceability is not being able to verify the origin of a computed value in an object-oriented language. In object-oriented languages, a field of an object might be assigned to from multiple locations in the code,

```
class Assignment {
  private float grade;

  public void someMethod(){
    if(someCondition)
      grade = someValue;
  }

  public void someOtherMethod(float parameter){
    grade = parameter;
  }
}
```

Figure 1.3 Bad traceability in Java code due to arbitrary state modifications. When `grade` has a particular value, it is hard to verify how that value was computed. First, `grade` can be assigned to from both methods. Second, an assignment might have been executed or not based on `someCondition`, and finally, temporary state such as method parameters might have contributed to the value.

and these locations might have been executed or not (Figure 1.3). Moreover, temporary state might have contributed to the value (second method in Figure 1.3), making it even harder to reconstruct how a value was computed. On the other hand, spreadsheet programs have good traceability, values are always computed by a single formula, and one can easily inspect this formula. Likewise, many declarative languages have built-in support for traceability [Jouault, 2005].

Reliability

Reliability concerns the risk of failure in information systems. An information system should not lose or corrupt data, even in the case of power outage or hardware failure [Hadzilacos, 1988]. Moreover, if an information system makes decisions, these decisions should be consistent with the data [Bharati and Chaudhury, 2004]. If the risk of failure is very small, users can trust the information system, and it will be useful for them. Moreover, the developers share this concern as they are responsible for the system. The challenge for developers is to ensure that their code and the underlying technologies satisfy these properties.

An example of bad reliability is manually trying to guarantee that user data is preserved by arbitrary application code (Figure 1.4). A contract, such as only allowing modification from user values from the user interface (by users), cannot be enforced in a general purpose programming language. Thus, guaranteeing that user values are not corrupted by arbitrary code requires manual code inspection. On the other hand, views in databases cannot corrupt the data in other tables.

Performance

The amount of data in information systems and the amount of concurrent users of these systems tends to grow over time. Moreover, the interaction behavior of users might change over time. This raises performance concerns

```
class Person {
  private String name;

  public void setName(String n){
    name = n;
  }
}

class PersonUI {
  private Person person;

  private save(){
    person.setName(userInterface.getName());
  }
}

class RandomClass {
  public randomMethod(Person p){
    p.setName("Random Data");
  }
}
```

Figure 1.4 It is hard to guarantee reliability in a general purpose language due to not being able to enforce contracts. Java cannot enforce that `Person.name` is only modified in the user interface and not in a `randomMethod`.

for information systems. If an information system is slow, its usefulness for users diminishes. Thus it is important that it keeps performing as the amount of data grows, the amount of users grows, and its workload changes. However, realizing a high performance implementation typically requires invasive changes to a basic expression of intent. Avoiding errors in high performance code is a daunting task for developers, especially if the information system requirements change continuously.

An example of a bad performance solution is manually keeping caches up to date for computed values (Figure 1.5). Developers have to make sure that changes to all different pieces of data that influence a computed value update the cache of that computed value. In Figure 1.5 this is both the relation between assignments and questions, and the progress on individual questions. Moreover, this code is only correct if the bidirectional relation between assignments and questions is kept up to date on changes. On the other hand, materialized relational views [Gupta and Mumick, 1995], and reactive programming languages such as REScala [Salvaneschi et al., 2014] do not have this issue. These technologies make cache updates error-free by construction.

Availability

Users expect information systems to be available and functioning at all times. As people and organizations schedule their activities, they need to be able to rely on the information system being available on the designated times. The information should stay available, especially when many people use the system at the same time, or when the system does internal tasks concurrently to user activity. Only when information systems have high availability, people

```
public class Assignment {
  private Double cachedAvgProgress;
  public Double getAverageProgress() {  return cachedAvgProgress;  }
  public Double calculateAverageProgress() {
    Stream<Double> progresss =
      questions.stream().map(q->q.getProgress()).filter(p -> p!=null);
    OptionalDouble average = progresss.mapToDouble(p -> p).average();
    return average.isPresent() ? average.getAsDouble() : null;
  }
  private Collection<Question> questions;
  public Collection<Question> getQuestions(){  return questions;  }
  public void addQuestion(Question q) {     q.setAssignment(this);  }
  public void removeQuestion(Question q) {  q.setAssignment(null);  }
  protected void _addQ(Question q){
    questions.add(q);    updateAvgProgress();
  }
  protected void _remQ(Question q){
    questions.remove(q); updateAvgProgress();
  }
  public void updateAvgProgress(){
    cachedAvgProgress=calculateAverageProgress();
  }
}
public class Question {
  private Assignment assignment;
  public Assignment getAssignment() {  return assignment;  }
  public void setAssignment(Assignment a) {
    if(assignment != null) { assignment._remQ(this); }
    if(a != null) { a._addQ(this); }
    assignment = a;
  }
  private Double progress;
  public Double getProgress() {  return progress;  }
  public void setProgress(Double p){
    progress=p; assignment.updateAvgProgress();
  }
}
```

Figure 1.5 Hard to guarantee error-free performance: is this code for caching and cache invalidation of `averageProgress` correct? Developers have to make sure that all changes that influence `averageProgress` also update its cache. This includes both `Assignment.questions` and `Question.progress`. Moreover, this code is only correct if `Assignment.questions` and `Question.assignment` are kept consistent with each other.

and organizations can depend on them. As with realizing high performance implementations, realizing high availability implementations is a daunting task.

An example of where achieving high availability is hard is making Figure 1.5 more available by allowing it to be accessed and updated concurrently by multiple threads. On the other hand, other technologies are designed with concurrent interaction in mind, such as relational databases [Bernstein et al., 1987].

Modifiability

Organizations change over time. So does their business logic, and the structure of their data. Software developers have dropped the traditional waterfall development approach in favor of continuous delivery [Boehm, 1988; Humble and Farley, 2010]. In order for developers to be able to change an information system to accommodate new user requirements, the code should be easy to modify [Oskarsson, 1982]. When information systems have good modifiability, changing requirements can be implemented faster, and the information system will be more useful for users.

An example of bad modifiability is boiler-plate code. The bidirectional relation maintenance code in Figure 1.5 is repeated for every bidirectional relation in object oriented languages. When such relations need to be changed, developers have to do a lot of manual work. On the other hand, bidirectional relations in relational databases are supported natively, so no boiler-plate code is required there.

While these concerns are relatively easy to address in isolation, addressing them all at the same time is non trivial. For example, performance and validatability are at odds with each other, as making code more performant often means obscuring its original intent in caching patterns. *Our vision is to address these concerns for developing information systems all at the same time, as it would improve information system development and use tremendously.*

1.2 RESEARCH CONTEXT

Information systems can be built with a plethora of technologies, including programming languages, libraries, frameworks, modeling tools, databases, and combinations of these. It would be impossible to list all state-of-the-art information system technologies in order to assess whether they address the listed concerns. So instead, we examine combinations of concerns to narrow down the list of viable technologies and define the scope of this dissertation.

First, let us examine the combination of performance and validatability. Many information systems filter and process data to create new data: *derived data*. Concurrent with this filtering and processing, users modify the original data. For these situations it is beneficial to not recompute all derived data from scratch after every small change, but to reuse previous results and only compute the changes to the derived data: *incremental computing*. Often, this results in orders of magnitude speedups. Programs can be made incremental manually, but this obfuscates the original intent of the business logic in caching patterns (such as Figure 1.5). Instead, we look at technologies which can make programs behave incrementally automatically. This limits the list of viable technologies to automatic incrementalization such as in materialized views, reactive programming, or incremental computing languages (ICLs).

Second, we examine the combination of reliability and modifiability. In order for developers to guarantee certain properties that users rely on (such as user data not being corrupted) developers need a tool to be able to give that guarantee. General purpose languages can provide easy modifiability, but do

not provide a way to guarantee properties (Figure 1.4). On the other hand, dependently typed languages [Xi and Pfenning, 1999] can encode some properties in which case a type checker can guarantee that these properties always hold. However, modifying dependently typed programs is much harder because the property proof burden is on the programmer. Modifiability would be much better if all programs written in a language would have the desired property by construction. This means creating a language in which only a specific set of programs can be written, which all have the desired property. Such a property is usually relevant for a specific domain, which makes the language a domain-specific language (DSL) [Fowler, 2010]. And indeed, a recent Delphi study[1] with 143 information systems academics identified model-driven (or in our case language-driven) generation of information system implementations as one of the information system research challenges [Becker et al., 2015].

However, none of the existing incremental computing technologies admits the type of calculations we want to express without boiler-plate code. Moreover, none of the existing information system DSLs supports incremental computing. Thus, in this dissertation we explore creating incremental computing DSLs for information systems. With these DSLs we try to address all the raised concerns simultaneously. It might be possible to address the raised concerns with other technologies, but in this dissertation we limit the scope to incremental computing and domain-specific languages for information systems. As specifications in incremental computing DSLs only specify 'what' needs to be computed, and not 'how', we call these declarative. *Our hypothesis is that declarative specification of information system data models and business logic is feasible and useful.*

1.3 CONTRIBUTIONS

The main contributions in this dissertation are new (incremental computing) DSLs for information systems: the Relations Language, IceDust, IceDust2, and PixieDust. Or more precisely, the main contributions are the language features of these DSLs. These language features improve either validatability, traceability, reliability, performance, availability, or modifiability over the state-of-the-art. For each of these language features we show their feasibility and claim (if we can) their usefulness. Moreover, we provide proper evidence for these claims. Our claims and corresponding evidence are summarized in Table 1.1.

1.3.1 *Native multiplicities and concise navigation of first-class n-ary bidirectional relations*

To specify an information system (or any other system) its data model and business logic over this data model need to be specified. This raises the question in what language these data models and business rules should be spec-

[1] The Delphi method is an iterative communication method relying on a panel of experts in which the range of the answers decreases and the group can converge.

	Claim	Evidence	Chapter
1	Native multiplicities are feasible	Relations language formalization	2.4, 2.6
		Rel, Ice, and Pixie implementation	3.4, 4.6, GitHub
		Multiplicity soundness mech. proof	A.1, A.4, GitHub
		Micro case studies	2.3
2	Native multiplicities are useful for information systems	Weblab case study (comparison)	6.5
3	Concise navigation for first-class n-ary relations is feasible	Relations language formalization	2.4, 2.6
		Relations language implementation	GitHub
4	Path-based incremental and eventual computing is feasible	IceDust and PixieDust formalization	3.3, 4.4, 5.5, 5.6
		IceDust and PixieDust implementa.	3.4, GitHub
		Incrementality pen and paper proof	4.4
		Micro benchmarks	3.5, 5.7
		Micro case studies	3.6, 5.7
5	Path-based eventual computing is useful for information systems	Weblab case study (comparison)	6.5
		Weblab application benchmarks	6.5
6	Path-based derived incremental bidirectional relations are feasible	IceDust2 formalization	4.3, 4.4
		IceDust2 implementation	4.6, GitHub
		Micro case studies	4.7
7	Strategy composition is feasible	IceDust2 formalization	4.4, 4.5
		IceDust2 implementation	4.6, GitHub
		Micro case studies	4.7
8	Strategy composition is useful for information systems	Weblab case study (comparison)	6.5

Table 1.1 The claims in this dissertation with their evidence

ified. Object-oriented programming languages support concise navigation of relations represented by references. However, relations are not first-class citizens and bidirectional navigation is not supported. The relational paradigm provides first-class relations, but with bidirectional navigation through verbose queries. Therefore, both object-oriented and relational code has encodings and bad validatability, moreover the object-oriented code for bidirectional relations also has bad modifiability.

In Chapter 2, we present a systematic analysis of approaches to modeling and navigating relations. By unifying and generalizing the features of these approaches, we developed the design of a data modeling language that features first-class relations, n-ary relations, native multiplicities, bidirectional relations and concise navigation. The data models expressed in this new data modeling language have less encoding which improves their validatability.

These language features are summarized by claims 1 through 3 in Table 1.1. Note that we only claim feasibility, and not usefulness, for concise navigation of first-class n-ary relations. Because we have not used these first-class n-ary relations in any real-life information system, we have not (yet) gathered any evidence for their usefulness.

1.3.2 *Path-based incremental and eventual computing*

Business logic in information systems specifies derived values which are calculated from base values. Derived can be expressed in object-oriented languages by means of getters calculating the derived value, and in relational or logic databases by means of (materialized) views. However, switching to

a different calculation strategy (for example caching) in object-oriented programming requires invasive code changes, and the databases limit expressiveness by disallowing recursive aggregation. Without enough expressiveness a technology cannot be used to develop information systems, and invasive code changes result in bad modifiability.

In Chapter 3, we present IceDust, a data modeling language for expressing derived attribute values without committing to a calculation strategy. IceDust provides three strategies for calculating derived values in persistent object graphs: Calculate-on-Read, Calculate-on-Write, and Calculate-Eventually. We have developed a path-based abstract interpretation that provides static dependency analysis to generate code for these strategies. Benchmarks show that different strategies perform better in different scenarios. In addition we have conducted a case study that suggests that derived value calculations of systems used in practice can be expressed in IceDust. Information systems expressed in IceDust can be performant without sacrificing modifiability. Moreover, the eventual computing strategy features good availability.

In Chapter 5, we present PixieDust, a declarative user-interface language for browser-based applications. PixieDust uses the same static dependency analysis to incrementally update a browser-DOM at runtime, without boilerplate code. We demonstrate that applications in PixieDust contain less boilerplate code than state-of-the-art approaches, while achieving on-par performance. Thus, user interfaces expressed in PixieDust can be performant without sacrificing modifiability.

These language features are summarized by claims 4 and 5 in Table 1.1. Note that we do not claim usefulness of incremental computing for information systems. In our in-depth case study (Chapter 6) only eventual computing could provide adequate availability. Moreover, we do not claim usefulness of incremental computing for user-interfaces. We have not done any case study (yet) supporting that claim.

1.3.3 *Derived bidirectional relations and strategy composition*

Derived values in information systems 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. This means bad validatability of relational views and bad modifiability and validatability for reactive programming. 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. Non-checked composition of strategies means bad reliability as developers have to manually ensure correct composition.

In Chapter 4, 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. Information systems written in IceDust2 have good modifiability, validatability, and reliability.

These features are summarized by claims 6 through 8 in Table 1.1. Note that we do not claim usefulness for derived incremental bidirectional relations. In our in-depth case study (Chapter 6) path-based derived bidirectional relations did not perform adequately, we had to use the relational engine from the underlying database to get proper performance.

In conclusion, these language features improve either validatability, reliability, performance, availability, or modifiability over the state-of-the-art. All the DSLs have good traceability as well by means of derived value attributes (explained in Chapter 2), but this is not a contribution in itself as previous work already featured derived value attributes. The DSLs, and the use of these DSLs, presented in this dissertation support our hypothesis that declarative specification of information system data models and business logic is feasible and useful.

1.4 RESEARCH METHODOLOGY

[Shaw, 2003] identified five types of software engineering research questions based on the submissions to previous year International Conference on Software Engineering (ICSE). The type of question we answer in this dissertation is a "method or means of development": what is a better way to develop information systems? Answering that question means *designing* a new method or means of development. In 2011, Österle et al. published a memorandum on design-oriented research in the European Journal of Information Systems [Österle et al., 2011]. We follow the iterative research process described in that memorandum. The first four core chapters of this dissertation are all iterations of that research process, while the fifth is a partial iteration.

The iterative process consists of four phases: analysis, design, evaluation, and diffusion. In the analysis phase we identify and describe information system development problems. We survey and analyze state-of-the-art approaches and outline possible improvements. All core chapters state the problems with state-of-the-art approaches being tackled in that chapter.

In the design phase we design new DSLs (or DSL features). We justify our design choices by design-space analyses and contrast our design to related work. All core chapters justify our design choices and contrast our work with existing solutions.

In the evaluation phase we evaluate our DSL design by applying it in practice and subjecting it to scrutiny. The practical evaluation consists of implementing our designed DSLs, and building information systems with these

Evidence	Methodology
Language formalizations	I-MSOS, inference rules, grammars
Language implementations	Continous integration & many tests
Mechanized proofs	Coq
Informal arguments	Standard logical constructs
Micro benchmarks	Benchmark maximizing internal validity
Application benchmarks	Benchmark maximizing external validity
Micro case study (examples)	Case study maximizing internal validity
Case study (implementation comparison)	Case study maximizing external validity

Table 1.2 The pieces of evidence in this dissertation with their methodology

DSLs (case studies). The DSLs are subjected to scrutiny by presenting them in a comprehensible manner (grammar, static semantics, dynamic semantics) and supplying them to peer review. This step pushes us to get to the essence of our DSLs, and often leads to removing accidental complexity from our language design. All core chapters contain a comprehensible presentation of our DSLs and case studies detailing their use in practice.

In the diffusion phase we publish our findings at scientific conferences, apply our research in real-life applications, and let others build new DSLs on top of our DSLs. All core chapters are peer-reviewed in leading programming language conferences. Chapter 6 details a real-life application, and Chapter 5 describes a DSL built on top of the DSLs in the preceding chapters.

1.4.1 *Individual artifact methodologies*

In this dissertation we introduce new notations (DSLs), new tools (DSL implementations), and some new techniques (which are embodied by these DSLs) [Shaw, 2003]. To show the feasibility and usefulness of these, we produced a variety of research artifacts (the evidence in Table 1.1). Each of these artifacts was produced by adhering to a methodology specific to that type of artifact (Table 1.2).

Language formalizations

The de facto standard to communicate a new programming language or DSL to the scientific community is describing its grammar, static semantics (optionally), and dynamic semantics. Dynamic semantics in this dissertation are formalized in the I-MSOS style [Mosses and New, 2009]. Similarly, static semantics are formalized using inference rules [Pierce, 2002]. Grammars are formalized in production rules. Using these familiar notations, our languages are properly understood and reviewed by peers.

Language implementations

To ensure our language implementations are correct we employ two techniques. First, we write a rigorous test suite for our languages. We write unit tests for syntax, static semantics, and dynamic semantics covering all language features, and integration test which include full programs. Second, we express our languages in DSLs closely resembling our language formalizations when possible. For this we use Spoofax [Kats and Visser, 2010], and

its DSLs for grammars [Visser, 1997; Vollebregt et al., 2012], static semantics [Konat et al., 2012; van Antwerpen et al., 2016], transformations [Visser, 2002, 2003], and tests [Kats et al., 2011]. To ensure that our language implementations stay correct, we use continuous integration to rebuild and run all tests after every commit to either our language or Spoofax implementation.

Proofs

The claimed properties of our languages are accompanied by informal arguments or proofs. These properties are formalized in lemmas about the formal semantics. The informal arguments are described in plain English, but one proof is mechanized in Coq [Barras et al., 1997]. The informal arguments have all been subjected to peer review.

Benchmarks

Benchmarks in our research serve two goals. They illustrate that our techniques work at all, and show that our techniques are useful in real life. To serve both goals we use two types of benchmarks as suggested by Vitek et al. [Vitek and Kalibera, 2012]. First, we use micro-benchmarks to maximize internal validity. Micro-benchmarks are effective at showing the effect of techniques [Siegmund et al., 2015]. Second, we use application benchmarks to maximize external validity. Application benchmarks are effective at establishing that a technique works in a real-life scenario.

We avoid the deadly sins mentioned by Vitek et al. [Vitek and Kalibera, 2012]. Hardware and software assumptions are made explicit to prevent innocuous aspects of experiments introducing a measurement bias. Our data is open where possible such that experiments can be repeated. We report uncertainty to ensure we do not report noise as improvement. We avoid meaningless measurements by changing various parameters and restarting the whole technology stack in micro-benchmarks. We report baseline performance for our benchmarks (either from manual implementations or from competitor languages). Finally, we report on a variety of workloads [Boral and DeWitt, 1984] for micro-benchmarks, and base our workloads for application benchmarks on real-life use of information systems.

Case Studies

Case studies in our research serve the same goals as benchmarks: illustrate feasibility and usefulness. Illustrating feasibility of techniques is shown by how techniques work on examples, while usefulness is shown by comparing implementations of systems in actual use [Shaw, 2003]. Small examples, based on larger systems, maximize internal validity. With these small examples, it is clear that improvements on information systems can be attributed to a new DSL. The large case studies maximize external validity by re-implementing a complete information system in a new DSL. These large case studies establish that our DSLs are useful for implementing real-life information systems. All case studies were performed within our university in collaboration with a group of scientific programmers which build information systems for inter-

nal customers (within the university) or external scientific organizations. All case studies were drawn from the real-life information systems maintained by these scientific programmers.

1.5 ORIGIN OF CHAPTERS

The core chapters (Chapter 2-6) in this dissertation are slight adaptations of peer-reviewed papers at programming language and software engineering conferences. Since these papers were published independently, they can also be read independently of each other. Since all papers have their own, individual contributions, there is some redundancy in the background material, motivation, and examples. In addition, some chapters end with a postscript section presenting our updated view on the chapter since its publication.

- Chapter 2 is an updated version of the SLE 2014 paper *Unifying and generalizing relations in role-based data modeling and navigation* [Harkes and Visser, 2014].
- Chapter 3 is an updated version of the ECOOP 2016 paper *Icedust: Incremental and Eventual Computation of Derived Values in Persistent Object Graphs* [Harkes et al., 2016].
- Chapter 4 is an updated version of the ECOOP 2017 paper *IceDust 2: Derived Bidirectional Relations and Calculation Strategy Composition* [Harkes and Visser, 2017].
- Chapter 5 is an updated version of the WPDAI @ WWW 2018 paper *PixieDust: Declarative Incremental User Interface Rendering through Static Dependency Tracking* [ten Veen et al., 2018].
- Chapter 6 is an updated version of the SLE 2018 paper *Migrating Business Logic to an Incremental Computing DSL: A Case Study* [Harkes et al., 2018].

Relations Language

Unifying and generalizing relations in role-based data modeling and navigation [1]

2

Object-oriented programming languages support concise navigation of relations represented by references. However, relations are not first-class citizens and bidirectional navigation is not supported. The relational paradigm provides first-class relations, but with bidirectional navigation through verbose queries. We present a systematic analysis of approaches to modeling and navigating relations. By unifying and generalizing the features of these approaches, we developed the design of a data modeling language that features first-class relations, n-ary relations, native multiplicities, bidirectional relations and concise navigation.

2.1 INTRODUCTION

Object-oriented programming languages model data with object graphs. Navigation through object graphs is simple; following references leads to related objects. But references in object graphs are one-directional and cannot be navigated backwards. Bidirectional navigation can be obtained by storing references on both sides of relations between objects. But keeping such redundant references consistent requires bookkeeping code. By contrast, relational databases support bidirectional navigation. Foreign keys can be used in queries to navigate both ways. There is no need for redundant references. Queries are however not as concise as navigation through references.

Proposals for object-oriented languages with first-class relations provide bidirectional navigation [Balzer et al., 2007]. These languages remove the need for manually keeping references consistent but navigation is done through querying, which is still verbose. There are modeling techniques that are yet different from object-oriented and relational modeling: Object-Role modeling [Halpin, 2006], Entity-Relationship modeling [Chen, 1976], UML [Jacobson et al., 1999] and undirected graphs.

In this chapter, we present a systematic analysis of the design space of relations in data modeling and present a new data modeling language that unifies and generalizes relations. In particular, our contributions are:

[1] This chapter has appeared as Harkes, D. C. and Visser, E. (2014). Unifying and generalizing relations in role- based data modeling and navigation. In Combemale, B., Pearce, D. J., Barais, O., and Vinju, J. J., 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

```
class Student { }

class Course {
  @any(ArrayList.class) Student student;

  void addStudent(@any(ArrayList.class) Student s){
    this.student += s;
  }
}
```

Figure 2.1 Multiplicity annotations in Java

- We extrapolate Steimann's approach [Steimann, 2013] to model multiplicities using annotations in Java to *native multiplicities* that are integrated into the type system (Section 2.2.2).
- A systematic analysis of approaches to modeling relations (Section 2.3).
- A new relational data modeling language featuring native multiplicities, bidirectional navigation, n-ary relations, first-class relations, and concise navigation expressions based on the analysis (Section 2.4).
- A formal definition of the type system (Section 2.5) and operational semantics (Section 2.6) of this language.

2.2 NATIVE MULTIPLICITIES

The first thing we need to fix to get relations right is the treatment of their cardinality or *multiplicity*. Encoding of *to-many* relations as associations to collections results in a discontinuity in programming style [Steimann, 2013]:

- Navigating *one-to-one* and *many-to-one* relations produces singleton values, while navigating through *one-to-many* and *many-to-many* relations produces collections of values. Thus, the caller has to unwrap the result before using it, for example by using an iterator.
- The caller has to deal with different sub-type substitution conditions. Suppose `Student` extends `Person`. Assigning an `Student` to a `Person` is fine (*to-one*), but trying to assign `Set<Student>` to `Set<Person>` will trigger a type error (*to-many*).
- The call semantics is call-by-value for *to-one* and call-by-reference for *to-many*. Collection objects are passed by reference, so that they can be modified the callee. Call-by-value semantics for collections requires immutable collections.

2.2.1 *Multiplicity Annotations*

To address these issues, Steimann proposes an extension of regular object-oriented programming with multiplicities [Steimann, 2013]. He presents an

```
class Student {
  String! name;
  Course* courses;
  int! numCourses(){ return count(this.courses); }
}
class Course {
  Student* students;
  void addStudent(Student+ s){ this.students += s; }
  int? avgNumCourses(){ return avg(this.students.numCourses()); }
}
```

Figure 2.2 Native multiplicities in Java

extension of Java with multiplicity. Expressions of a singleton value type can return an arbitrary number of objects of this type. Figure 2.1 illustrates the approach with a small example in which a `Course` has an association to `Student`. Through the `@any` annotation the association is declared to be to-many instead of using a collection type.

2.2.2 *Native Multiplicities*

We have extrapolated Steimann's annotations based approach and integrated multiplicities into the type system to arrive at *native* multiplicities. Type expressions use one of the following four multiplicity operators (similar to regular expressions) to denote the possible range of values:

- `t?` is $[0,1]$ an optional value of type `t`
- `t!` is $[1,1]$ a required value of type `t`
- `t*` is $[0,n)$ zero or more values of type `t`
- `t+` is $[1,n)$ one or more values of type `t`

The `!` can be omitted as $[1,1]$ is the default multiplicity.

As a sketch, Figure 2.2 illustrates native multiplicities in an extension of Java. We have not formalized an extension of Java, but rather integrated native multiplicities in our relational data modeling language. In Section 2.5 we formalize a type system for that language including multiplicities. The type system ensures that the actual number of values at run-time is always inside the specified range. For example, assigning an optional string (a value of type `String?`) to a `student.name` will trigger a type error: *multiplicity error:* $[1,1]$ *expected,* $[0,1]$ *given*. Our language also supports expected multiplicities for function arguments. The built-in function `count` handles any multiplicity and any type and it returns exactly one integer with the number of values passed. The built-in function `avg` also handles $[0,n)$ values and the argument type must be numeric. The return multiplicity of `avg` depends on its input multiplicity. If a programmer supplies $[0,n)$ as input the return multiplicity will be $[0,1]$. The average of no values does not exist, so no value will be returned in that case. If the programmer supplies $[1,n)$ as input the return multiplicity is $[1,1]$. With at least one value there is always an average computable. We use this model of multiplicities, reasoning over ranges, in the type system of our language.

2.3 DESIGN SPACE FOR ROLE-BASED RELATIONS

There are several proposals in the literature for extending data modeling to better support data modeling with relations. This section presents a systematic analysis of the design space of relations in data modeling taking in into account these proposals. Figures 2.3 and 2.4 summarize the complete design space in tabular form emphasizing its regularities. From this analysis a new data modeling language emerges which unifies and generalizes the various approaches to modeling relations.

In all our examples we assume the language to have native multiplicities instead of using collections that would be needed in a plain OO approach. The running example data model defines `Student`s who are enrolled in `Course`s, sometimes via a first-class `Enrollment` relation. For the sake of the example, students can be enrolled in zero or more courses (`*` multiplicity), and courses should have at least one student (`+` multiplicity). In the example expressions we use `Student` 'bob' and `Course` 'math'. For each point in the design space we give a type graph diagram describing the data model, a textual specification of the data model, and expressions for querying the model. For the expressions we use `=>` to express the result of evaluation.

2.3.1 *Overview*

Before discussing each point in the design space (Figures 2.3 and 2.4) individually, we first introduce the categories represented by the columns and rows.

Columns: Four Modeling Paradigms

The four columns in the design space represent four modeling paradigms.

Object-Oriented Relations between objects are defined through reference valued attributes, which can be navigated in one direction only. The name of the relation is the name of the attribute in the source class. The relation is unknown to the target class. A relation can also be modeled by, redundantly, maintaining a reference attribute on the other side of the relation, as well, allowing bidirectional navigation. However, this requires code for keeping the two sides of the relation consistent. We do not cover models with redundant information in our design-space analysis, as this is an undesirable property.

Relational In a relational database schema references are expressed as foreign keys; an identifier corresponds to a memory address and a foreign key to a reference into memory. An important difference is that these references can be navigated in two directions through queries in a query language (SQL). ER and UML diagrams are also located in this column, but they only provide schema definitions, not queries. Because queries are verbose we introduce our own notation for forward and backward navigation through references. For forward navigation we use the the normal field access notation. For backward navigation from an object `o` we need to find all the objects of type `T` that re-

fer to `o` through references `r`, which is expressed by `o<-(T.r)`. For example, to find the students enrolled in a course `c` we use the navigation expression `c<-(Student.courses)`.

Object-Role Modeling A distinguishing feature of ORM [Halpin, 2006] is that associations between objects have a different name on both sides. This conceptually solves the problem of not being able to refer to a reference backwards. Similarly, inverse properties in WebDSL [Visser, 2007] and bidirectional bindings in JavaFX [url, 2019] tie two fields in different classes together as inverses.

Graph databases In contrast to the directed edges in the previous three paradigms, graph databases feature undirected edges. In this model the edge names are defined in both source and target namespaces. As with the ORM paradigm there is always a name available in the namespace of participating objects, but in this case this name is identical for both sides. There is one disadvantage of this model: modeling asymmetric same type relations is non-trivial. Consider a `TreeNode` with a parent and children. If a node `p` has a parent edge to another node `q`, then `q` also has a parent edge to `p`. This can be solved through indirection (J and K), but that is not particularly elegant.

Rows: Three Relation Models

The three rows in the design space correspond to three ways of modeling a relation.

Edge The simplest way of representing a relation is through an edge between two nodes (either directed or undirected). This is a concise way of specifying a relation but it has the disadvantage that the relation is not a first-class citizen (see below). Also it is not possible to declare ternary, or higher arity, relations with edges.

Tuple (Ordered Roles) By lifting relations to objects they become *first-class citizens*, i.e. relations can have attributes, and relations can be the subject in other relations. A relation object modeled as a tuple has ordered roles. The absence of role names requires the order (or position) of the roles to be used for navigation. For binary relations this entails four predefined navigation operators (see E). But for higher arity relations 2^n operators are required, which does not scale.

Object (Named Roles) Giving the roles in a relation names makes navigation understandable and makes modeling n-ary relations feasible.

2.3.2 *Detailed Description of Points in Design Space*

We discuss each of the points A to K in the design space (Figures 2.3 and 2.4).

Object-Oriented (A, B and C) There are multiple patterns for modeling relations in objected-oriented languages [Noble, 1997]. As mentioned before, we replace collections by multiplicities and do not consider patterns with redundant references for bidirectional navigation. Three basic patterns remain: reference (A), relation tuple (B), and relation class (C), which we assume to

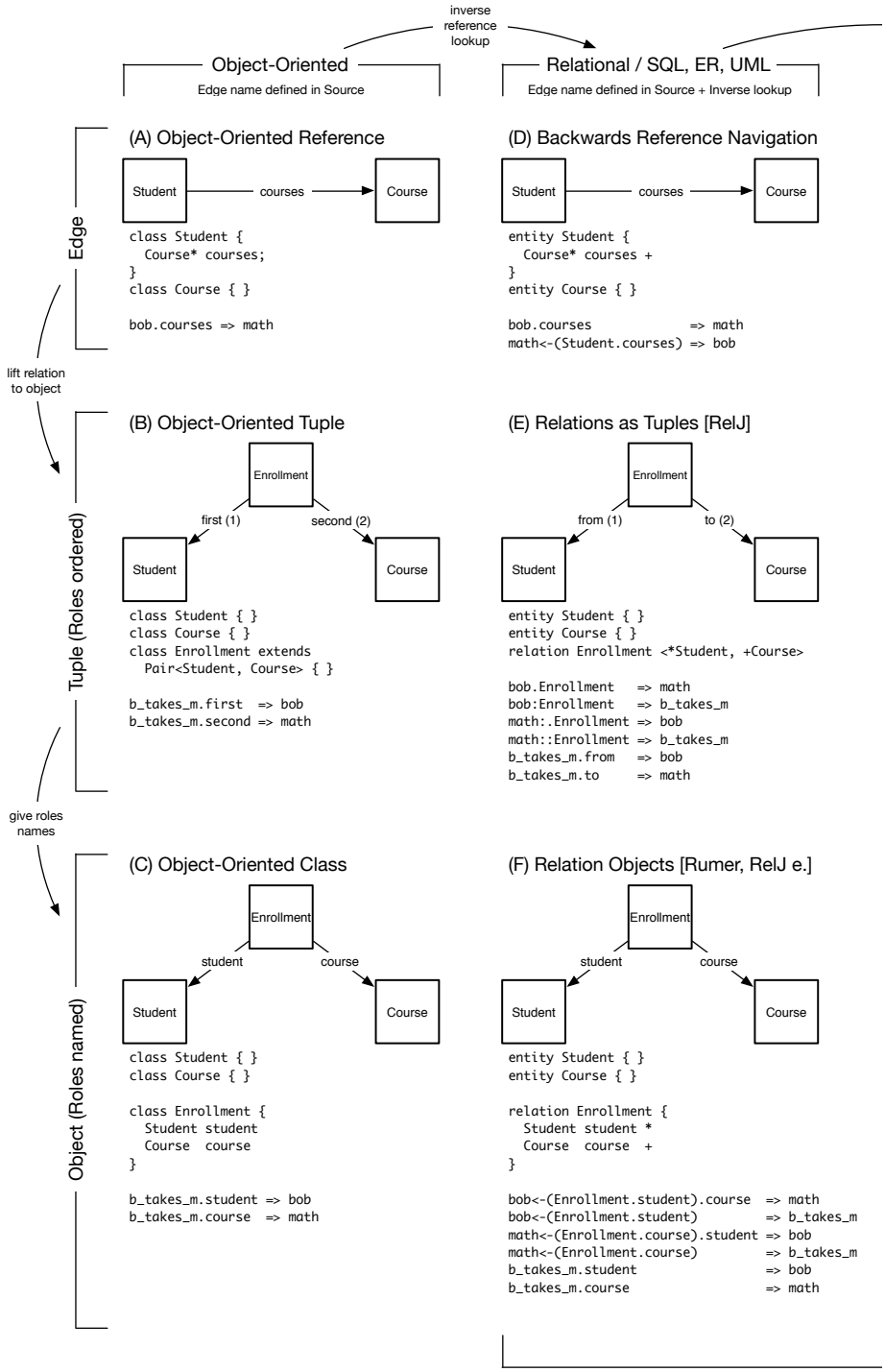


Figure 2.3 Design space of relations in data modeling and navigation (part 1)

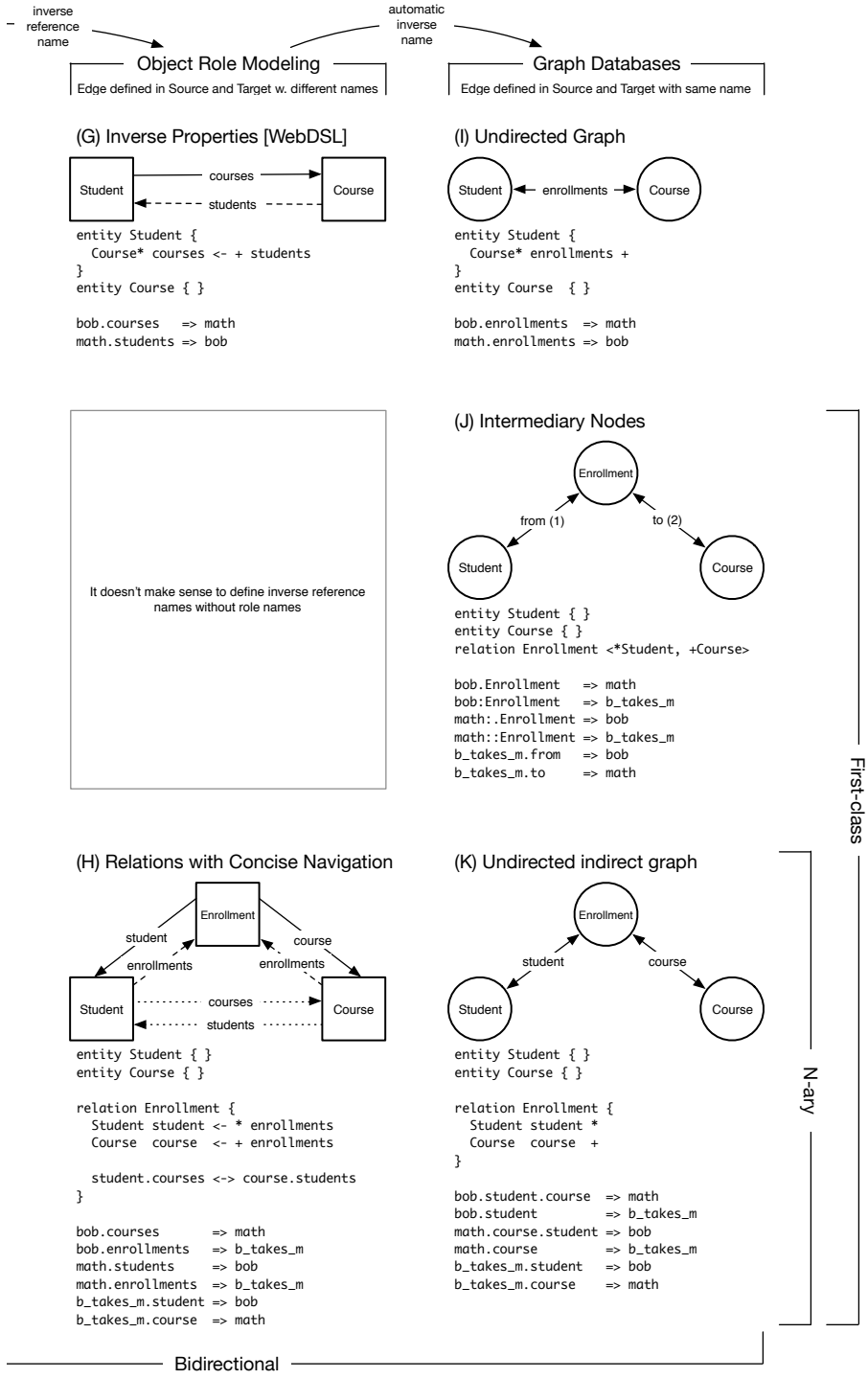


Figure 2.4 Design space of relations in data modeling and navigation (part 2)

```
class Student { }
class Course { }
relationship Enrollment (Student, Course) { int grade; }

bob.Enrollment       // bob's courses
bob:Enrollment       // Enrollment-type relation objects
bob:Enrollment.grade
b_takes_m.from       // bob
b_takes_m.to         // math
```

Figure 2.5 First-class citizen tuple based relations in RelJ [Bierman and Wren, 2005].

```
class Student { }
class Course { }
relationship Enrollment participants (Student student, Course course){
  int grade;
}
Enrollment.select(s_c: s_c.course == math).student;
```

Figure 2.6 First-class relations with named roles in Rumer [Balzer, 2011; Balzer et al., 2007].

be familiar to the reader. It is noteworthy that a language extension is not required for the representation of first-class relations. The term first-class is sometimes used for having a dedicated language construct, but a dedicated language construct is not required for adding attributes to relations or letting relations participate in other relations. First-class relations based on tuples (B) have been implemented as a Java library [Nelson et al., 2008].

Backwards reference navigation (D) If we extend an object-oriented language with facilities for backwards reference lookup (`o<-(T.r)`) we can use a single reference for bidirectional navigation. Note that in this case the object graph is identical to the single reference pattern (A).

Relation as Tuples (E) The RelJ Java extension lifts relations to tuple objects [Bierman and Wren, 2005]. In RelJ different operators are used to disambiguate between different navigation operations (Figure 2.5). RelJ provides no facilities for bidirectional navigation. However, that is not a conceptual limitation. Adding two operators (`:.` and `::`) would allow backward navigation, as suggested in (E). While this is theoretically extensible to relations with more than two participants, it requires adding new operators for each participant.

Relation Objects (F) Naming roles allows usable extension to n-ary relations. This is the model used by Rumer [Balzer, 2011; Balzer et al., 2007] as illustrated in Figure 2.6. While Rumer's implementation does not support n-ary relations, it provides the ingredients needed for n-ary relations: role names and first-class citizenship. A proposed extension for RelJ [Wren, 2007] adds names to roles, as illustrated in Figure 2.7, and is essentially equivalent to Rumer's syntax. As an alternative query syntax, we propose `math<-(Enrollment.course).student`, which is closer to the usual navigation syntax: from an object (`math`) find all relations with that object in one of its

```
class Student { }
class Course { }
relationship Enrollment
     extends Relation (Student student, Course course, Student tutor){
  int grade;
}
Enrollment[course == math].student; // math students
```

Figure 2.7 Ternary relation extension proposal for RelJ [Wren, 2007]

```
entity Student { courses : Set<Course> }
entity Course { students : Set<Student> (inverse=Student.courses) }
math.students  // math students
bob.courses    // bobs courses
```

Figure 2.8 Inverse properties in WebDSL

roles (`Enrollment.course`), and produce objects in the other role (`student`). All these notations are rather verbose, even if more concise than full blown SQL queries. We would prefer a more concise notation for navigating n-ary relations.

Inverse Properties (G) WebDSL [Visser, 2007] supports bidirectional navigation without a verbose syntax for inverse lookups by means of *inverse properties* [Hemel et al., 2011] as illustrated in Figure 2.8. Explicit names on both sides of an association simplifies navigation to just following named references. However, these names have to be defined in both the source and target class. In (G) we have normalized this to a single property definition with two names; the second name is used for the backwards reference from target to source.

Concise Relations (H) Combining the advantages of (F) and (G), we arrive at our proposal for a unified and generalized approach to modeling relations (H). Relations are first-class citizens: (1) relations can have attributes and (2) relations can be the subject in other relations. In addition, relations can have any number of roles (n-ary relations). By explicitly providing a name for the navigation between each pair of participants in the relation we get concise navigation expressions: (1) from relation to participant and back (`b_takes_m.student` and `bob.enrollments`), and (2) from participant to other participant (`bob.courses`) and back (`math.students`). Instead of defining these names in the source and target classes, as in (G), all names are introduced in the relation. The declaration of a role `T r <- m i` introduces a role `r` of type `T` with inverse `i` with multiplicity `m`. This provides navigation from relation to participant through `r` and navigation from participant to relation through `i`. A declaration `r1.n1 <-> r2.n2` introduces names for navigation between participants: `r1.n1` leads to `r2` and `r2.n2` leads to `r1`. In contrast to (G), these declarations do not introduce attributes in the participant classes, but rather shortcuts. For example, `bob.courses` is a shortcut for `bob.enrollments.course`. This approach naturally extends to n-ary relations, as illustrated in Figure 2.9.

```
entity Student { }
entity Course { }
relation Enrollment {
  Student student <- * enrollments
  Course  course  <- + enrollments
  Student tutor   <- * tutoring

  student.courses <-> course.students
  student.tutors  <-> tutor.students
  course.tutors   <-> tutor.courses
}
```

Figure 2.9 Ternary relation with concise navigation (H) (our design)

Undirected Graphs (I, J, K) Graph databases also feature three relation patterns. The simple edge (I), adding an intermediary node without role names (J), and an intermediary node with role names (K). Since without edge names, edge directionality does not matter (J) is equivalent to (E). So we will only cover (I) and (K).

The simple edge (I) cannot be used to model asymmetric same type relations. Asymmetric relations of the different types can be disambiguated by the type one starts navigating from, but if both participants have the same type their role is ambiguous. Disambiguation can be done through indirection (I or K). With indirection (K) navigation from participant to participant is navigating two edges. With undirected edges role names cannot be reused with different relations concerning the same entity. Consider adding another relation where `Course` also participates as `course`. `math.course` now becomes ambiguous. The language could then be extended with the type of the node navigating to, but this is equivalent to the backwards reference navigation: naming the edge and the type on the other side. So that would bring us back at (F).

It seems there is a fundamental trade-off between undirected and directed graphs when considering reference names. The directed graph (column two) requires an extra identifier (the target type) to navigate edges backwards. To get rid of this extra identifier we can automatically define the edge name on both sides. This is gets us to the undirected graph (column four). In undirected graphs we have ambiguities. Adding an extra identifier (the target type) to disambiguate brings us back at the directed graphs.

2.4 A RELATIONAL DATA MODELING LANGUAGE

We have designed a language for data modeling featuring native multiplicities, bidirectional navigation, n-ary relations, first-class relations, and concise navigation expressions based on point (H) in the design space. In this section we discuss two extensions of the basic idea of (H) and the grammar of the language. In the next sections we give a formal definition of the type system and operational semantics.

```
relation Enrollment { Student* Course+ }
```

expands to (lower case participant and relation type, add s for * and +)

```
relation Enrollment {
  Student student <- * enrollments
  Course  course  <- + enrollments
}
```

expands to (use role name, add s for * and +)

```
relation Enrollment {
  Student student <- * enrollments
  Course  course  <- + enrollments
  student.courses <-> course.students
}
```

Figure 2.10 Expansion of concise relation definition

```
entity Student {
  Int? avgGrade = avg( this.enrollments.grade )
}
```

Figure 2.11 Relations language with derivation

Concise Definition of Relations While navigation according to (H) is very concise, the definition of a relation is somewhat verbose due to the introduction of names for each of the arrows in the diagram. In many cases we can derive these names from the types of the roles. Figure 2.10 illustrates how a definition with implicit names is expanded to a definition with explicit names. This automatic expansion can of course lead to name collisions, for example if the participant classes have an attribute with a name introduced by a relation. In this case the programmer has to (partially) specify names explicitly.

Derived Attributes To express business logic in data models, we extend entities and relations with *derived attributes*. The value of a derived attribute is described in terms of the values of other attributes and relations as illustrated in Figure 2.11. Thus, if one of the underlying values changes, the derived attribute is updated.

Grammar The grammar of the relations language is given in Figure 2.12. a, i, r and t are respectively attribute, inverse, role and entity-type names. The roles, r, are the solid arrows in the design space diagram and the inverses/shortcuts, i, are the dashed and dotted arrows. a', i', r', r'', and t' refer to these names. The lookup expression (`t [ a == e ]`) is only intended to look up objects of a certain type with a certain attribute value in the heap. It is not our intention to provide a full-fledged query language; our focus is on navigation expressions.

Prototype We have implemented this language on the language designers workbench Spoofax [Kats and Visser, 2010]. The prototype is publicly available.[2] The type system and semantics described in the next sections matches

[2] https://github.com/metaborg/relations tag v0.2.0

$$
\begin{aligned}
\mathit{Program} &::= \texttt{model}\ \mathit{Entity}\ast\ \texttt{execute}\ e \\
\mathit{Entity} &::= \texttt{entity}\ t\ \{\ \mathit{Attribute}\ast\ \} \\
&\mid \texttt{relation}\ t\ \{\ \mathit{Attribute}\ast\ \mathit{Role}\ast\ \mathit{Shortcut}\ast\ \} \\
\mathit{Attribute} &::= p\ m\ a \\
&\mid p\ m\ a = e \\
\mathit{Role} &::= t'\ r\ \texttt{<-}\ m\ i \\
\mathit{Shortcut} &::= r'\ .\ i\ \texttt{<->}\ r''\ .\ i \\
p \in \mathit{PrimitiveType} &::= \texttt{Boolean} \mid \texttt{Int} \mid \texttt{String} \\
m \in \mathit{Multiplicity} &::= \texttt{?} \mid \texttt{!} \mid \texttt{*} \mid \texttt{+} \\
e \in \mathit{Expr} &::= f\ (\ e\ ) \mid e_1 \oplus e_2 \mid !\,e \mid e_1\ ?\ e_2 : e_3 \\
&\mid e\ .\ a' \mid e\ .\ i' \mid e\ .\ r' \\
&\mid \texttt{true} \mid \texttt{false} \mid \mathit{literalInt} \mid \mathit{literalString} \\
&\mid \texttt{this} \mid t\ [\ a == e\ ] \\
f \in \mathit{AggrOp} &::= \texttt{min} \mid \texttt{max} \mid \texttt{avg} \mid \texttt{sum} \mid \texttt{concat} \mid \texttt{count} \mid \texttt{conj} \mid \texttt{disj} \\
\oplus \in \mathit{BinOp} &::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{\&\&} \mid \texttt{||} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{==} \mid \texttt{=!} \mid \texttt{<+} \mid \texttt{++}
\end{aligned}
$$

Figure 2.12 The grammar of the relations language

$$
\begin{aligned}
\mathcal{P} \in \mathit{Program} &: \mathit{EntityMap} \times \mathit{Expr} \\
\mathcal{E} \in \mathit{EntityMap} &: \mathit{EntityName} \to \mathit{AttributeMap} \times \mathit{InverseMap} \times \mathit{RoleMap} \\
\mathcal{A} \in \mathit{AttributeMap} &: \mathit{AttrName} \to \mathit{PrimitiveType} \times \mathit{Multiplicity} \times \mathit{Expr} \\
\mathcal{I} \in \mathit{InverseMap} &: \mathit{InverseName} \to \mathit{EntityName} \times \mathit{RoleName} \times \mathit{RoleName} \\
\mathcal{R} \in \mathit{RoleMap} &: \mathit{RoleName} \to \mathit{EntityName} \times \mathit{Multiplicity}
\end{aligned}
$$

Figure 2.13 Meta variables used in static and dynamic semantic rules

those of the prototype.

2.5 TYPE SYSTEM

Our language features static typing. Everything in the language has both a *type* and a *multiplicity*. These are defined orthogonally.

2.5.1 *Meta variables*

In the the static and dynamic semantic rules we use meta variables for looking up definitions on usage sites (Figure 2.13).

A program $\mathcal{P}$ is a tuple, $(\mathcal{E}, e)$, where $\mathcal{E}$ is a map from entity (and relation) names to entity definitions and e is the main expression.

Entity definitions are triples $(\mathcal{A}, \mathcal{I}, \mathcal{R})$, where $\mathcal{A}$ is a map from attribute names to attribute definitions, $\mathcal{I}$ is a map of inverse names to their origin and $\mathcal{R}$ is a map from role names to role definitions. Both entities and relations define entities. We refer to an entity t's attribute, inverse and role map as $\mathcal{A}_t$, $\mathcal{I}_t$ and $\mathcal{R}_t$ respectively.

Attribute definitions are triples (p, m, e), where p is the primitive type, m is the multiplicity and e is the optional derivation expression. If e has no

derivation expression it is equal to `nil`. Role definitions are tuples (t, m), where t is an entity name and m is a multiplicity. Inverse (and shortcut) definitions are triples (t, r_1, r_2) where r_1 and r_2 are roles in entity t. The inverse map definition is best explained by example:

```
entity Enrollment {
  Student student <- * enrollment
  Course  course  <- + enrollment
  student.courses <->  course.students
}
```

Is translated to the following:

$$
\begin{array}{rlllll}
\mathcal{I}_{Student}: & \text{'enrollment'} & \to \text{'Enrollment'} & \times \text{'student'} & \times \mathit{nil} \\
 & \text{'courses'} & \to \text{'Enrollment'} & \times \text{'student'} & \times \text{'course'} \\
\mathcal{I}_{Course}: & \text{'enrollment'} & \to \text{'Enrollment'} & \times \text{'course'} & \times \mathit{nil} \\
 & \text{'students'} & \to \text{'Enrollment'} & \times \text{'course'} & \times \text{'students'}
\end{array}
$$

The inverses of roles are mapped back to the role in the relation they are the inverse of. In this case r_2 is `nil`. The shortcut is translated to two records, one for both participant types. The inverse maps are used as the backwards reference navigation mechanism.

Lastly, to simplify static and dynamic semantics we transform the shortcut expressions to an inverse and a role expression by the transformation rule:

$$
\frac{e : t_1 \quad \mathcal{I}_{t1}(i_1) = (t_2, r_1, r_2) \quad \mathcal{I}_{t1}(i_2) = (t_2, r_1, \text{nil})}{e \,.\, i_1 \;\to\; e \,.\, i_2 \,.\, r_2}
$$

2.5.2 *Types*

There are two type sorts: p (*primitive types*) and t (*entity types*). All attributes are primitive types. Entities and relations define entity types. Roles, inverses and shortcuts in a relation are entity types.

Most typing rules are straightforward, so we only cover the rules that are non-standard. The aggregation rule (Aggr) is interesting. Since multiplicities are encoded orthogonally the aggregation functions are of type int $\to$ int. The multiplicity operators choice and concatenate work with any type. They only check whether both operands have the same type and propagate the type (Mult).

With roles and inverses one can conceptually navigate over the type graph defined by the entities and relations. The type of a navigation expression is naturally the place where one ends up in the model after navigating. When navigating from a relation to a participant the type is the participant's type (RoleNav). When navigating from a participant to a relation, by an inverse, we find the type of the relation by looking up the inverse definition (InvNav).

2.5.3 *Multiplicities*

For multiplicities there are two notational conventions: single characters from the concrete syntax and ranges. We use the ranges notation in the multiplicity rules as it gives us access to the upper and lower bounds directly.

Expression type $\theta \vdash Expr : T$

$$\frac{c \in \{\text{true}, \text{false}\}}{c : \text{boolean}} \text{[Bool]}$$

$$\frac{}{literalInt : \text{int}} \text{[Int]}$$

$$\frac{}{literalString : \text{string}} \text{[Str]}$$

$$\frac{}{\theta \vdash \text{this} : \theta} \text{[This]}$$

$$\frac{\oplus \in \{+, -, *, /, \%\} \quad e_1 : \text{int} \quad e_2 : \text{int}}{e_1 \oplus e_2 : \text{int}} \text{[Math]}$$

$$\frac{e_1 : \text{string} \quad e_2 : \text{string}}{e_1 + e_2 : \text{string}} \text{[Conc]}$$

$$\frac{\oplus \in \{\&\&, ||\} \quad e_1 : \text{boolean} \quad e_2 : \text{boolean}}{e_1 \oplus e_2 : \text{boolean}} \text{[AndOr]}$$

$$\frac{e : \text{boolean}}{!\, e : \text{boolean}} \text{[Not]}$$

$$\frac{\oplus \in \{>, >=, <, <=\} \quad e_1 : t \quad e_2 : t \quad t \in \{\text{int}, \text{string}\}}{e_1 \oplus e_2 : \text{boolean}} \text{[Cmp]}$$

$$\frac{e_1 : t \quad e_2 : t \quad \oplus \in \{==, !=\}}{e_1 \oplus e_2 : \text{boolean}} \text{[Eq]}$$

$$\frac{e_1 : \text{boolean} \quad e_2 : t \quad e_3 : t}{e_1 \ ?\ e_2 \ ":"\ e_3 : t} \text{[Cond]}$$

$$\frac{e : \text{int} \quad f \in \{\text{avg}, \text{min}, \text{max}, \text{sum}\}}{f(e) : \text{int}} \text{[Aggr]}$$

$$\frac{e : \text{boolean} \quad f \in \{\text{conj}, \text{disj}\}}{f(e) : \text{boolean}} \text{[Logic]}$$

$$\frac{e : \_}{\text{count}(e) : \text{int}} \text{[Count]}$$

$$\frac{e_1 : t \quad e_2 : t \quad \oplus \in \{<+, ++\}}{e_1 \oplus e_2 : t} \text{[Mult]}$$

$$\frac{e : t \quad \mathcal{A}_t(a) = (p, \_.\_)}{e\ .\ a : p} \text{[Attr]}$$

$$\frac{e : t_a \quad \mathcal{A}_t(a) = (t_a, \_)}{t\ [\ a == e\ ] : t} \text{[Lookup]}$$

$$\frac{e : t \quad \mathcal{R}_t(r) = (t_r, \_)}{e\ .\ r : t_r} \text{[RoleNav]}$$

$$\frac{e_1 : t_1 \quad \mathcal{I}_{t1}(i) = (t_2, \_, \text{nil})}{e\ .\ i : t_2} \text{[InvNav]}$$

Figure 2.14 Type rules

Binary operators mimic maybe-Monad behaviour for zero or one values: a maybe value as input for the computation returns a maybe value as output. Taking the Cartesian product between the bags of values and applying the operation to each pair provides this behaviour. The multiplicity range is expressed as taking the minimum of both lower bounds and the maximum of the upper bounds (BinOp). The division and modulo operators exhibit slightly different behaviour (DivOp). Since dividing by zero has no result, at least one value in both operands might still result in no answer. Instead of throwing a division by zero exception zero answers are given for any denominator equal to zero.

The Choice operator chooses at runtime the left expression if it has a result, and otherwise the right expression. The multiplicity is defined as the maximum of both upper and lower bound, except if the left lower bound is

Expression multiplicity $Expr \sim M$

$$\frac{c \in \{\text{this}, \text{true}, \text{false}, Int, String\}}{c \sim [1,1]} \text{ [Const]}$$

$$\frac{\oplus \in \{+, -, *, \&\&, ||, >, >=, <, <=, ==, !=\} \quad e_1 \sim [l_1, u_1] \quad e_2 \sim [l_2, u_2]}{e_1 \oplus e_2 \sim [min(l_1, l_2), max(u_1, u_2)]} \text{ [BinOp]}$$

$$\frac{\oplus \in \{/, \%\} \quad e_1 \sim [\_, u_1] \quad e_2 \sim [\_, u_2]}{e_1 \oplus e_2 \sim [0, max(u_1, u_2)]} \text{ [DivOp]}$$

$$\frac{e_1 \sim [l_1, 1] \quad e_2 \sim [l_2, u_2] \quad e_3 \sim [l_3, u_3] \quad m = [min(l_1, l_2, l_3), max(u_2, u_3)]}{e_1 \text{ ? } e_2 \text{ ":" } e_3 \sim m} \text{ [Cond]}$$

$$\frac{e \sim m}{! \, e \sim m} \text{ [Not]}$$

$$\frac{f \in \{\text{avg}, \text{min}, \text{max}, \text{conj}, \text{disj}\} \quad e \sim [l, \text{n})}{f(e) \sim [l, 1]} \text{ [Aggr]}$$

$$\frac{f \in \{\text{sum}, \text{count}\}}{f(e) \sim [1,1]} \text{ [Aggr2]}$$

$$\frac{e_1 \sim [0, u_1] \quad e_2 \sim [l_2, u_2]}{e_1 <+ e_2 \sim [l_2, max(u_1, u_2)]} \text{ [Choice]}$$

$$\frac{e_1 \sim [1, u_1]}{e_1 <+ e_2 \sim [1, u_1]} \text{ [Choice2]}$$

$$\frac{e_1 \sim [l_1, \_] \quad e_2 \sim [l_2, \_]}{e_1 ++ e_2 \sim [max(l_1, l_2), n)} \text{ [Concat]}$$

$$\frac{e \sim [l_1, u_1] \quad \mathcal{A}_{t_e}(a) = (\_, [l_2, 1], \_)}{e \, . \, a \sim [min(l_1, l_2), u_1]} \text{ [Attr]}$$

$$\frac{}{t \, [\, a == e \,] \sim [0, n)} \text{ [Lookup]}$$

$$\frac{e : t \quad e \sim m \quad \mathcal{R}_t(r) = (\_, \_)}{e \, . \, r \sim m} \text{ [RoleNav]}$$

$$\frac{e_1 : t_1 \quad \mathcal{I}_{t1}(i) = (t_2, r, \text{nil}) \quad \mathcal{R}_{t2}(r) = (\_, [l_2, u_2])}{e \, . \, i \sim [min(l_1, l_2), max(u_1, u_2)]} \text{ [InvNav]}$$

Figure 2.15 Multiplicity rules

one. Then we know that the left expression will always be chosen. Note that it does not make sense to use the choice operator in that case, because the right expression will be dead code. The CONCAT operator combines the results of both expressions. This means that we might always have more than one value at runtime; thus the upper bound is n. The lower bound is the maximum of both.

Attributes are allowed to be either [0,1] or [1,1]. In the first case attribute access decreases the lower bound to zero, as the attribute might not be set (ATTR). A role always has exactly one value, so role navigation leaves multiplicity intact (ROLENAV). Navigation to relations entities participate in behaves like a SQL join between the input expression entities and the relation. Like binary operators this means taking the lowest lower bound and the highest upper bound.

2.5.4 *Well-formedness*

Programs are well-formed if they satisfy the rules in Figure 2.16. Attributes are only allowed to have a multiplicity of at most one, their type has to be primitive (which is enforced by the syntax definition already) and if a deriva-

Program well-formedness $\vdash Attr|Role|Inv|Shortcut|Entity|Program$

$$\frac{a = (\_, [\_, 1], \text{nil})}{\vdash a} \quad \text{[AttrDec]}$$

$$\frac{a = (p, [l_1, 1], e) \quad e : p \quad e \sim [l_2, 1] \quad l_1 \leq l_2}{\vdash a} \quad \text{[AttrDec2]}$$

$$\frac{r = (t, m) \quad \mathcal{E}(t) = (\_, \_)}{\vdash r} \quad \text{[RoleDec]}$$

$$\frac{i = (t, r_1, \text{nil}) \quad \mathcal{R}_t(r_1) = (\_, \_)}{\vdash i} \quad \text{[InvDec]}$$

$$\frac{i = (t, r_1, r_2) \quad \mathcal{R}_t(r_1) = (\_, \_) \quad \mathcal{R}_t(r_2) = (\_, \_)}{\vdash i} \quad \text{[ShortcutDec]}$$

$$\frac{\theta' = t \quad \forall a \in dom(\mathcal{A}_t) : \theta' \vdash a \quad \forall r \in dom(\mathcal{R}_t) : \theta' \vdash r \quad \forall i \in dom(\mathcal{I}_t) : \theta' \vdash i}{\vdash t} \quad \text{[EntityDec]}$$

$$\frac{\theta' = \bot \quad \forall t \in dom(\mathcal{E}) : \theta' \vdash t \quad \theta' \vdash e : \_ \quad \theta' \vdash e \sim \_}{\vdash (\mathcal{E}, e)} \quad \text{[ProgramDec]}$$

Figure 2.16 Attribute, role, inverse, shortcut, entity and program well-formedness

tion is specified, it should be of the correct type and its multiplicity should fit inside the target range. Role declarations are well-formed if the entity playing the role exists in the entity map. Inversions are well-formed if the role exists in the entity of which they are the inverse and shortcuts are well-formed if both roles exist in the entity. Entity definitions are well-formed if all their attributes, inverses and roles are well-formed and a program is well-formed if all its entities and the main expression are well-formed. We only consider well-formed programs.

2.6 DYNAMIC SEMANTICS

We specify evaluation rules for a big-step semantics. We use the I-MSOS notational style, which implicitly propagates stores if they are not mentioned [Mosses and New, 2009].

2.6.1 *Stores*

In order to evaluate a program an entity store Σ and relation store Δ must be passed; our language is a data modeling and navigation language and does not provide facilities to add, edit or remove data. Expressions in addition get passed a this-reference θ.

Entity store well-formedness $\boxed{\vdash \Sigma}$

$$\frac{\forall (ref \to astore) \in \Sigma : \vdash (ref \to astore)}{\vdash \Sigma} \quad \text{[EntityStore]}$$

$$\frac{\begin{array}{l} ref : t \\ \forall (a \to v) \in astore : \ ref \vdash (a \to v) \\ \forall (a \to p, [1,1], \_) \in \mathcal{A}_t : \ astore(a) = \_ \\ \forall (r \to \_, \_) \in \mathcal{R}_t : \Delta(t, ref, r) = \_ \\ \forall (i \to t_2, r_2, \text{nil}) \in \mathcal{I}_r : \\ \quad \left(|\{v \mid \Delta(t_2, \_, r_2) = v\}| = m \quad \mathcal{R}_{t2}(r_2) = (\_, [l,u]) \quad l \le m \le u\right) \end{array}}{\vdash (ref \to astore)} \quad \text{[EntityRecord]}$$

$$\frac{e : t \quad \mathcal{A}_t(a) = (t_a, \_, \_) \quad v : t_a}{e \vdash a \to v} \quad \text{[AttrRecord]}$$

Relation store well-formedness $\boxed{\vdash \Delta}$

$$\frac{\forall (t\ v_1\ r \to v_2) \in \Delta : \vdash (t\ v_1\ r \to v_2)}{\vdash \Delta} \quad \text{[RelationStore]}$$

$$\frac{v_1 : t \quad \Sigma(v_1) = \_ \quad \mathcal{R}_t(r) = (t_2, \_) \quad v_2 : t_2 \quad \Sigma(v_2) = \_}{\vdash t\ v_1\ r \to v_2} \quad \text{[RelationRecord]}$$

This store well-formedness $\boxed{\vdash \theta}$

$$\frac{\Sigma(\theta) = \_}{\vdash \theta} \quad \text{[ThisReference]}$$

Figure 2.17 Store well-formedness

$$\begin{array}{rl} \Sigma, \Delta \vdash p \Downarrow v & \text{(Evaluation of program)} \\ \Sigma, \Delta, \theta \vdash e \Downarrow v & \text{(Evaluation of expressions)} \end{array}$$

The entity store corresponds to the usual heap: a map from object references to a map from attribute names to their values. The relation store is used for storing all relations between entities. It is a map from relation name, relation object reference and role name to the reference of the object playing this role. The this-reference is a single reference to an object.

$$\begin{array}{rl} \Sigma \in EntityStore & : Reference \to AttributeStore \\ AttributeStore & : AttrName \to Value \\ \Delta \in RelationStore & : EntityName \times Reference \times RoleName \to Reference \\ \theta \in ThisReference & : Reference \end{array}$$

Expression evaluation $\Sigma, \Delta, \theta \vdash Expr \Downarrow \{|Value|\}$

$$\frac{c \text{ is constant}}{c \Downarrow \{|\, c \,|\}} \text{ [Const]} \qquad \frac{e \Downarrow \emptyset}{\text{sum}(e) \Downarrow \{|\, 0 \,|\}} \text{ [Sum]}$$

$$\frac{}{\theta \vdash \text{this} \Downarrow \{|\, \theta \,|\}} \text{ [This]} \qquad \frac{e \Downarrow V}{\text{count}(e) \Downarrow \{|\, |V| \,|\}} \text{ [Count]}$$

$$\frac{\oplus \in \{+, -, *, \&\&, ||, >, >=, <, <=, ==, !=\} \quad e_1 \Downarrow V_1 \quad e_2 \Downarrow V_2 \quad V_3 = \{|\, v_1 \oplus v_2 \mid v_1 \in V_1,\ v_2 \in V_2 \,|\}}{e_1 \oplus e_2 \Downarrow V_3} \text{ [BinOp]}$$

$$\frac{e_1 \Downarrow V_1 \quad e_2 \Downarrow V_2}{e_1 <+ e_2 \Downarrow (V_1 \,!= \emptyset)\ ?\ V_1 : V_2} \text{ [Choice]}$$

$$\frac{e_1 \Downarrow V_1 \quad e_2 \Downarrow V_2}{e_1 +\!+ e_2 \Downarrow V_1 \cup V_2} \text{ [Concat]}$$

$$\frac{e_1 \Downarrow V_1 \quad e_2 \Downarrow V_2 \quad \oplus \in \{/, \%\} \quad V_3 = \{|\, v_1 \oplus v_2 \mid v_2 \,!= 0,\ v_1 \in V_1,\ v_2 \in V_2 \,|\}}{e_1 \oplus e_2 \Downarrow V_3} \text{ [Div]}$$

$$\frac{e \Downarrow V \quad e : t \quad \mathcal{A}_t(a) = (\_, \_, \text{nil})}{\Sigma \vdash e\ .\ a \Downarrow \{|\, \Sigma(v)(a) \mid v \in V \,|\}} \text{ [Attr]}$$

$$\frac{e \Downarrow V \quad e : t \quad \mathcal{A}_t(a) = (\_, \_, e_2) \quad V_2 = \{|\, v_2 \mid (\theta' \vdash e_2 \Downarrow \{v_2\}),\ \theta' \in V \,|\}}{e\ .\ a \Downarrow V_2} \text{ [At2]}$$

$$\frac{e \Downarrow V}{!\, e \Downarrow \{|\, \neg\, v \mid v \in V \,|\}} \text{ [Not]}$$

$$\frac{e_1 \Downarrow V_1 \quad e_2 \Downarrow V_2 \quad e_3 \Downarrow V_3 \quad V_4 = \{|v_1 ?\, v_2 : v_3 \mid v_1 \in V_1, v_2 \in V_2, v_3 \in V_3 \,|\}}{e_1\ ?\ e_2 : e_3 \Downarrow V_4} \text{ [Cond]}$$

$$\frac{e \Downarrow V \quad e : t}{\Delta \vdash e\ .\ r \Downarrow \{|\, \Delta(t, v, r) \mid v \in V \,|\}} \text{ [RoleNav]}$$

$$\frac{f \in \{\text{avg}, \text{min}, \text{max}, \text{conj}, \text{disj}, \text{sum}\} \quad e \Downarrow V \quad |V| \geq 1}{f(e) \Downarrow \{|\, f(V) \,|\}} \text{ [Aggr]}$$

$$\frac{e \Downarrow V \quad e : t \quad \mathcal{I}_t(i) = (t, r, \text{nil}) \quad V_2 = \{|\, v_2 \mid \Delta(t, v_2, r) = v,\ v \in V \,|\}}{\Delta \vdash e\ .\ i \Downarrow V_2} \text{ [InvNav]}$$

$$\frac{f \in \{\text{avg}, \text{min}, \text{max}, \text{conj}, \text{disj}\} \quad e \Downarrow \emptyset}{f(e) \Downarrow \emptyset} \text{ [Aggr2]}$$

Program evaluation $\Sigma, \Delta \vdash Expr \Downarrow \{|Value|\}$

$$\frac{p = (\mathcal{E}, x) \quad \theta' = \bot \quad \Sigma, \Delta, \theta' \vdash x \Downarrow v}{\Sigma, \Delta \vdash p \Downarrow v} \text{ [Program]}$$

Figure 2.18 Evaluation rules (Big Step SOS). "{| |}" is bag notation [Buneman et al., 1994]. Stores are omitted if not used in rules.

2.6.2 *Store well-formedness*

Figure 2.17 describes what it means means for these stores to be well-formed. The entity store is well-formed if all the entities in it are well-formed. An entity is well-formed if (1) all records in its attribute store are well-formed, (2) all its required, non-derived attributes have been set (3) all its roles have a value and (4) the number of relation records, that point to it for a certain role that he plays, is within the multiplicity range specified for that role.

An attribute record is well-formed if it has a value of the correct type.

The relation store is well-formed if all its records are well-formed. A relation record is well-formed if its references point to entities. Finally the this-reference is well-formed if it points to an entity. We assume a well-formed entity and relation stores for evaluation.

2.6.3 *Evaluation rules*

All the evaluation rules have a specific form: they operate on bags. Expressions can return any number of values, modeling this with bags is a natural choice. A nice example of this is the rule for binary operations (BinOp). The left and right expressions evaluate to a bag of values, the Cartesian product of these bags is taken and on each pair of values the operator is applied. For single values a normal computation is performed, for maybe values a maybe computation and for many values a Cartesian product computation. Most evaluation rules follow this pattern.

Aggregation operations are defined for at least a single value (Aggr) and for empty lists there is predefined behaviour (Aggr2 and Sum). Choice returns the value of the left expression, if it has at least one value, otherwise the value of the right expression. Concat combines all values, regardless of how many there are. Attributes can either be normal or have a derivation expression. For normal attributes a lookup is done in the attribute map of each entity passed into the expression (Attr). The lookup of unset attributes fails, but these are filtered out. Derivations behave like a method call without arguments (At2). Navigation works differently for navigating through a role or through an inverse. Navigating by role does a simple map lookup for each value (RoleNav). Navigating by inverse does a reverse map lookup on the role it is the inverse of (InvNav). Finally the program executes the main expression with the stores.

2.7 RELATED WORK

Our work builds on research in different fields: language constructs for relations, navigating and querying relations and multiplicities. Specific differences with our work are highlighted per article.

Languages with first-class relations The Rumer language by Balzer has first-class relations [Balzer, 2011; Balzer et al., 2007]. It features first-class relations with named roles and queries. Rumer provides reactive queries as well as imperative code. It has cardinalities specified in constraints and implements binary relationships. Our approach differs in the fact that our modeling language does not support imperative code, multiplicities are part of the type system and we implement relations of all degrees.

Classages is a language that also features relations [Liu and Smith, 2005]. Classages is targeted at modelling the interactions and interaction life span between objects. It features static and dynamic relations, bidirectional relations and multiplicities. Our approach has in common that it has bidirectional relations but we are focused on modeling data instead of interactions.

Pearce and Noble extended Java with first-class relationships using aspects [Pearce and Noble, 2006]. Relations are modeled as external tuples and objects are agnostic to relations they are in. Their approach to behavioural changes of objects based on their relations should be implemented by aspects, externally. Our approach is the opposite, entities know what relations they participate in. This allows specifying relation dependent behaviour in derivations.

RelJ is first-class relationship extension to Java by Biermann and Wren [Bierman and Wren, 2005; Wren, 2007]. In their approach they support relationships as first-class citizens. The relations are also modeled as tuples, where the roles have a position in the tuple but no name. In our approach the roles are named and unordered; allowing navigation based on roles. Their relations are binary and one-directional. In the technical report they also sketch an extension with named roles [Bierman and Wren, 2005]. In this sketched extension relations can have any arity and support bidirectional navigation.

Nelson implemented first-class relationships in Java [Nelson et al., 2008]. This is a library and not a language extension. Mutable sets of tuples are used as first-class constructs to model relations. Without specific language constructs this approach does not supply additional semantics for relations and thus cannot provide additional static type checking.

Languages with non first-class relations In 1987 Rumbaugh was the first to add relations to a language [Rumbaugh, 1987]. His approach is pre-processor based and dynamic. It does not have relations as first-class citizens.

In 1991 a relationship mechanism for a Strongly Typed Object-Oriented Database Programming language introduced statically typed relations as part of a language [Albano et al., 1991]. The paper explains the data model definition and transactions. It does however not explain in detail how querying or navigation is done.

WebDSL introduced inverse properties which inspired the inverses [Visser, 2007]. Refer to Section 2.3 for details.

Queries of relations in object-oriented languages The Java Query Language (JQL) adds queries to Java [Willis et al., 2006]. There is no additional support for relations, so navigation uses value-based joins like in SQL. LINQ also uses value-based joins [Meijer et al., 2006]. These approaches are in the left column of the design space (Section 2.3). In contrast, our navigation is based on the role names of relations.

Multiplicities in programming languages In Content over Container: Object-Oriented Programming with multiplicities Steimann adds multiplicity annotations to Java in order to remove the Collection containers [Steimann, 2013]. Refer to Section 2.2.2 for details.

Lerner et al. introduced a static type system for jQuery to track multiplicities of selector expressions and sums [Lerner et al., 2013]. We support navigation expressions, operators, conditionals, and aggregation operations (including sums). In their type system the types are nested in multiplicities similar to monads. In contrast, our types and multiplicities are orthogonal.

In the array programming language Remora operations are automatically lifted based on multiplicity [Slepak et al., 2014]. Our work is not aimed at array programming, but uses the same automatic lifting for expressions over primitive values.

Finally the ideas for this chapter were presented in the ACM Student Research Competition [Harkes, 2014]. The design space analysis and formal semantics of the language are new to this chapter. Also the syntax changed as a result of the design-space analysis.

2.8 CONCLUSION

Unification and generalization of relations led to a new data modeling and navigation language. This goes hand in hand with native multiplicities. Both the relations aspect and the native multiplicities aspect lead to more a more concise definition and navigation of relationships; removing maintenance of reference consistency, removing collection classes and providing single identifier navigation by inverses and shortcuts.

Future work We would like to add more aspects orthogonally to the type system. Our first candidates are ordered/unordered and unique/duplicates. It is worth exploring how well different aspects can be modelled orthogonally in a type system.

Also we would like to extend our language to provide type-and-multiplicity-safe operations on data. Adding or removing entities and relations might invalidate the multiplicity constraints on relations. We would like to catch these potential errors by static analysis and indicate to the programmer that he should catch that situation. The goal is to make sure that multiplicity-safe operations will never trigger runtime errors because a multiplicity constraint for a relation is violated. We would like to explore if we can ensure correct multiplicities at runtime statically.

Postscript: Relations Language

The relations language introduced first-class, n-ary, and bidirectional relations; native multiplicities; and concise navigation. However, in practice first-class, and n-ary relations have not seen a lot of use. Moreover, the multiplicity-safe semantics presented in this chapter are read-only, leaving multiplicity safety for update operations unspecified. In this postscript we discuss these two issues.

FIRST-CLASS CITIZEN AND N-ARY RELATIONS

When this chapter was written, the runtime of the relations language was in-memory. Soon after this chapter was written, we targeted WebDSL as backend for the language (as described in the next chapter). This changed the runtime to an Object-Relational Mapper (ORM) instead of plain in-memory objects. First-class citizen and n-ary relations require extra tables in a relational database, which is undesirable. Moreover, existing WebDSL applications already modeled first-class or n-ary relations with 'relation objects' explicitly. Migrating these existing data models also was undesirable. For these reasons first-class citizen, and n-ary relations have not seen a lot of use in practice.

Later we did not evolve first-class citizen and n-ary relations like we did with bidirectional relations. In Chapter 4 we introduce derived relations. These derived relations are bidirectional, but not first-class citizen or n-ary. First-class citizen or n-ary derived relations would mean deriving new objects as derived values. This would make IceDust higher-order (similar to higher-order attribute grammars) and Turing-complete. Higher-order attribute grammars have bad incremental performance as their runtime involves deep equality checks of derived objects. Thus, we kept IceDust first-order which means first-class citizen and n-ary cannot be derived like bidirectional relations.

MULTIPLICITY-SAFE UPDATE OPERATIONS

The multiplicity-safe semantics in this chapter are read-only. It seemed obvious to extend multiplicity-safety to update operations. This would be similar to static multiplicity checks with the bounded model checking language Alloy [Jackson, 2006]. However, we could not express multiplicity-safety in a simple type system and abandoned that research direction. In retrospect it might not be possible to express multiplicity-safety of update operations in a simple type-system at all: Alloy uses bounded model checking to guarantee multiplicity-safety. Instead of statically guaranteeing multiplicity-safety IceDust uses transactions to guarantee multiplicity-safety at runtime. These semantics are described in Chapter 4.

IceDust

Incremental and Eventual Computation of Derived Values in Persistent Object Graphs [1]

3

Derived values are values calculated from base values. They can be expressed in object-oriented languages by means of getters calculating the derived value, and in relational or logic databases by means of (materialized) views. However, switching to a different calculation strategy (for example caching) in object-oriented programming requires invasive code changes, and the databases limit expressiveness by disallowing recursive aggregation.

In this chapter, we present IceDust, a data modeling language for expressing derived attribute values without committing to a calculation strategy. IceDust provides three strategies for calculating derived values in persistent object graphs: Calculate-on-Read, Calculate-on-Write, and Calculate-Eventually. We have developed a path-based abstract interpretation that provides static dependency analysis to generate code for these strategies. Benchmarks show that different strategies perform better in different scenarios. In addition we have conducted a case study that suggests that derived value calculations of systems used in practice can be expressed in IceDust.

3.1 INTRODUCTION

Derived values are values calculated from base values (provided by users). When a base value changes, the derived values depending on it should change accordingly. Hence, the important events for interacting with derived values are writes to base values and reads of derived values. This specification of derived values leaves room for multiple strategies for calculating derived values. Derived values can be calculated when they are read or they can be cached and updated when the underlying base values change. The performance of these strategies depends on characteristics of the data model and usage scenarios. When neither of these calculation strategies provides acceptable performance, updates can be postponed, temporarily allowing reads to return outdated derived values.

Object-oriented programming languages express derived values through getters containing code that calculates a derived value, implying that the derived value is recalculated each time it is read. Switching to calculating the derived value when an underlying value changes, or switching to eventually calculating the derived value, requires invasive code changes. By contrast,

[1] This chapter has appeared as Harkes, D. C., Groenewegen, D. M., and Visser, E. (2016). Icedust: Incremental and eventual computation of derived values in persistent object graphs. In Krishnamurthi, S. and Lerner, B. S., editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

most relational databases allow easy switching between calculate-on-read and calculate-on-write as they support both materialized and non-materialized views for calculating derived values. However, relational databases only provide limited expressiveness for recursion, and do not support eventual calculation of derived values. Datalog provides more expressiveness than relational database views, but also limits recursion, and does not support eventual calculation of derived values.

This chapter presents the IceDust language, which supports definition of attributes with derived values without committing to a calculation strategy. The compiler provides three different implementation strategies for calculating derived values: (1) Calculate-on-Read, which calculates the derived value every time it is read, (2) Calculate-on-Change, which maintains a cache incrementally by calculating the derived value every time an underlying value is changed, and (3) Calculate-Eventually, which schedules calculations of derived values, and thus sacrifices consistency temporarily. All these strategies allow unrestricted recursion, but do not provide termination guarantees. In particular, our contributions are:

- The IceDust language for data modeling with derived values (Section 3.2)
- A formal analysis of the dependencies in IceDust programs (Section 3.3)
- Three calculation strategies to satisfy different non-functional requirements (Section 3.4)
- Benchmarks showing the performance differences between the strategies (Section 3.5)
- A case study of migrating a custom eventual calculation system to IceDust (Section 3.6)

3.2 DECLARATIVE DATA MODELING WITH DERIVED VALUES

This section discusses three issues of data modeling with derived values in object-oriented programming languages and shows how data modeling in IceDust addresses these issues and leads to concise specifications. As running example we use (an aspect of) a learning management system in which students solve assignments. Figure 3.1 shows a Java implementation with classes `Assignment` and `Question`, where assignment represents a collection of questions and its progress is the average of the progress on the individual questions.

3.2.1 *Bidirectional Relations*

Object-oriented languages model bidirectional relations as properties in the classes on both sides of the relation. Keeping these properties consistent requires code that has to be repeated for every bidirectional relation. Figure 3.1 includes five methods concerned with keeping relation `Assignment-Question` consistent on updates: `setAssignment`, `addQuestion`, `removeQuestion`, `_-`

```
public class Assignment {
  public Double getAverageProgress() {
    return calculateAverageProgress();
  }
  public Double calculateAverageProgress() {
    Stream<Double> progresss =
      questions.stream().map(q->q.getProgress()).filter(p -> p!=null);
    OptionalDouble average = progresss.mapToDouble(p -> p).average();
    return average.isPresent() ? average.getAsDouble() : null;
  }
  private Collection<Question> questions;
  public Collection<Question> getQuestions(){
    return new HashSet<>(questions);
  }
  public void addQuestion(Question q) {     q.setAssignment(this);  }
  public void removeQuestion(Question q) {  q.setAssignment(null);  }
  protected void _addQ(Question q) {        questions.add(q);  }
  protected void _remQ(Question q) {        questions.remove(q);  }
}
public class Question {
  private Assignment assignment;
  public Assignment getAssignment() {  return assignment;  }
  public void setAssignment(Assignment a) {
    if(assignment != null) { assignment._remQ(this); }
    if(a != null) { a._addQ(this); }
    assignment = a;
  }
  private Double progress;
  public Double getProgress() {  return progress;  }
  public void setProgress(Double p) {  progress = p;  }
}
```

Figure 3.1 Object-oriented assignment system (Calculate-on-Read implementation strategy).

`addQ`, and `_remQ`. This pattern is identical for all one-to-many relations, but cannot be abstracted over in an object-oriented language. To avoid such boilerplate code, IceDust supports *bidirectional relations* as a language feature:

```
entity Assignment { }
entity Question { }
relation Assignment.questions * <-> 1 Question.assignment
```

These bidirectional relations are named on both sides of the relation, inspired by Object-Role Modeling [Halpin, 2006]. The IceDust compiler keeps both sides of the association consistent without additional boilerplate code.

3.2.2 *Native Multiplicities*

Explicit collections and possible null values in object-oriented languages lead to boilerplate code to deal with the cardinalities of values returned by an expression. Operators in object-oriented languages are defined for operands with a cardinality of exactly one. Safely applying an operator to a nullable operand, requires a `null`-check. Applying an operator to a collection of val-

ues, requires lifting it to a map. For example, accessing the progress of each individual question in Figure 3.1 is encoded as

```
questions.stream().map(q -> q.getProgress()).filter(p -> p != null)
```

IceDust adopts *native multiplicities* [Harkes and Visser, 2014], delegating the handling of the cardinality of values returned by an expression to the language. For example, retrieving the progress for all questions is simply a projection:

```
questions.progress
```

Language constructs to get expressions of cardinality exactly one, such as `map`, `filter`, and `!= null`, are no longer required, as the type system knows how many values an expression returns (multiplicity denoted by ~, where * is [0,n), + is [1,n), ? is [0,1], and 1 is [1,1]):

```
mathAssignment                          // : Assignment ~ 1
mathAssignment.questions                // : Question   ~ *
mathAssignment.questions.progress       // : Float      ~ *
avg(mathAssignment.questions.progress) // : Float      ~ ?
```

Sometimes it is still necessary to reflect explicitly on the cardinality of a value. To that end one can use the `count` operator, for example, for counting the number of questions:

```
count(questions)
```

Reflection on the cardinality of values is also often used to select an alternative if no value is present. For specifying alternatives the choice operator (<+) can be used:

```
input <+ myDefault //if (count(input) > 0) input else myDefault
```

3.2.3 *Derived Value Attributes*

Last but not least, object-oriented languages force early decisions on the implementation strategy for calculating derived values. In an object-oriented language, a derived value calculation can be expressed with a method that computes the value. However, this encodes a Calculate-on-Read implementation strategy. For cheap calculations or calculations that are done infrequently that may be fine. But for others, it may be necessary to cache the calculated value. Such an alternative computation strategy requires an invasive redefinition of the implementation. For example, Figure 3.2 implements a caching strategy for the `getAverageProgress` computation of Figure 3.1. Instead of computing the average on read, it is computed on writes of `progress` and `questions`. For this example, the impact of the change was relatively minor because in Figure 3.1 we had already factored `calculateAverageProgress` into a separate method. However, in real code the impact is typically non-trivial. In particular, because the introduction of a cached value requires taking into account all of its dependencies in order to trigger recomputation on any change that affects it. For example, `averageProgress` depends on

```
//Take all Code from Calculate-on-Read and add/change the following:
public class Assignment {
  private Double cachedAvgProgress;
  public Double getAverageProgress() {  return cachedAvgProgress;  }
  public void updateAvgProgress(){
    cachedAvgProgress=calculateAverageProgress();
  }
  protected void _addQ(Question q){
    questions.add(q);    updateAvgProgress();
  }
  protected void _remQ(Question q){
    questions.remove(q); updateAvgProgress();
  }
}
public class Question {
  public void setProgress(Double p){
    progress=p; assignment.updateAvgProgress();
  }
}
```

Figure 3.2 Object-oriented assignment system (Calculate-on-Write implementation strategy).

`progress` and `questions`. Thus, `setProgress`, `_addQ`, and `_remQ` all need to trigger recalculation of `averageProgress`.

IceDust provides *derived value attributes* for declarative specification of the value of attributes in terms of other attributes without committing to an implementation strategy:

```
entity Assignment{  avgProgress : Float? = avg(question.progress)  }
```

This separation of concerns enables focusing on specification of the logic of the derived value. The derived value expression specifies what the value of the attribute should be. Derived value attributes in IceDust support recursive definitions, including recursive aggregation (which is not supported in materialized views or stratified Datalog):

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

3.2.4 *Language Definition*

We have combined the ideas for improving data modeling by means of bidirectional relations, native multiplicities, and derived value attributes in the design of the experimental IceDust language. In order to embed IceDust data models in full fledged web applications the compiler generates code in the WebDSL programming language [Visser, 2007].

The design of IceDust was heavily influenced by previous work on relations as a first-class language construct. From Rumer [Balzer, 2011] and RelJ [Bierman and Wren, 2005] we adopt the restriction to binary, bidirectional relations. From the Relations language [Harkes and Visser, 2014] we adopt the syntax of declarations and property access, integrating multiplicities in relations. Multiplicities derive from the work of Steimann [Steimann, 2013, 2015],

```
Program ::= model Entity* Relation*
Entity ::= entity E { Attribute* }
Relation ::= relation E.r m <-> m E.r
Attribute ::= a : T m
            | a : T m = e
            | a : T m = e (default)
T ∈ PrimitiveType ::= Boolean | Int | Float | Datetime | String
m ∈ Multiplicity ::= ? | 1 | * | * (ordered) | + | + (ordered)
e ∈ Expr ::= f (e) | e1 ⊕ e2 | !e | e1 ? e2 : e3 | e.a | e.r | e as T | this | Literal
Literal ::= true | false | null | int | float | datetime | string
f ∈ AggrOp ::= min | max | avg | sum | concat | count | conj | disj
⊕ ∈ BinOp ::= + | - | * | / | % | && | || | > | >= | < | <= | == | =! | <+ | ++
```

Figure 3.3 Syntax of the IceDust data modeling language

which extends an object-oriented language with multiplicity annotations to support uniform treatment of values of different cardinality and avoids the boilerplate code required to support different multiplicities. We adopt the integration of such multiplicities in the type system (dubbed native multiplicities) of the Relations language [Harkes and Visser, 2014].

Figure 3.3 defines the grammar of IceDust. `E`, `a` and `r` are entity, attribute and relation names respectively. An IceDust program consists of entities and relations. Entities contain three kinds of attributes: 'normal' attributes (`a : T m`), derived value attributes (`a : T m = e`), and default value attributes (`a : T m = e (default)`). Users can set the value of 'normal' attributes and read the value later. Users cannot set the value of derived value attributes, but they can read the value calculated with expression `e`. Finally, users can set the value of default value attributes and read the value later, but they can also set the value to null (or not set it all) and read the value calculated by `e`. Attributes are limited to primitive types, as an entity type would create a unidirectional relation (which would give problems in the dependency analysis). A relation defines a bidirectional relation with a name and multiplicity on both sides. The domain of the expression language is primitive types (Boolean, Int, Float, Datetime and String) and objects. The language covers object graph navigation and calculations over the primitive types. Note the aggregation operations over primitive types to deal with multiplicities `*` and `+`. The expression language is expressive enough to specify derived values, and simple enough to allow multiple implementation strategies.

The type system of IceDust is mostly concerned with native multiplicities. A type in IceDust is a tuple of two lattice values (Figure 3.4). The primitive types, the declared entities, the `null` and error type form a lattice. Multiplicity and ordering form another lattice. During derived value calculation all values are read-only in IceDust. A value which is lower or equal in both lattices can be used in a place where a certain type, multiplicity, and ordering is expected. For example, a `Float?` can be supplied where a `Float*` is expected.

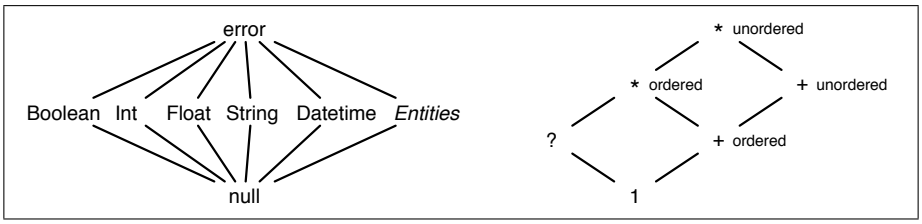


Figure 3.4 IceDust's type lattice (left), and multiplicity and ordering lattice (right)

3.3 DEPENDENCY AND DATA FLOW ANALYSIS

IceDust specifies the value of attributes in terms of other attributes. These definitions are declarative in the sense that they abstract from the implementation strategy used to calculate the values. In the next section we define three implementation strategies for the calculation of attribute values: Calculate-on-Read, Calculate-on-Write, and Calculate-Eventually. The latter two strategies require dependency and data flow information. In this section we define the computation of dependencies between attributes by means of a path-based abstract interpretation of expressions. Since IceDust does not have statements, data flow coincides with control flow, and the data flow relation is the inverse of the dependency relation. The static dependency and data flow analysis is performed in three steps: (1) compute attribute dependencies by means of path-based abstract interpretation, (2) reverse the dependencies to construct the data flow relation, and (3) organize the data flow in a graph and extract strongly connected components with a topological ordering.

3.3.1 *Example*

To illustrate the analysis we use a more complex version of the learning management system (Figure 3.5). This example features a tree of assignments, and grade calculation logic for submissions by students to these assignments. `Assignment`s are structured in a tree through the relation `parent-children`. A `Submission` represents the solution created for an assignment by a student. Leaf submissions are graded by assigning a grade to the `grade` attribute (overriding the default value), while the grades of non-leaf submissions depend (indirectly) on the grades of their child submissions:

```
grade      : Float?  = if(childPass) childGrade else null (default)
pass       : Boolean = grade >= (assignment.minimum<+0.0) <+ false
childGrade : Float?  = avg(children.grade)
childPass  : Boolean = conj(children.pass)
```

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. The `minimum` for an assignment is optional, without `minimum` the grade should be higher than or equal to 0.0, which is always true. Submissions are (one of) the `best` of an assignment when their `grade` equals the highest grade. Finally, every

```
entity Assignment {
  name       : String
  question   : String
  minimum    : Float?
  avgGrade   : Float? = avg(submissions.grade)
  passPerc   : Float? = sum(submissions.passInt)*100/count(submissions)
}
entity Student {
  name       : String
}
entity Submission {
  name       : String  = assignment.name + " " + student.name
  answer     : String?

  grade      : Float?  = if(childPass) childGrade else null (default)
  pass       : Boolean = grade >= (assignment.minimum<+0.0) <+ false
  childGrade: Float?  = avg(children.grade)
  childPass : Boolean = conj(children.pass)

  passInt    : Int     = if(pass) 1 else 0
  best       : Boolean = grade == max(assignment.submissions.grade)
                         <+ false
}
relation Assignment.parent     ? <-> * Assignment.children
relation Submission.parent     ? <-> * Submission.children
relation Submission.student    1 <-> * Student.submissions
relation Submission.assignment 1 <-> * Assignment.submissions
```

Figure 3.5 Example program for dependency analysis

assignment has an average grade and pass percentage. This example is interesting for dependency analysis as it features mutually recursive definitions of `grade`, `pass`, `childGrade` and `childPass` through the `parent-child` relation of `Submission`.

3.3.2 *Step 1: Dependencies*

The dependencies of an attribute are all the attributes and relations that are needed to compute the derived value of that attribute. The dependencies are reachable from the entity of the attribute via a path. A dependency is denoted by $(Ent.Attr \leftarrow \pi)$, where $Ent.Attr$ is the attribute and π is the path to an attribute or relation.

Computing the dependencies requires extracting paths from expressions defining derived values. The *path-based abstract interpretation* relation (Figure 3.6) defines the dependency paths of an expression. We use the notation $Expr \searrow \{\pi\}\{\rho\}$, where $Expr$ is the expression that is abstractly interpreted, and $\{\pi\}$ and $\{\rho\}$ are the sets of paths defined by the abstract interpretation. The paths in $\{\pi\}$ are extensible, while the paths in $\{\rho\}$ are not. All paths start with `this` [This] or with object graph navigation [NavStart]. When navigating the object graph by means of `e.attrOrRel` all dependency paths $\{\pi\}$ in `e` are extended with `attrOrRel` [Nav]. The `if` [If] only allows extension of paths in the second and third operand, so Π_1 is passed to $\{\rho\}$. Operators

Path-based abstract interpretation $\boxed{Expr \searrow \{\pi\}\{\rho\}}$

$$\frac{}{\text{this} \searrow \{\text{this}\}\{\}} \text{[This]} \qquad \frac{e \in Literal}{e \searrow \{\}\{\}} \text{[Literal]}$$

$$\frac{}{attrOrRel \searrow \{attrOrRel\}\{\}} \text{[NavStart]} \qquad \frac{e \searrow \Pi\ \mathrm{P}}{!\, e \searrow \Pi\ \mathrm{P}} \text{[Not]}$$

$$\frac{e \searrow \Pi\ \mathrm{P}}{e\ .\ attrOrRel \searrow \{\pi\ .\ attrOrRel \mid \pi \in \Pi\}\ \mathrm{P}} \text{[Nav]} \qquad \frac{T \in PrimitiveType \quad e \searrow \Pi\ \mathrm{P}}{e \text{ as } T \searrow \Pi\ \mathrm{P}} \text{[Cast]}$$

$$\frac{\oplus \in BinOp \quad e_1 \searrow \Pi_1\ \mathrm{P}_1 \quad e_2 \searrow \Pi_2\ \mathrm{P}_2}{e_1 \oplus e_2 \searrow \Pi_1 \cup \Pi_2 \quad \mathrm{P}_1 \cup \mathrm{P}_2} \text{[Op]} \qquad \frac{f \in AggrOp \quad e \searrow \Pi\ \mathrm{P}}{f(e) \searrow \Pi\ \mathrm{P}} \text{[Aggr]}$$

$$\frac{e_1 \searrow \Pi_1\ \mathrm{P}_1 \quad e_2 \searrow \Pi_2\ \mathrm{P}_2 \quad e_3 \searrow \Pi_3\ \mathrm{P}_3}{e_1\,?\,e_2 : e_3 \searrow \Pi_2 \cup \Pi_3 \quad \Pi_1 \cup \mathrm{P}_1 \cup \mathrm{P}_2 \cup \mathrm{P}_3} \text{[If]}$$

Dependencies $\boxed{Attr|Ent|Prog \searrow\!\searrow \{(Ent.Attr \leftarrow \pi)\}}$

$$\frac{e \searrow \Pi\ \mathrm{P} \quad E = \text{entity-of}(attr) \quad \Pi_2 = \bigcup \{\text{trans-pref}(\text{remove-this}(\pi)) \mid \pi \in \Pi \cup \mathrm{P}\}}{attr : T\ m\ \{= e, = e\ (\text{default})\} \searrow\!\searrow \{(E.attr \leftarrow \pi) \mid \pi \in \Pi_2\}} \text{[Att]}$$

$$\frac{}{\text{entity } t\ \{a^*\} \searrow\!\searrow \bigcup \{dep \mid a \searrow\!\searrow dep,\ a \in a^*\}} \text{[Ent]}$$

$$\frac{}{\text{model } E^*\ R^* \searrow\!\searrow \bigcup \{dep \mid E \searrow\!\searrow dep,\ E \in E^*\}} \text{[Prog]}$$

remove-this(this . π) = π
remove-this($attrOrRel$. π) = $attrOrRel$. π
trans-pref(π . $attrOrRel$) = { π . $attrOrRel$ } $\cup$ trans-pref(π)
trans-pref($attrOrRel$) = { $attrOrRel$ }

Figure 3.6 Dependency analysis step 1: path-based abstract interpretation

Dependency inversion $\boxed{(Ent.Attr \leftarrow \pi) \nearrow (Ent.AttrOrRel \rightarrow \pi)}$

$$\frac{E_2 = \text{entity-of}(attrOrRel)}{(E\ .\ attr \leftarrow \pi\ .\ attrOrRel) \nearrow (E_2\ .\ attrOrRel \rightarrow \text{inv-path}(\pi)\ .\ attr)} \text{[InvDep]}$$

inv-path(π . $attrOrRel$) = $attrOrRel^{-1}$. inv-path(π)
inv-path($attrOrRel$) = $attrOrRel^{-1}$
inv-path(null) = null

Data flow $\boxed{Prog \nearrow\!\nearrow \{(Ent.AttrOrRel \rightarrow \pi)\}}$

$$\frac{\text{model } E^*\ R^* \searrow\!\searrow Dep}{\text{model } E^*\ R^* \nearrow\!\nearrow \{df \mid dep \nearrow df,\ dep \in Dep\}} \text{[Prog]}$$

Figure 3.7 Dependency analysis step 2: data flow

Data flow graph $Prog \nearrow\!\!\nearrow\!\!\nearrow (\{AttrOrRel\}, \{(AttrOrRel, Attr)\})$

$$\frac{\text{model } E^* \; R^* \nearrow\!\!\nearrow DFlow \quad E = \{(x,y) \mid (Ent.x \rightarrow \pi.y) \in DFlow\} \quad V = \{x \mid (x,y) \in E\} \cup \{y \mid (x,y) \in E\}}{\text{model } E^* \; R^* \nearrow\!\!\nearrow\!\!\nearrow (V, E)} \text{ [Prog]}$$

Figure 3.8 Dependency analysis step 3: data flow graph

with multiple operands take the union of the paths of their operands [Op], unary operators pass on paths [Not,Cast,Aggr], and literals do not contain any paths at all [Literal]. Path-based abstract interpretation of the expression defining `pass`

```
grade >= (assignment.minimum <+ 0.0) <+ false
```

produces a set containing the following paths:

```
grade
assignment.minimum
```

The *dependencies* relation (Figure 3.6) defines the dependencies of an attribute, entity and program. We write $Attr|Ent|Prog \searrow\!\searrow \{(Ent.Attr \leftarrow \pi)\}$, where $Attr|Ent|Prog$ is an attribute, entity or program, and $\{(Ent.Attr \leftarrow \pi)\}$ is a set of dependencies. When an attribute depends on a value at the end of a path, it also depends on the relations en route. So, the rule for attributes [Att] takes the transitive prefix of the paths of its expression. As paths are concatenated later, and a `this` keyword in the middle would produce an invalid path, the `this` is removed from paths. As an example, the dependencies of `pass` are:

```
(Submission.pass ← grade)
(Submission.pass ← assignment.minimum)
(Submission.pass ← assignment)
```

The dependencies in for the individual attributes together constitute the dependencies for a full program [Ent,Prog].

3.3.3 *Step 2: Data Flow*

The data flow of an attribute or relation is the set of all the attributes that depend on it to compute their derived value. The data flow relation is the inverse of the dependency relation. We use the notation $(Ent.AttrOrRel \rightarrow \pi)$ to denote the data flow relation from the source, $Ent.AttrOrRel$, to the target, the end of the path π.

The *dependency inversion* relation, $(Ent.Attr \leftarrow \pi) \nearrow (Ent.AttrOrRel \rightarrow \pi)$, in Figure 3.7 defines the inverse of a dependency. A dependency is inversed by swapping source and target, and inverting the path π to get the path from target to source. The function `inv-path(`π`)` inverts the names in on path, and inverts their order. Name inversion is selecting the name on the opposing side of a relation; all relations in IceDust are bidirectional, and have names on both sides. All names in π can be inverted because they are relations. (π is

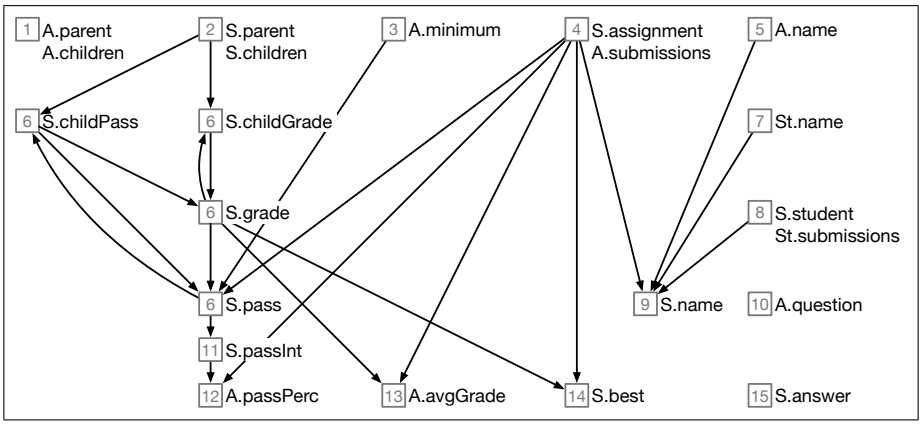


Figure 3.9 Step 3 example: strongly connected components and topological ordering in data flow.

the prefix of a full path, and only the last name of a path can be an attribute.) If the dependencies of the attribute `pass` are inversed the resulting data flow is:

```
(Submission.grade      → pass)
(Assignment.minimum    → submissions.pass)
(Submission.assignment → pass)
```

3.3.4 *Step 3: Data Flow Graph*

The data flow graph relation $Prog \nearrow\!\!\!\nearrow\!\!\!\nearrow (V, E)$ in Figure 3.8, defines a data flow graph in terms of the data flow relation. The nodes in the graph are attributes and relations in an IceDust program. The edges (x, y) in the graph are $(AttrOrRel, Attr)$ from the data flow relation $(Ent.AttrOrRel \rightarrow \pi.Attr)$. Using Tarjan's algorithm [Tarjan, 1972] we find strongly connected components and a topological ordering for the data flow. The strongly connected components correspond to recursive dependencies.

The data flow graph for our example application is shown in Figure 3.9. The attributes `grade`, `pass`, `childGrade`, and `childPass` mutually depend on each other, a cycle in the graph (group 6). (The data flow is not cyclic: data flows up the submission tree.) The `minimum` precedes group 6 in the topological ordering, as `pass` in group 6 depends on it but minimum itself depends on nothing. On the other hand, the `passPerc`, `averageGrade`, and `best` depend on the results in group 6. The derived `name` for submissions is disconnected from the grade calculation, as the name of the submission does not have anything to do with the grade calculation. Relations only flow to attributes, and not vice versa. In IceDust, relations cannot be derived. This limits the expressiveness of IceDust, but also avoids 'dynamic dependencies', dependencies that are discovered while computing derived values.

Topological ordering can be used to statically schedule the computation of derived values. This is used in stratified Datalog, where a topological sort of

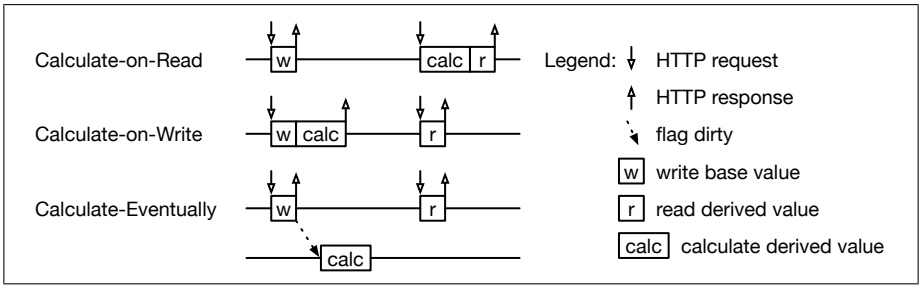


Figure 3.10 Thread activation diagrams for code generated by different implementation strategies.

the dependencies between rules is used to determine the order of computation [Apt et al., 1986; Green et al., 2013]. We will elaborate on computation scheduling, and on similarities with existing approaches, in later sections.

3.4 IMPLEMENTATION STRATEGIES

The declarative specification of derived values in IceDust allows deferring the decision about implementation strategy from implementation to compilation time, and allows switching strategies to realize different non-functional requirements without invasive code changes. In this section we present three implementation strategies: Calculate-on-Read, Calculate-on-Write, and Calculate-Eventually . For each of these we have a compilation scheme that specifies what code to generate for IceDust's concepts.

On a high level the difference between the generated code for the different implementation strategies is the point in time at which derived values are calculated. Figure 3.10 shows the differences by means of thread activation diagrams in response to incoming HTTP requests. The code generated by Calculate-on-Read 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 code generated by Calculate-on-Write calculates the derived values that depend on changed base values right away. Writes will be slow, but reads will be fast. The code generated by Calculate-Eventually schedules calculation of derived values on a separate thread. Writes and reads will be fast, but consistency is not guaranteed: possibly outdated derived values might be read.

3.4.1 *Compiling to WebDSL*

IceDust is used to specify the data model and derived values for web applications. Our compiler compiles IceDust specifications to the WebDSL web programming language [Visser, 2007], which is a high-level target language for the implementation of data models. WebDSL persists its data in a relational database. This provides (1) data safety in case of a power outage, (2) enables large data sets, and (3) enables concurrent data access for concur-

rent HTTP requests. WebDSL's data modeling language is close to IceDust; it features entities and attributes (including Calculate-on-Read derived values):

```
entity Assignment {
  name     : String
  avgGrade: Float := avg([s.grade | s  in subs where s.grade!=null])
}
```

Note the list comprehension syntax for applying a map to access the grade for each submission and filter on null values. WebDSL does not have bidirectional relations like IceDust, but it does have inverse properties:

```
entity Submission{ assignment : Assignment (inverse = submissions) }
entity Assignment{ submissions: Set<Submission>                    }
```

If a property is an inverse of another property, WebDSL keeps the values in the properties consistent. Our compiler targets these inverse properties for bidirectional relations, giving us the consistency of bidirectional relations for free.

With WebDSL already providing data persistence, large data sets, concurrency, Calculate-on-Read derived values, and inverse property consistency, our compilation schemes can focus on the essentials: default value behavior, multiplicities, bidirectional relations, and the Calculate-on-Write and Calculate-Eventually implementations for derived values.

The rest of this section describes the three implementation strategies in detail, using Morph][Huang and Smaragdakis, 2008]-style code generation templates for the compilation schemes. The templates use WebDSL (`black` with `purple keywords`) as target language and template-level control statements (*`blue italic`*) that iterate over entities, attributes, relations, and data flow edges (*`orange italic`*). We explain WebDSL code along the way, using callouts (for example: [1]) to refer to specific parts of generated WebDSL code.

3.4.2 *Calculate on Read*

Figure 3.11 defines the Calculate-on-Read compilation scheme. To translate IceDust with Calculate-on-Read to WebDSL we need to translate three IceDust features: (1) multiplicities, (2) default value attributes, and (3) bidirectional relations.

Multiplicities `?` and `1` are translated to WebDSL primitives, while multiplicities `*` or `+` are translated to lists. The getter for a normal attribute [2] (see Figure 3.11) is static for null-safety, it might be called on a null value, for example: `Assignment.get_passPerc(null)`. The getter is lifted to deal with a list of entities for which the attribute is referenced [10]. Attributes can only have multiplicity `?` or `1`, so there is no generation for multiplicity `*` or `+`. (List typed attributes would create overhead in WebDSL's mapping to the underlying database.)

Default value attributes are translated to two attributes [6,7] and one getter [9] in WebDSL. The first attribute [6] corresponds to the value possibly set by the user. The second attribute [7] corresponds to the default value expression. The getter [9] will return the user provided value, if any, and otherwise the default

value. When only [6] is used to write values, and only [9] is used to read values the default value attribute will have IceDust's semantics. WebDSL features no `private` attributes and methods, so this behavior cannot be encapsulated.

Bidirectional relations are translated to properties and inverse properties (which are kept consistent by WebDSL). The right-hand side of the relation is translated to a normal WebDSL property [12,14,16], and the left-hand side is translated to a property with an inverse [11,13,15]. Unordered to-many relations are translated to sets, while the ordered relations are translated to lists. It would suffice to translate them all to lists, but WebDSL's relational database mapping has more overhead for lists than for sets. Relation navigation is overloaded on multiplicity: navigate from single entity via a to-one relation [17], or via a to-many relation [19], and navigate from multiple entities via a to-one relation [18], or via a to-many relation [20].

Calculate on Read Properties

The compiled Calculate-on-Read programs have the following properties: (1) derived value reads are consistent, (2) transactions might fail, and (3) cyclic derived values cause a stack overflow exception at runtime.

Derived value consistency is based on database transactions. HTTP requests see all changes to base data from previous requests, and no changes to base data from concurrent requests. They compute the derived values, so these are consistent. The database performs optimistic locking, consequently transactions with concurrent edits to the same values are rejected. A cycle in the static dependency graph, such as group 6 in Figure 3.9, can admit a cyclic attribute value definition (for example a submission being a child of itself, and its grade being the average of its child grades). Such a cyclic derived value cannot be computed. The generated code will keep recursing into the getters until stack space is exhausted.

3.4.3 *Calculate on Write*

Figure 3.12 defines the Calculate-on-Write compilation scheme. The Calculate-on-Write compilation scheme builds on the Calculate-on-Read compilation scheme, only mentioning the new or changed WebDSL code. The general idea for Calculate-on-Write is caching all derived values, and incrementally maintaining the cached values on writes (like materialized views [Gupta and Mumick, 1995]). Updating a derived value can lead to having to update other derived values. This behavior is realized by dirty flagging (and updating) all dependent attributes on updating an attribute or relation (like push-based reactive programming [Nilsson et al., 2002]). To avoid unnecessary recomputation, updates are scheduled using the topological sort of the data flow graph (like stratified Datalog [Apt et al., 1986; Green et al., 2013]). So, to translate IceDust with Calculate-on-Write to WebDSL, we need to generate caches, dirty flagging, and recalculation.

Derived value caches store the derived values [22,25]. The properties containing the cached derived values are managed by code keeping track of dirty

```
for E in Entities
  entity E {
    for a : T m in E.attributes
      a : T (default=null)[1]

      static function get_a(e : E) : T {  return if(e == null) null else e.a; }[2]

    for a : T m = e1 in E.attributes
      a : T := calculate_a()[3]

      function calculate_a() : T {  return e1;  }[4]

      static function get_a(e : E) : T {  return if(e == null) null else e.a; }[5]

    for a : T m = e1 (default) in E.attributes
      a : T (default=null)[6]
      a_default : T := calculate_a()[7]

      function calculate_a() : T {  return e1;  }[8]

      static function get_a(e : E) : T {
        return if(e == null) null else if(e.a == null) e.a_default else e.a;
      }[9]

    for a : T m {, = e1, = e1 (default)} in E.attributes
      static function get_a(entities : [E]) : [T] {
        return [E.get_a(e) | e : E in entities where E.get_a(e) != null];
      }[10]

    for relation E.l {1,?} <-> m2 E2.r in Relations
      l : E2 (inverse=r)[11]
    for relation E2.r m2 <-> {1,?} E.l in Relations
      l : E2[12]
    for relation E.l {*,+} (unordered) <-> m2 E2.r in Relations
      l : {E2} (inverse=r)[13]
    for relation E2.r m2 <-> {*,+} (unordered) E.l in Relations
      l : {E2}[14]
    for relation E.l {*,+} (ordered) <-> m2 E2.r in Relations
      l : [E2] (inverse=r)[15]
    for relation E2.r m2 <-> {*,+} (ordered) E2.r in Relations
      l : [E2][16]

    for relation E.l {1,?} <-> m2 E2.r
    and relation E2.r m2 <-> {1,?} E.l in Relations
      static function get_l(e : E) : E2 { return if(e == null) null else e.l;}[17]

      static function get_l(entities : [E]) : [E2]{
        return [E.get_l(e) | e : E in entities where E.get_l(e) != null];
      }[18]

    for relation E.l {+,*} <-> m2 E2.r
    and relation E2.r m2 <-> {+,*} E.l in Relations
      static function get_l(e : E) : [E2]{
        return if(e == null) null else [e2 | e2 : E2 in e.l];
      }[19]

      static function get_l(entities : [E]) : [E2]{
        return [e2 | e : E in entities, e2 : E2 in e.l];
      }[20]
  }
```

Figure 3.11 Compilation scheme for Calculate-on-Read implementation strategy

values [29,30,36], and code for updating dirty values [27,28].

Dirty flagging of derived values happens when underlying values are updated. WebDSL provides `extend function` hooks to intercept calls to setters. When a setter is called, all dependent values are dirty flagged by traversing the data flow `paths` [31–34]. Attributes and relations with multiplicity `?` and `1` only dirty flag when the value changes [31], while relations with multiplicity `*` and `+` dirty flag on additions and removals [32,33]. As relations have two names, dirty flagging is done for both names. Moreover, updating a relation can also implicitly remove another relation. For example, moving a submission to a different assignment

```
bobsSubmissionToMath.assignment := logicAssignment;
```

will trigger:

```
bobsSubmissionToMath.set_assignment(logicAssignment);
mathAssignment.remove_from_submissions(bobsSubmissionToMath);
logicAssignment.add_to_submissions(bobsSubmissionToMath);
```

Recalculation of derived values [35] is performed after user code is run, and before the flush to database. The computation is scheduled statically by means of the topological sort of the connected components in the data flow graph. Within a connected component, a `while` continues computing derived values until none of the derived values is dirty anymore.

Calculate on Write Properties

This compilation scheme yields programs with the following properties: (1) the derived value reads are consistent, (2) transactions might fail, (3) cyclic derived values can cause non-termination, (4) scheduling is optimal for acyclic dependency graphs, and (5) scheduling is naive for connected components inside the dependency graph.

Consistency of derived values is based on consistency of derived values within a single HTTP request, and database concurrent transaction semantics. For any changed attribute or relation, all the values that depend on it are dirty flagged and recomputed. By induction, all values that depend transitively on a changed value get dirty flagged and recomputed. Computation only stops if all dirty flags are processed. As such, for a specific HTTP request, all derived values in memory are up to date when computation terminates. Flushing to the database only succeeds if previously read data remains unchanged, guaranteeing consistency. Failing transactions occur more often in Calculate-on-Write than in Calculate-on-Read, as both the updates to base values, and the updates to derived value caches can cause conflicts.

Cyclic derived values, such as the average submission grade depending on itself, can cause non-termination. If an updated value dirty flags itself (transitively), and its new value is different, the computation loops. A diverging value causes non-termination, while a converging value is a fix point calculation. Incremental Datalog implementations guarantee termination by disallowing recursive aggregation and negation: stratified negation [Apt et al., 1986] and stratified aggregation [Mumick et al., 1990]. We allow recursive aggregation, but do not guarantee termination.

```
//All code from Calculate-on-Read, except generated fields for attributes.
for E in Entities
  entity E {
    for a : T m in E.attributes
      a : T (default=null)[21]

    for a : T m = e1 in E.attributes
      a : T (default=calculate_a())[22]

      function update_a() {  a := calculate_a();  }[23]

    for a : T m = e1 (default) in E.attributes
      a : T (default=null)[24]
      a_default : T (default=calculate_a())[25]

      function update_a() {  a_default := calculate_a();  }[26]

    for a : T m {= e1, = e1 (default)} in E.attributes
      static function a_update_all() {
        for(e in E.get_and_empty_a_dirty()){  e.update_a();  }
      }[27]

      static function get_and_empty_a_dirty() : {E} {
        var values := E_a_dirty; E_a_dirty := Set<E>(); return values;
      }[28]

      static function a_has_dirty() : Bool {  return E_a_dirty.length != 0;  }[29]

      static function a_flag_dirty(entities:{E}){ E_a_dirty.addAll(entities);}[30]

    for E.a->path.a2 in DataFlow where a.multiplicity in {?,1} and E2=a2.entity
      extend function set_a(newV : T){ if(a!=newV){ E2.a2_flag_dirty(path); }}[31]

    for E.l->path.a2 in DataFlow where l.multiplicity in {*,+} and E2=a2.entity
      extend function add_to_l     (n:T){ if(l!=n){ E2.a2_flag_dirty(path); }}[32]

      extend function remove_from_l(n:T){ if(l!=n){ E2.a2_flag_dirty(path); }}[33]

    for E.a->path.a2 in DataFlow where a2 : T m = e1 (default) and E2=a2.entity
      extend function set_a_default(newValue : T) {
        if(a == null && a_default != newValue){  E2.a2_flag_dirty(path);  }
      }[34]
  }

// update_derivations gets called before flush to database
function update_derivations(){
  var not_empty : Bool;
  for ConnectedComponent cc in DataFlowGraph topologically sorted
    not_empty := true;
    while(not_empty){
      for a : T m {= e1, = e1 (default)} in cc where E = a.entity
        E.a_update_all();
      not_empty := false;
      for a : T m {= e1, = e1 (default)} in cc where E = a.entity
        not_empty := not_empty || E.a_has_dirty();
    }
}[35]

for a : T m {= e1, = e1 (default)} in Attributes and E = a.entity
  request var E_a_dirty : {E} := Set<E>()[36]
```

Figure 3.12 Compilation scheme for Calculate-on-Write implementation strategy

Derived values should only be recomputed after all values they depend upon are already updated. With acyclic data flow graphs, topological scheduling completely removes unnecessary recomputation. With cyclic data flow graphs, topological scheduling only partially removes unnecessary recomputation: the connected components are statically scheduled, but the derived values inside a connected component are updated without scheduling.

3.4.4 *Calculate Eventually*

Figure 3.13 defines the Calculate-Eventually compilation scheme. It builds on the Calculate-on-Write compilation scheme, only stating additions and changes. The idea is to take the dirty flags from Calculate-on-Write, but pass these on to a separate, dedicated thread, allowing the HTTP request handlers to finish early. The writes to base values will still be synchronous, but the updates to derived values will be asynchronous. So, to translate to WebDSL we need to generate code that (1) dirty flags cross-thread, and (2) updates derived values in a separate thread.

Cross-thread dirty flagging communicates dirty flags from request handlers to the updater thread. WebDSL abstracts over concurrent handling of requests by running request handlers completely separated from each other. Communication between the threads handling HTTP requests, normally, is through the database. However, the database cannot notify the updater thread, so in memory communication is required. To communicate in memory between threads in WebDSL we need `native` Java code. For each derived value attribute we generate a `ConcurrentLinkedQueue` [45], and make this queue available in WebDSL by means of a static function in a `native class` [44]. As an HTTP request is handled, derived values get dirty flagged locally (as in Calculate-on-Write). After the changes are flushed to database, the local dirty flags are communicated cross-thread. Because an entity can be mapped from the relational database to an object in memory multiple times (once per request handler), the cross-thread dirty flagging needs to communicate an entity's unique identifier (UUID) [39].

The *derived value recalculation thread* is started with WebDSL's recurring tasks mechanism [42]. Every millisecond the thread is started, if not still running. The thread performs the same calculations as Calculate-on-Write, but uses the cross-thread dirty flags [43]. It loads entities with dirty flagged derived values into memory [37], then updates derived values, and finally propagates its own local dirty flags to the cross-thread dirty flags [39].

Calculate Eventually Properties

The Calculate-Eventually programs have the following properties: (1) derived values will eventually be up to date, (2) derived value reads are not glitch-free, (3) derived value calculation can starve under load, (4) after load subsides only relevant updates are calculated, and (5) cyclic values can cause non-termination.

```
//All code for Calculate-on-Write, except for update_derivations.
for E in Entities
  entity E {
    for a : T m {= e1, = e1 (default)} in E.attributes
      static function get_and_empty_a_dirty_async() : {E} {
        var queue := DirtyQueues.get_E_a_queue();  var vals : {E};
        while(!queue.isEmpty()){
          vals.add(loadEntity(E, UUIDFromString(queue.poll() as String)) as E);
        }
        return vals;
      }[37]

      static function a_has_dirty_async() : Bool {
        return !DirtyCollections.get_E_a_queue().isEmpty();
      }[38]

      static function a_flag_dirty_async() {
        var dirty := E.get_and_empty_a_dirty();
        DirtyCollections.get_E_a_queue().addAll([v.id.toString() | v in dirty]);
      }[39]

      static function a_update_all_async() {
        for(e in E.get_and_empty_a_dirty_async()){  e.update_a();   }
      }[40]
  }

//flag_dirty_async is called on every request after write to database
function flag_dirty_async() {
  for a : T m {= e1, = e1 (default)} in Attributes and E = a.entity
    E.a_flag_dirty_async();
}[41]

invoke update_derivations() every 1 milliseconds[42]
function update_derivations(){
  var not_empty : Bool;
  for ConnectedComponent cc in DataFlowGraph topologically sorted
    not_empty := true;
    while(not_empty){
      for a : T m {= e1, = e1 (default)} in cc where E = a.entity
        E.a_update_all();
      flagDirtyAsync();
      not_empty := false;
      for a : T m {= e1, = e1 (default)} in cc where E = a.entity
        not_empty := not_empty || E.a_has_dirty();
    }
}[43]

native class derivations.DirtyQueues as DirtyQueues {
  for a : T m {= e1, = e1 (default)} in Attributes and E = a.entity
    static get_E_a_queue() : Queue[44]
}
```

```
// Not expressible in WebDSL: Java code
public class DirtyQueues {
  for a : T m {= e1, = e1 (default)} in Attributes and E = a.entity
    private static Queue<String> E_a_queue =
      new ConcurrentLinkedQueue<String>();[45]
    public static Queue<String> get_E_a_queue(){  return E_a_queue;  }[46]
}
```

Figure 3.13 Compilation scheme for Calculate-Eventually implementation strategy

Eventual calculation is guaranteed by the invariant that outdated derived values are always accompanied by a dirty flag. Dirty flags are only sent by request handling threads after their changes are flushed to the database, ensuring the updater thread never processes dirty flags without seeing the changes. During updates (the same) derived values might be dirty flagged again. To ensure new dirty flags are processed, the updater thread copies and empties the dirty flag queues before processing flags. New flags will be processed subsequently.

Glitch-freedom is not provided inside connected components, as there is no topological ordering on the instance level. Also, derived value calculation starvation happens when server load is high. However, successive changes to the same base values will not create extra dirty flags. So when the system has spare resources, it will just compute the derived values based on the latest base values, and ignore all the intermediate base values. And finally, like Calculate-on-Write, cyclic derived values can cause non-termination.

3.5 EVALUATION

The declarative specification of derived values in IceDust allows switching implementation strategies to realize different non-functional requirements without invasive code changes. In this section we benchmark different generated implementations, to evaluate whether they are indeed able to satisfy different non-functional requirements. The benchmarks differ in (1) the read/write ratio, (2) the number of base values derived values depend upon, and (3) the number of fully unrelated derived values. The measured non-functional properties are (1) throughput of derived value reads and base value writes per second, (2) the number of failing writes per second, and (3) the response time for reading derived values and writing base values. In this section we will discuss the benchmark setup and results.

3.5.1 *Benchmark Setup*

In the case study (Section 3.6) we encounter derived values that depend on up to 60000 values transitively. The essence of the calculation is a tree-like structure with aggregations on every level. For the benchmark evaluation of the different implementation strategies we use a simplified model, a simple tree with an average on each level:

```
entity Node{ avgValue : Float? = avg(children.avgValue) (default) }
relation Node.parent ? <-> * Node.children
```

The tree branches out with a factor of 10, up to 6 deep (size 1, 11, ..., 111111).

The benchmarks consist of read and write requests. Read requests retrieve the average at the top node of the tree. Write requests update a value at a random leaf node of the tree. Benchmarks are warmed up for 10 seconds, and then measured for 15 minutes. The Siege[2] tool is used to execute the

[2] https://www.joedog.org/siege-home/

benchmark requests. It is configured to use 10 concurrent threads, which initial benchmarks indicated to be a reasonable concurrency level. If the concurrency level is too low, the computer is not using maximum resources; if it is too high, too many requests will queue up increasing response times but not improving throughput. The benchmarks were performed on an early 2013 Macbook Pro laptop with Intel Core i7 2,7Ghz, 4 cores (8 threads), and 16 GB memory. The Java servlet web application generated by WebDSL was deployed on OS X 10.11, Java 1.8.0_60, MySQL 5.6.27, and Tomcat 7.0.40.

3.5.2 *Benchmark Results*

The first two benchmarks determine the behavior in extreme workloads with only read or only write requests. Figure 3.14 shows that the performance for a workload of 100% decreases as the tree gets deeper. Beyond depth 4 the response takes longer than 0.1 second, and is noticeable for users. Calculate-on-Read response times increase linearly with the number of read objects, indicating that this is the limiting factor. The other implementation strategies stay at a steady high throughput with low latency, because they only retrieve a single node with a cached value on each request. The maximum throughput is around 1900 transactions per second, indicating the general overhead of the system.

The benchmark in Figure 3.15 shows the performance for write-only workloads. In addition to the throughput of successful requests, the failed requests are indicated by the dashed lines. When a database transaction fails due to conflicting writes, WebDSL retries it up to 3 times before failing the entire request. This improves usability for typical scenarios where a single transaction conflict may occur occasionally. Multiple subsequent failed transactions only occur in extreme situations where many concurrent requests conflict. For example, in this benchmark at tree depths 1 (1 object) and 2 (11 objects), all implementation strategies have repeated transaction failures resulting in failed requests. In general, the maximum throughput the system supports for concurrent edits on a single object is close to 300 edits per second (tree depth 1, all implementation strategies). Calculate-on-Write has many request failures (around 60%) at all tree depths, which makes the implementation unusable in practice for this use case. Calculate-on-Read and Calculate-Eventually have overall high throughput and low latency, except for the low tree depths where transaction failures occur.

Figure 3.16 shows the trade-off between Calculate-on-Read and Calculate-on-Write in mixed read/write ratio workloads. Calculate-on-Write suffers from many transaction failures, except for 100% reads. So, it is unusable if all base values aggregate into a single value, even with a small number of concurrent writes. Calculate-on-Read improves when the workload shifts from reads to writes. However, the average response time for anything except 100% writes is unacceptebly high. Calculate-Eventually has stable high throughput and low latency for all the different workload scenarios. If all base values in a system aggregate transitively into a single derived value, the

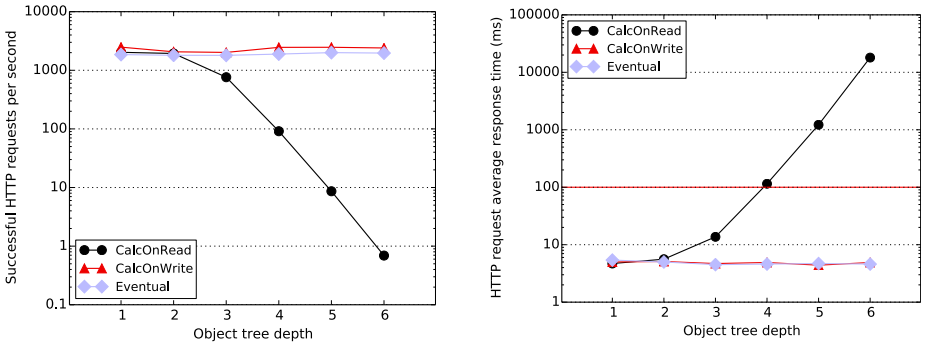


Figure 3.14 Read-only workload benchmark throughput (left) and latency (right)

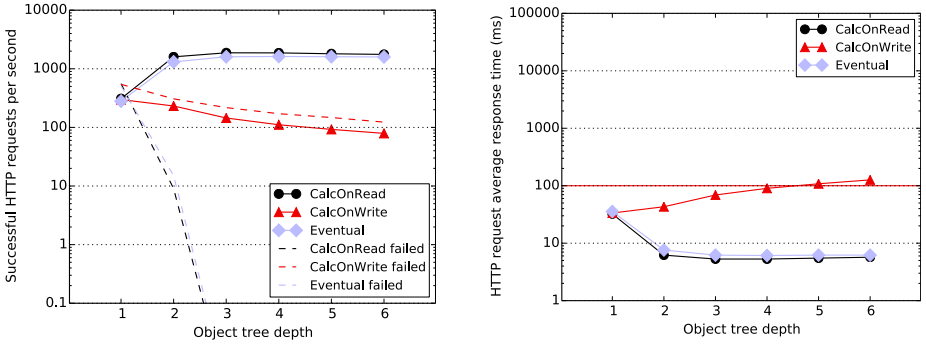


Figure 3.15 Write-only workload benchmark throughput (left) and latency (right)

only viable strategy is Calculate-Eventually.

In the final benchmark, shown in Figure 3.17, we investigate whether not aggregating all base values into the same derived value makes the Calculate-on-Write strategy viable. We compare the implementation strategies when there are up to 256 separate trees of depth 4. Calculate-on-Write performs better indeed in scenarios with more disconnected derived values. With 16 trees, the number of failed transactions drops below 0.5%. If consistency is desired, and derived values depend on roughly 1000 base values, the trade-off throughput-wise between Calculate-on-Read and Calculate-on-Write is at 2 separate trees. However, Calculate-on-Write still has many failing requests. Only at around 16 trees the number of failing transactions falls below 0.5%, and Calculate-on-Write becomes a viable option. When eventual calculation is acceptable, it is always the most performant solution.

3.5.3 *Discussion*

We could perform many more benchmarks (for example with other data structures than trees, or with workloads with more structure than a read/write ratio). However, the presented benchmarks show that each implementation strategy is useful in specific cases.

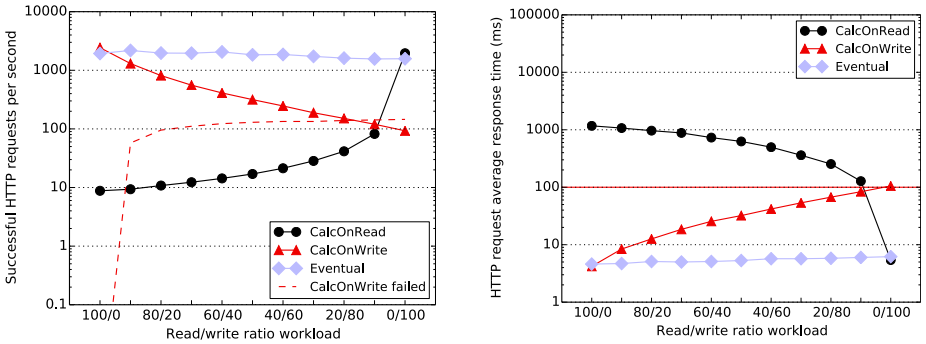


Figure 3.16 Varying workload benchmark throughput (left) and latency (right) with tree depth 5.

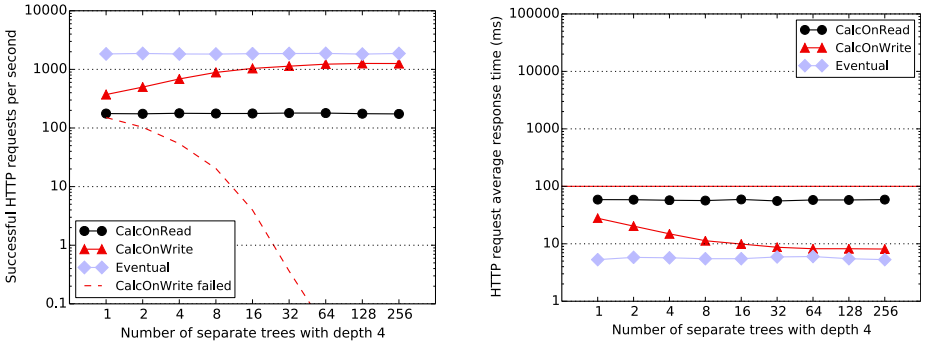


Figure 3.17 Separate trees benchmark throughput (left) and latency (right) with 50-50 workload.

The form of non-functional requirements determines the form of verification required [Glinz, 2005]. The verification for quantitative requirements is measurements, and for operational requirements is review, test or formal verification [Glinz, 2005]. Consistency and eventual calculation are operational requirements. We tested whether our implementations satisfy consistency and eventual calculation. In future work this can be improved with formal verification.

3.6 CASE STUDY

We applied IceDust to the grading policies in a learning management system in which students can submit assignments that get graded semi-automatically. The system contains complex derived value calculations, contains a lot of data (hundreds of thousands of entities, with millions of derived values), and is subject to intense workloads on a small subset of the data. The complex derived value calculations were specified in IceDust's declarative derived value attributes, and the Calculate-Eventually strategy was used to generate an implementation. In this section we (1) reflect on the expressiveness of IceDust, based on the experiences from the case study, and (2) highlight parts of the

resulting declarative specification that are quite different from the original imperative implementation.

First, let us introduce the learning management system in more detail. The system is a much more complicated version of the one introduced in Section 3.3. It features semi-automatic grading, programming assignments with test cases, and automatically graded multiple choice questions. Assignments are structured in a tree, and students get a weighted average for each node in the tree up to the top, which is their final grade for the course. The grading logic also includes deadlines, deadline extensions, late penalties, minimum grades, and alternative assignments. The courses, and their assignments, have statistics such as the percentage of students with a passing grade. For the largest course in the system, the statistics depend transitively on ± 60000 individual submissions (± 250 students, with ± 15 assignments per week, running for 12 weeks, and exams with multiple questions in the end).

Explicit Not Yet Calculated Values Expressing the grading logic in IceDust forced us to look at previously implicit things. Students which did not attempt an assignment get a 0.0 on a scale of 1.0 to 10.0. The other students would get a 1.0, as grading would be triggered:

```
// only call calculateGrade if submission is attempted
function calculateGrade() {  grade := max(1.0, calculatedGrade);  }
```

The grading logic states that grades cannot be lower than 1.0, but if grading is not triggered the float default value is used. As the compiled code from IceDust detects everything that should be calculated, these grades would be changed from 0.0 to 1.0. To model assignments that are not attempted by students, `attempted` should be explicitly mentioned:

```
grade : Float = if(not attempted) 0.0 else max(1.0, calculatedGrade)
```

Explicit Stateful Calculations The imperative code, also implicitly, kept old grades when grades where published and a newly calculated grade was lower:

```
if(assignment.statsPublic()){ newgrade := max(oldgrade, newgrade); }
```

In the new specification this requires an explicit self-reference:

```
grade : Float = if(public) max(grade, calculatedGrade)
                else       calculatedGrade
```

Note that the IceDust specification only works for push-based implementations: Calculate-on-Write or Calculate-Eventually. Calculate-on-Read would throw a stack overflow exception. The sentence *'only update when grade is higher'* implies push-based calculation: the previous calculated grade needs to be cached for when a new grade becomes available. It is arguable whether someone should express logic like this, as once grades are visible, it is not traceable anymore how a grade was calculated. This is on the border of what is expressible in IceDust.

Code Factorization Differences In IceDust the value of an attribute is defined in a single place: the derived value expression. In imperative code, assignments

to attributes can happen in multiple places, which means assignments can be distributed over `if`s:

```
if(assignment.passOne){
  passSub := disj([s.pass() | s:Submission in submissions]);
  grade   := max([s.grade() | s:Submission in submissions]);
} else{
  passSub := conj([s.pass() | s:Submission in submissions]);
  grade   := avg([s.grade() | s:Submission in submissions]);
}
```

In IceDust `if`s need to be distributed over derived value attributes:

```
grade   : Float?  = if(assignment.passOne) max(children.grade)
                    else                   avg(children.grade)

passSub : Boolean = if(assignment.passOne) disj(children.pass) //one
                    else                   conj(children.pass) //all
```

Whether the old or new specification is preferable is arguable. If the cases would be more complex than a single if it would lead to repeated code in IceDust.

Application Analysis Finally, we made a more analytic observation: even though the data-flow graph of the specification in IceDust contains more than 100 nodes, it contains just a single connected component. Grades, weightedGrades (weighted averaging is used), pass, and child-pass are mutually recursive (like Figure 3.9). All other dependencies are acyclic. This system has derived value-wise just a single complex part: the grade calculation. Intuitively we already knew this, but now we can quantify this with properties of the data-flow graph.

3.7 RELATED WORK

The related work is organized along language design and the three implementation techniques (Calculate-on-Read, Calculate-on-Write, and Calculate-Eventually).

3.7.1 *Languages with Relations*

There are multiple languages that feature relations as a language construct. We will cover closely related languages and highlight the differences.

Rumer [Balzer, 2011] features first-class citizen relations with queries for navigation. IceDust relations are not first-class citizen, and navigation is through member access instead of queries. In Rumer multiplicities can be specified in constraints which are enforced at runtime, while in IceDust these are part of the type system. Rumer does not support derived value attributes, but queries can be used to specify Calculate-on-Read derived values. Finally, Rumer is an imperative in-memory language, while IceDust is declarative and persists its data.

RelJ [Bierman and Wren, 2005] also features first-class citizen relations. In RelJ participants of relations do not have names, so navigation is positional

(using `from` and `to`). RelJ features only multiplicity upper bounds, no lower bounds. These multiplicities are enforced at runtime: either by throwing exceptions, or by implicitly removing previous relations. RelJ does not feature derived value attributes, and RelJ is an imperative in-memory language like Rumer.

Relations [Harkes and Visser, 2014] features multiplicities as part of the type system and derived value attributes like IceDust. Its derived values are, however, only Calculate-on-Read. Relations is declarative, like IceDust, but its data is only in memory, not persistent. Relations in this language are first-class citizen like Rumer and RelJ, but feature navigation through member access. IceDust relations are not first-class citizen, but feature the same member access navigation.

Alloy [Jackson, 2002] is a language for bounded model checking which features language constructs similar to IceDust: bidirectional relations, multiplicities, and derived values. Alloy is more expressive than IceDust: it features n-ary relations, and its derived values specify derived relations (as opposed to derived attribute values). However, Alloy's bounded model checker only works on small data sets, and primitive values (only integers in Alloy) should be avoided as they blow up the state space. IceDust, on the other hand, supports derived values over arbitrary primitive values (int, string, float, datetime, and boolean), and admits efficient implementation strategies applied to large data sets. To compute derived values in large data sets from an Alloy specification, Alloy would need an operational semantics not based on bounded model checking or SAT solving. An approach for an operational semantics for Alloy was proposed in [Giannakopoulos et al., 2009], but this approach is not complete. As Alloy has much greater expressive power (first-order logic), we also expect such an Alloy operational semantics to not be efficient. Finally, another difference is that in IceDust the multiplicities are checked in the type system, while in Alloy these are only checked during bounded model checking.

3.7.2 *Calculate on Read*

We do not cover Calculate-on-Read extensively, as it is the default implementation for many formalisms. We cover only the object-oriented approaches.

Object-oriented languages lend themselves for various Calculate-on-Read optimization techniques. Wiedermann and Cook take imperative code with for loops and if statements and convert those to SQL queries [Wiedermann and Cook, 2007]. Their approach is similar to our work in that it operates on persistent objects by means of an object-relational mapper. Also their approach for analyzing dependencies is similar: path-based abstract interpretation. They optimize imperative code that can be expressed as queries. Our approach, on the other hand, treats code that cannot be expressed as queries, recursive aggregation. The Java Query Language (JQL) adds queries to Java [Willis et al., 2006]. The rationale for queries is that these are more succinct to write, and more efficient than nested for loops. JQL has been incrementalized,

we will cover this in the next subsection. This chapter adds over these approaches the possibility to easily switch to an incremental or eventual calculation implementation strategy.

3.7.3 *Calculate on Write (Incremental Computation)*

Incremental computation is present in many fields in computer science. We relate our Calculate-on-Write implementation scheme to existing incremental approaches.

Materialized views in relational databases can be incrementally maintained [Gupta and Mumick, 1995]. Recursion and stratified aggregation can be supported [Gupta et al., 1993]. Stratified aggregation does not admit recursive aggregation. (See next paragraph for relaxations of stratified aggregation in logic databases.) Switching between implementation strategies in relational database also do not require invasive code changes: the definitions for materialized and non-materialized views are identical. Relational databases do, however, not support eventually-calculated views.

Logic Databases or Deductive Databases are a more expressive than relational databases. Logic Databases support stratified aggregation like relational databases [Mumick et al., 1990]. Since stratified aggregation does not support recursive aggregation, more relaxed notions of aggregation have been introduced, such as Monotone Aggregation [Ross and Sagiv, 1992]. Monotone Aggregation has also been incrementalized [Ramakrishnan et al., 1994]. A recent survey [Green et al., 2013] states that at present, the Datalog community seems not to have converged on any of the proposed semantics for aggregation through recursion. This means that in practice recursive aggregation is often not supported. For example LogiQL [Green, 2015], the language of LogicBlox, does not support recursive aggregation.

Functional reactive programming (FRP) [Elliott, 2009], for example REScala [Salvaneschi et al., 2014], Scala.React [Maier and Odersky, 2013], or i3QL [Mitschke et al., 2014], provides incremental computation. Calculate-on-Read style code wrapped with FRP libraries behaves as Calculate-on-Write. FRP abstractions provide single-threaded, in-memory derived values. In contrast, IceDust provides concurrent, persistent derived values.

Spreadsheets provide incremental computation. The data structure in a spreadsheet is a 2d grid. IceDust's data structure is an object graph. Moreover our object graph is typed, while spreadsheets are free form. Spreadsheets do mostly have an implicit structure [Hermans et al., 2010]. IceDust with Calculate-on-Write can be seen as a structured spreadsheet without a 2d grid.

Object-oriented programs can also be incrementalized. Incremental Updates for Materialized OQL views [Gluche et al., 1997] proposes to generalize incremental view maintenance from relational databases to support the Object Query Language (OQL) as view definition language. MOVIE [Ali et al., 2003] develops this work further. They also provide an overview of relational incremental view maintenance implementations, with either a relational or an object-oriented surface syntax. These approaches, even though some have an

object-oriented surface syntax, are part of the relational paradigm (with the limitations previously mentioned for materialized views).

The Java Query Language is incrementalized [Willis et al., 2008]. Their benchmarks show, like ours, that for different read-write ratio workloads the incremental or calculate-on-read solution offers better performance. Demand-Driven Incremental Object Queries [Liu et al., 2015] improves over JQL by using auxiliary indices for incrementality. Similar to [Wiedermann and Cook, 2007] they transform imperative code to a relational calculus. But instead of performing relational queries like [Wiedermann and Cook, 2007] they use the relational model to generate code that incrementally maintains the caches. Our approach uses path-based abstract path interpretation instead of a relational calculus to generate maintenance code. Both of the above approaches slightly differ in use cases from our approach: they target set membership of objects e.g. whether an object belongs to a set specified by a query, while our approach targets derived value attributes.

Graph queries can be incrementally evaluated in IncQuery [Szárnyas et al., 2014]. IncQuery's data structure is a graph, like ours, but its goal is to pattern match. Our approach does not support pattern matching on graph structures, rather it computes derived attribute values.

Attribute grammars feature a declarative style of specifying derived values. Attribute grammars can also be incrementally computed [Demers et al., 1981]. As attribute grammars only support trees, one could look at reference attribute grammars to support full blown graphs [Söderberg and Hedin, 2012]. Reference attribute grammars do support graph structures, but there is a clear distinction between the tree, and the derived graph edges. In our approach the graph is the basis. Fitting our data models onto attribute grammars would require extracting a spanning tree, and deriving the other edges. In this process we would lose the correspondence to the data flow graph, and derived edges would become dynamic dependencies, which would complicate scheduling.

Self-adjusting computation [Acar, 2009] does not cover a single programming paradigm as it features multiple languages (including SLf, for functional programming, and SLi, for imperative programming). Self-adjusting computation automatically transforms a Calculate-on-Read style program to a Calculate-on-Write style program. Our approach does not take Calculate-on-Read as basis, but instead provides a declarative language to express derived values.

3.7.4 *Calculate Eventually*

The code generated by the Calculate-Eventually implementation makes derived values of attributes eventually consistent with base values. We cover existing work on eventual (or asynchronous) computation of derived values in this subsection.

Event and Actor programming, with for example Akka [Gupta, 2012] or RX [Meijer, 2010], provide an asynchronous update mechanism for calculating derived values. Updates to derived values are asynchronous, meaning that

there is no consistent view of base values and derived values at the same time. As such, these do not provide consistency, like the code produced by our Calculate-Eventually implementation strategy.

Eventual consistency for distributed data also features eventual calculation, but is unrelated. As a recent survey on Eventual Consistency states: "shared data is updated at different replicas, updates are transmitted asynchronously, and conflicts are resolved consistently" [Burckhardt, 2014]. Our approach does not have different replicas of data, there is a single database. Our approach does not have asynchronous updates, the update is synchronous as a HTTP response is only sent after the transaction in the database is completed. And finally, our approach does not have conflicts during the calculation of derived values, as the base values define unambiguously what the derived values of attributes should be.

3.8 CONCLUSION

Data modeling with declarative derived value attributes in IceDust allows deferring the decision about implementation strategy from implementation to compilation time, and allows switching strategies without invasive code changes. We have demonstrated that these different strategies provide different non-functional properties, so that a specific strategy can be chosen to realize certain non-functional requirements. Finally, a case study indicated our approach is useful for expressing derived values of systems used in practice.

In future work, we would like to explore more implementation strategies, such as transitive dirty flagging on writes with recalculation on reads, or eventually calculated with flags indicating whether the values are up to date or not. We also would like to explore more flexibility in implementation strategies by allowing composition of different strategies, and live switching between strategies. A type system should restrict compositions to only sound ones: consistent values cannot depend on eventually calculated values, and calculate on write values cannot depend on calculate on read values. Finally, we would like to guarantee termination by specifying non-circular relations and runtime non-circularity checking.

IceDust 2

Derived Bidirectional Relations and Calculation Strategy Composition [1]

4

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 chapter 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.

4.1 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 [Gupta et al., 1993]. More recently it has seen much attention in new fields. Incremental programming [Hammer et al., 2015, 2014; Harkes et al., 2016; Mitschke et al., 2014; Ujhelyi et al., 2015] uses previously calculated values to efficiently compute new ones. In (functional) reactive programming

[1] This chapter has appeared as Harkes, D. C. and Visser, E. (2017). Icedust 2: Derived bidirectional relations and calculation strategy composition. In Müller, P., editor, *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, volume 74 of LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

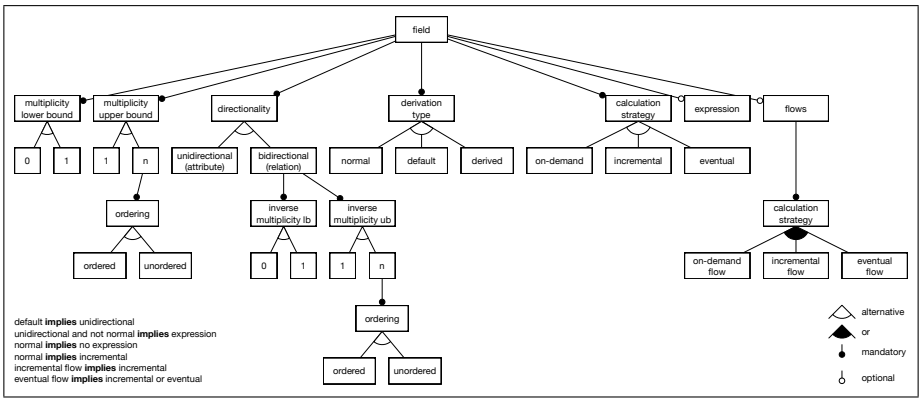


Figure 4.1 Feature model for configuration of a field in IceDust and IceDust2

[Elliott, 2009; Maier and Odersky, 2013; Meijer, 2010; Salvaneschi et al., 2014] 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 [Harkes et al., 2016], or composition is not statically checked to guarantee that derived values will be consistent with the values they depend upon [Meijer, 2010; Salvaneschi et al., 2014]. 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 [Harkes et al., 2016] 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[2] in Figure 4.1. 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

[2] A feature model is a compact representation of all the products of a software product line (SPL)[Kang et al., 1990]. A product configuration is determined by a selection of features satisfying the constraints of the feature model.

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

- 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 chapter 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 4.3 we analyze the run-time interaction between derived values, bidirectional relations, multiplicity bounds, and various calculation strategies. In Section 4.4 we define the operational semantics covering all possible feature combinations. In Section 4.5 we describe the type system guaranteeing sound composition of calculation strategies. In Section 4.6 we discuss two implementations of IceDust2. In Section 3.6 we evaluate the expressiveness of the language with case studies. In Section 4.8 we analyze the limitations entailed by static multiplicity checks on derived relations. In Section 4.9 we compare IceDust2 to other approaches to declarative data modeling.

4.2 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.

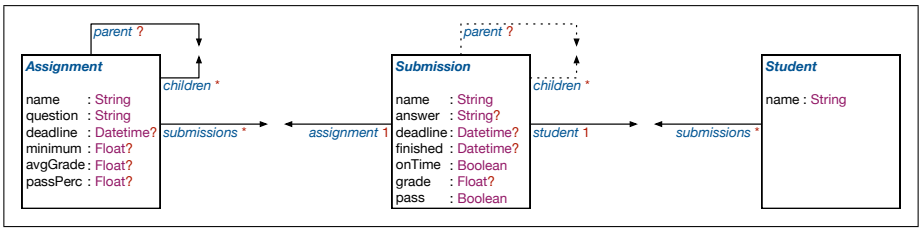


Figure 4.2 Running example class diagram. Bidirectional relations are denoted by $\rightarrow\leftarrow$, and dotted lines express derived relations.

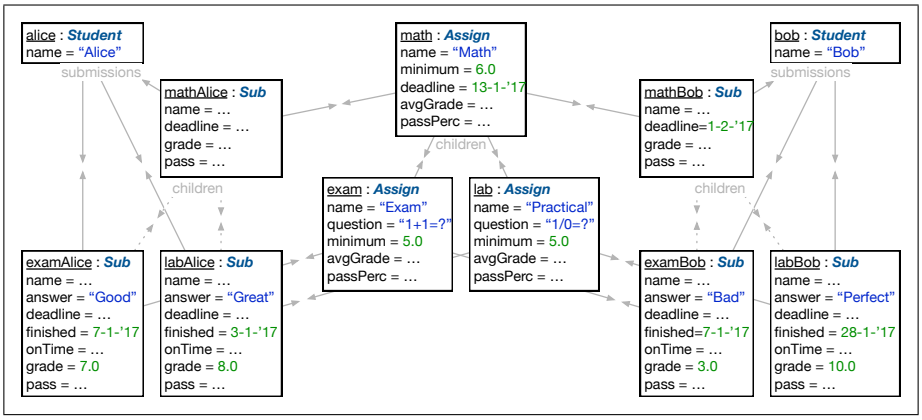


Figure 4.3 Running example data. References are denoted by $\rightarrow$, bidirectional relation values are denoted by $\rightarrow\leftarrow$, derived references are dotted arrows, and derived attribute values are dots.

4.2.1 *Running Example*

To illustrate data modeling in IceDust and IceDust2, we use a simplified learning management system as running example (Figures 4.2–4.4). `Assignments` are structured as a tree. For example, the `math` assignment consists of an `exam` and a `lab` (Figure 4.3 center). `Students` submit `Submissions` to these assignments. These submissions form trees as well, mirroring the assignment tree (see Alice's and Bob's submission trees in Figure 4.3). The tree structure of submissions is derived in order to avoid redundant data, which can lead to inconsistencies.

Assignments have optional `deadlines`. 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 4.3 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 submis-

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

Figure 4.4 Running example IceDust2 specification

sion 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.

4.2.2 *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 4.1.

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 [Steimann, 2013] and Harkes et al. [Harkes and Visser, 2014]. 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:

```
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)
```

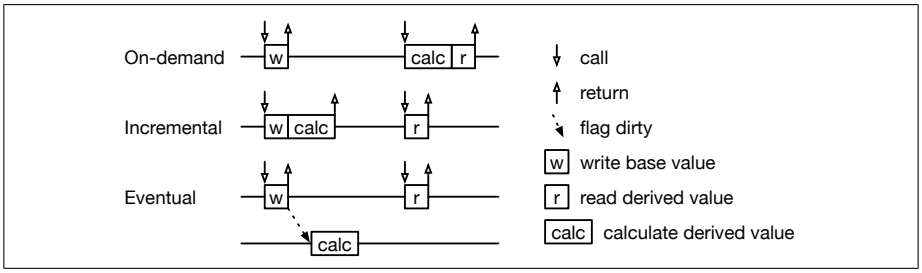


Figure 4.5 Thread activation diagrams for different calculation strategies

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 4.5 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.

4.2.3 *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 chapter 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 4.2 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 [Salvaneschi et al., 2014], 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 4.2 and 4.3 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()
}
```

Sound 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 4.5 for more details).

Constraints on Feature Composition IceDust2 allows almost all combinations of features in Figure 4.1, but we impose three restrictions. First, we disallow unsound composition of calculation strategies as we will discuss in Section 4.5.

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 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
```

4.3 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 4.3, consider executing `lab.setParent(exam)`, moving the `lab` from `math` to `exam`. Bidirectional maintenance will update `math.children` and `exam.children`. This will trigger 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`. This is identical to executing `lab.setParent(exam)`. Figure 4.6 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 [Davies et al., 2006]; 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`

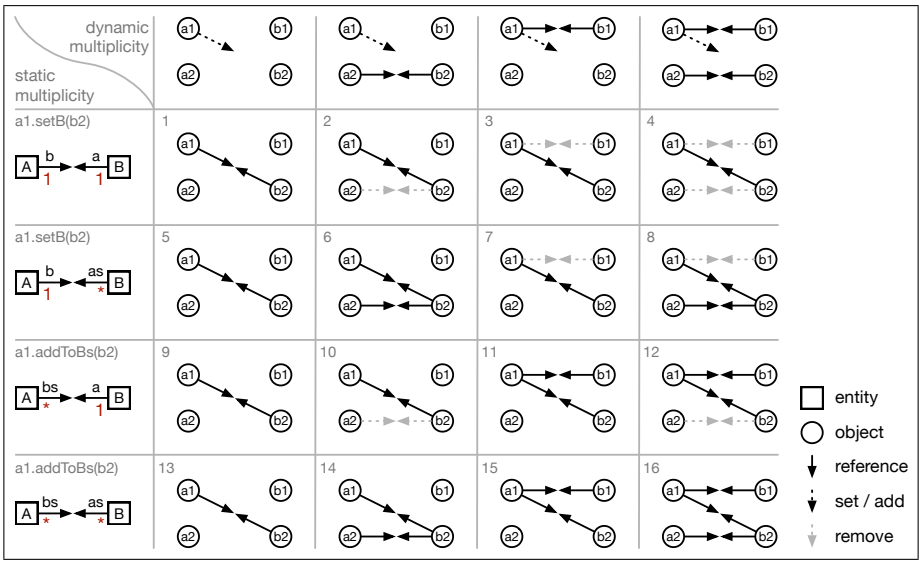


Figure 4.6 Update a bidirectional relation and preserve both bidirectionality and multiplicity upper bounds. Left column shows class diagram with multiplicity bounds, the top row shows starting object graph, and 1-16 show the object graph after update.

and `examBob.assignment` are violated. However, it is undesirable to set `examAlice.assignment` to `lab` implicitly. 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 4.6, 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 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 4.3. 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

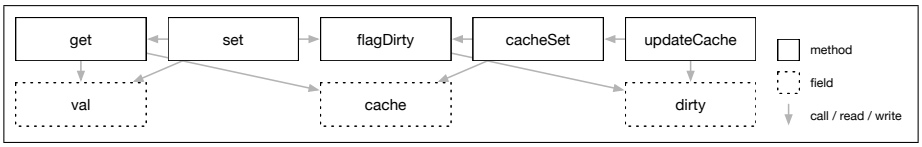


Figure 4.7 General overview for the semantics of a single field in IceDust2

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 need only 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 4.4). 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 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.

4.4 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.

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

Figure 4.8 The store maps combinations of references and field names to tuples of three: user value, cached value, and dirty flag.

Figure 4.7 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 `getter` is responsible for returning the correct value on a read. The `setter` is responsible for maintaining bidirectionality and multiplicity bounds in the `userValue`. Moreover, it calls `flagDirty` on observable changes. The `cacheSetter` does the same for `cacheValue`s. The incremental update algorithm (not shown in Figure 4.7, 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 4.8). 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.

4.4.1 *Getter*

Figure 4.9 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)
$\forall\ E.f \in \mathit{incremental},\ \ \forall\ o \in E,\ \ \Sigma[o, f, \mathit{dirty}] = \mathit{false} \quad \Rightarrow$
$\forall\ E.f \in \mathit{incremental},\ \ \forall\ o \in E,\ \ o.\mathit{calc}(f) \Downarrow \Sigma[o, f, \mathit{cache}]$

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.

4.4.2 *Setter*

Figure 4.10 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 4.6. 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.

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)
$\forall\ E.f \in \mathit{bidir},\ \forall\ o_1 : E,\ o_2 \in o_1.f_1\ \Rightarrow\ o_1 \in o_2.(f.\mathit{inverse})$

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 4.6 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.

4.4.3 *Flag Dirty*

Whenever a value is observably changed, all `incremental` derived values that depend on it are flagged dirty. Figure 4.11 defines the evaluation rules for dirty flagging. Method `dirtyFlows` traverses the data-flow expressions,

Getter evaluation $\boxed{\mathit{Statement}/\Sigma \Downarrow [\text{Value}]/\Sigma}$

$$\frac{f \in \text{normal}}{o.\text{get}(f)/\Sigma \Downarrow \Sigma[o,f].\text{val}/\Sigma} \quad [\text{Get1}]$$

$$\frac{V_3 = [v | v \in V_2,\ o.\text{get}(f) \Downarrow V_2,\ o \in V]}{V.\text{get*}(f) \Downarrow V_3} \quad [\text{Get}*]$$

$$\frac{f \in \text{default} \quad \Sigma[o,f].\text{val} = V \neq [\,]}{o.\text{get}(f)/\Sigma \Downarrow V/\Sigma} \quad [\text{Get2}]$$

$$\frac{f \in \text{on-demand} \quad o.\text{calc}(f) \Downarrow V}{o.\text{getCalc}(f) \Downarrow V} \quad [\text{GetCalc1}]$$

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

$$\frac{f \in \text{incremental} \quad V_2 = \Sigma[o,f].\text{cache}}{o.\text{getCalc}(f)/\Sigma \Downarrow V_2/\Sigma} \quad [\text{GetCalc2}]$$

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

$$\frac{o \vdash (f.\text{expr}) \Downarrow V}{o.\text{calc}(f) \Downarrow V} \quad [\text{Calc}]$$

Figure 4.9 Getter evaluation rules

Setter evaluation $\boxed{\mathit{Statement}/\Sigma \Downarrow /\Sigma}$

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

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

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

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

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

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

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

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

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

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

Figure 4.10 Setter evaluation rules

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 Chapter 3.)

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.

4.4.4 *Update Cache*

After changes, the caches have to maintained, so that reads return up-to-date values. Figure 4.12 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.

4.4.5 *Incremental Update Algorithm*

The update algorithm is responsible for cleaning all caches. The evaluation rules for the update algorithm are defined in Figure 4.13. The data-flow analysis provides a topological ordering which can be used for scheduling updates [Harkes et al., 2016]. Method `maintCache*` invokes `maintGroup*` for each connected component in topological order. Method `maintGroup*` invokes itself recursively while 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 are 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`.

4.4.6 *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 ob-

Dirty flagging $\boxed{\mathit{Statement}/\Sigma \Downarrow [\text{Value}]/\Sigma}$

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

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

Figure 4.11 Flag dirty evaluation rules

Update caches $\boxed{\mathit{Statement}/\Sigma \Downarrow [\text{Value}]/\Sigma}$

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

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

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

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

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

Caches dirty $\boxed{\mathit{Statement}/\Sigma \Downarrow \mathit{Boolean}/\Sigma}$

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

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

Figure 4.12 Update evaluation rules

Update algorithm $\boxed{\mathit{Statement}/\Sigma \Downarrow [\text{Value}]/\Sigma}$

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

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

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

Figure 4.13 Update algorithm evaluation rules

jects. Creation and deletion behavior do not vary based on different field features.

4.4.7 *Multiplicity Lower Bounds*

So far we have ignored multiplicity lower bounds:

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

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.

4.4.8 *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)
$$\begin{array}{lll} \forall\ E.f \in \mathit{incremental},\ \ \forall\ o \in E,\ \ \Sigma[o, f, \mathit{dirty}] = \mathit{false} & \wedge \\ \forall\ E.f \in \mathit{eventual},\ \ \forall\ o \in E,\ \ \Sigma[o, f, \mathit{dirtyEventual}] = \mathit{false} & \Rightarrow \\ \forall\ E.f \in \mathit{eventual},\ \ \forall\ o \in E,\ \ o \vdash f.\mathit{expr} \Downarrow \Sigma[o, f, \mathit{cache}] & \end{array}$$

4.4.9 *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 [Maier and Odersky, 2013; Salvaneschi et al., 2014]. In incremental computing with Adapton, a stronger property is guaranteed: incremental computation returns identical results to from-scratch computation [Hammer et al., 2015, 2014]. 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.

4.5 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 :DSignal[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.

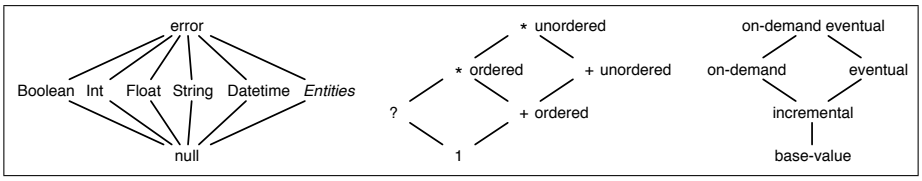


Figure 4.14 IceDust2's type lattice (left), multiplicity and ordering lattice (middle), and composition of calculation strategies lattice (right).

4.5.1 *Type Checking Strategy Composition*

IceDust2 features three calculation strategies: `on-demand`, `incremental`, and `eventual` (Figure 4.5). The first strategy is pull-based, while the latter two 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 4.14, 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, 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 4.15. 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].

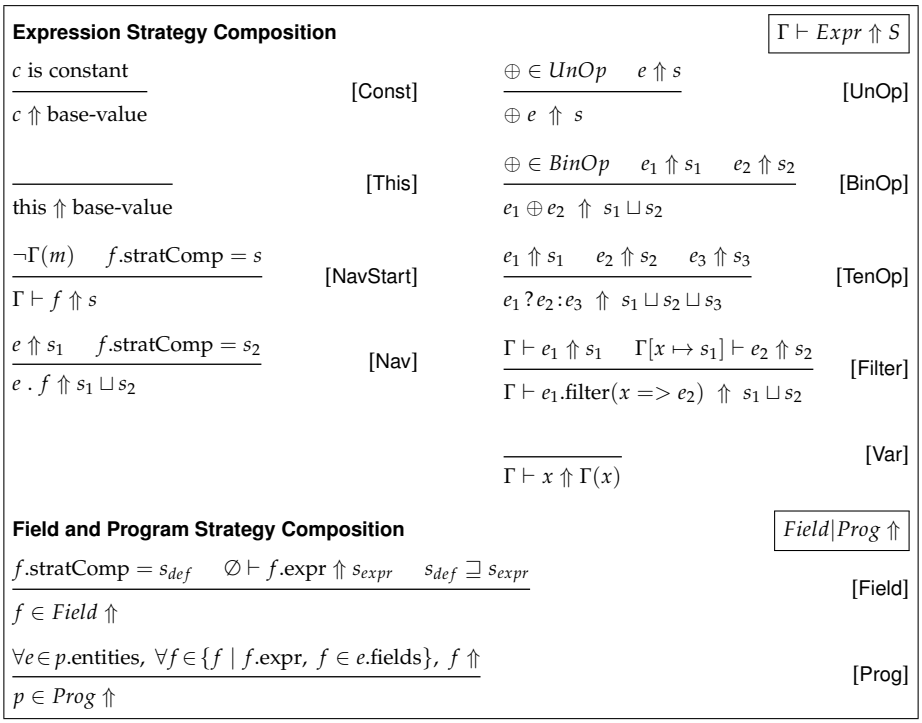


Figure 4.15 Strategy composition rules

4.5.2 *Example*

Lets apply these rules to an example. 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 (literal)
(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 calculated `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.

4.6 IMPLEMENTATIONS

We discuss two IceDust2 compilers. The first compiler closely matches the operational semantics in Section 4.4. 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.

4.6.1 *Compilation to Java*

The compilation to Java closely matches the semantics in Section 4.4. 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 4.16). 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 4.17 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.)

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.

4.6.2 *Compilation to WebDSL*

The second compiler compiles IceDust2 to WebDSL, a domain-specific language for building web applications [Visser, 2007]. The to-WebDSL compiler features all IceDust2 features, including multiplicity lower-bound runtime checks, and the `eventual` calculation strategy. WebDSL differs from

```
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 4.16 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 4.17 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 4.18 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 `Set`s. 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 4.19 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.

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 4.18 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, WebDSL provides `setter` hooks, similar to aspect-oriented pointcuts [Kiczales et al., 1997]. Figure 4.19 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.

4.7 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).

4.7.1 *Conference Management System*

Figure 4.20 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 chapter, 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 4.20, 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`, 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 4.20, bottom). It is similar in structure to the submission parent-children relation in Figure 4.4. 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.

4.7.2 *Learning Management System*

Our running example (Figure 4.4) 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.

```
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 4.20 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.

Figure 4.21 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 4.22 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 4.22 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 4.4) 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

```
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 4.21 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 4.22 Bidirectional relation `next` and `previous` is derived from the ordering of `children`.

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 re-

quire all kinds of boilerplate changes, on top of the semantic changes. This has been argued before for multiplicities [Steimann, 2013], bidirectional relation maintenance [Harkes and Visser, 2014], and calculation strategy switching [Harkes et al., 2016] individually. But combined, it is certainly true as well.

4.8 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 4.22 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: 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 `parent`s 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 4.23 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 4.2) does

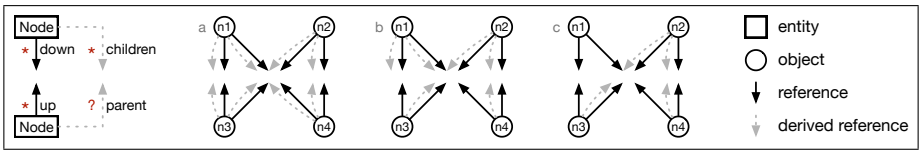


Figure 4.23 Contradictory specification solutions: (a) give up multiplicity bounds, (b) give up bidirectionality, or (3) give up derivation semantics.

not preserve bidirectionality (Figure 4.23b). 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 4.23c). 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 4.22.

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.

4.9 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.

4.9.1 *Derived Bidirectional Relations*

Various languages feature bidirectional relations as a language feature. Rumer [Balzer, 2011], RelJ [Bierman and Wren, 2005], Relations [Harkes and Visser, 2014], and IceDust [Harkes et al., 2016] 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 [Gupta and Mumick, 1995]. Traditional algorithms for materialized views limit recursive aggregation [Gupta et al., 1993]. Some forms of recursive aggregation can be incrementalized [Ramakrishnan et al., 1994; Ross and Sagiv, 1992], but until now the community has not converged to a recursive aggregation technique [Green et al., 2013]. LogiQL [Green, 2015] 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 [Mitschke et al., 2014], Materialized Object Query Language (OQL) [Gluche et al., 1997], and MOVIE [Ali et al., 2003] 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 [Ujhelyi et al., 2015] 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 [Jackson, 2002] (with operational semantics Alchemy [Krishnamurthi et al., 2008]) and Booster [Davies et al., 2006] 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.

4.9.2 *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) [Elliott, 2009], with for example REScala [Salvaneschi et al., 2014], or Scala.React [Maier and Odersky, 2013] 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 [Acar, 2009] and Adapton [Hammer et al., 2014] 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 [Hammer et al., 2015] 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) [Willis et al., 2008], and Demand-Driven Incremental Object Queries (DDIOQ) [Liu et al., 2016] 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 [Demers et al., 1981]. Reference attribute grammars (RAGs) support derived relations [Söderberg and Hedin, 2012]. 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.

4.9.3 *Eventual Calculation without Bidirectional Relations*

Reactive programming (RP), with for example RX [Meijer, 2010], 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 4.14).

4.9.4 *Software Product Lines and Language Engineering*

Völter and Visser have investigated the combination of software product lines (SPLs) and domain-specific languages (DSLs) [Völter and Visser, 2011]. 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 [Porter et al., 2016] 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.

4.10 SUMMARY AND FUTURE WORK

In this chapter 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?

Postscript: IceDust 2

IceDust2 introduces derived bidirectional relations with multiplicity bounds and calculation strategy composition. Correctness for incrementality is guaranteed by the type system, including incremental updates of derived bidirectional relations. However, performance bottlenecks are not prevented by static checks in IceDust2. With the introduction of derived relations, a serious performance caveat was introduced.

The caveat is the specific situation that a derived bidirectional relation with multiple paths to the same object (such as `Submission.parent` in Figure 4.4) is calculated eventually. In this situation the calculation of derived values after a change to base data can take a long time. In IceDust derived value calculation times are dominated by loading and saving data. In the WebLab case study (Chapter 6) calculating a complete course incrementally (in a single transaction) takes a minute, of which half of the time is loading from database and a quarter of the time is saving back to the database. When such a derived relation is calculated eventually, every individual derived value is calculated in a separate transaction. These individual derived values use the same objects during the calculation, causing objects to be loaded into memory many times.

In Chapter 6 we work around this performance bottleneck by using a relational HQL query instead of an IceDust expression for calculating the derived value. This avoids loading objects into memory altogether. However, these HQL queries do not provide multiplicity guarantees. So, in future work it might be interesting to explore automatically transforming queries into expressions and vice versa to obtain both good performance and multiplicity bounds.

PixieDust

Declarative Incremental User Interface Rendering through Static Dependency Tracking [1]

5

Modern web applications are interactive. Reactive programming languages and libraries are the state-of-the-art approach for declaratively specifying such interactive applications. However, programs written with these approaches contain error-prone boilerplate code for efficiency reasons.

In this paper we present PixieDust, a declarative user-interface language for browser-based applications. PixieDust uses static dependency analysis to incrementally update a browser-DOM at runtime, without boilerplate code. We demonstrate that applications in PixieDust contain less boilerplate code than state-of-the-art approaches, while achieving on-par performance.

5.1 INTRODUCTION

Modern web applications are interactive. Data edits do not trigger page reloads, but in-place DOM updates. These DOM updates could be written by hand, but this is a tedious and error-prone exercise. A declarative, but naive, solution would be to rebuild the entire DOM from a declarative render function on each edit. However, DOM operations are slow, so this approach leads to unresponsive interfaces for large applications. Furthermore DOM elements would lose their local state (such as focus and event handlers). Current state-of-the art declarative solutions maintain a virtual DOM, and patch the browser DOM based on the diffs between virtual DOM renders. When data is edited, these solutions compare the view before and after the data edit and apply DOM updates to patch the difference. Since calculating the minimal difference between two trees is $O(n^3)$ [Demaine et al., 2009], these solutions use $O(n)$ non-minimal tree-diffing algorithms. Possible scalability issues with non-minimal tree diffing can be mitigated by identifying which sub-trees need to be updated on a change. However, the programmer is responsible for correctly identifying these sub-trees, which leads to boilerplate code.

In this paper we present PixieDust, a web programming language that enables concise declarative definition of user interfaces by automatic derivation of code to compute incremental view updates based on compile-time static dependency analysis. The contributions of this paper are:

[1] This chapter has appeared as ten Veen, N., Harkes, D. C., and Visser, E. (2018). Pixiedust: Declarative incremental user interface rendering through static dependency tracking. In *Companion of the The Web Conference 2018 on The Web Conference 2018*, pages 721–729. International World Wide Web Conferences Steering Committee.

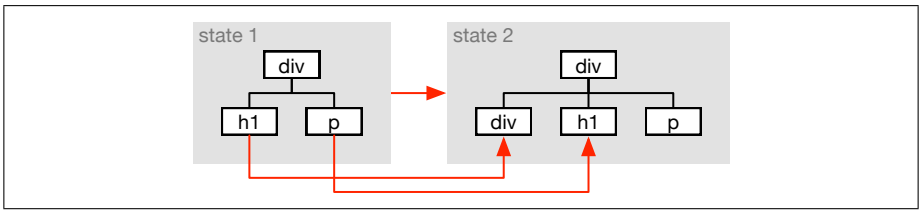


Figure 5.1 By default, the diffing algorithm of existing frameworks compare children in order. Adding a child node at the front causes all children to be completely rerendered. This issue can be fixed by manually adding identities to children.

- The design of the PixieDust language supporting concise and declarative definition of data model and view.
- A static dependency analysis of the impact of model updates to views.
- A mapping of PixieDust programs to an implementation in JavaScript of incremental view updates using the React framework as basis.
- An evaluation showing that the performance of the approach is on-par with state-of-the-art approaches, with a factor 2 reduction in code size.

We proceed as follows. In the next section we analyze the state-of-the-art solutions, to see where error-prone boilerplate code is introduced. In Section 5.3 we propose an approach for static dependency tracking to identify sub-trees for rerendering. In Section 5.4 we present the PixieDust language for specifying data models and declarative views which incorporates this static dependency tracking. In Section 5.5 we formally define the dependency analysis for PixieDust. In Section 5.6 we formally define the operational semantics of PixieDust, detailing its interaction with the browser. In Section 5.7 we evaluate our language design, and in Section 5.8 we compare related work to PixieDust.

5.2 EXISTING APPROACHES

In this section we analyze techniques for efficient DOM updates used by state-of-the-art approaches and we identify problems with these techniques.

5.2.1 *Linear Tree Diffing*

All state-of-the-art approaches use linear tree diffing (for example React [url, 2017b]). Linear tree diffing algorithms compare old and new virtual DOM trees recursively per level. If the tag of a node is equal to the previous version, the browser DOM node remains intact. The attributes of intact nodes are compared, and any differences are patched in the DOM. The children of these nodes are traversed in the next level. If the tags are different, the entire node with its children are removed from the DOM and is rebuilt from scratch.

```
<ul>
  { this.props.todos.map(todo =>
    <li key={todo.id}>
      <TodoView todo={todo}/>
    </li>
  }
</ul>
```

Figure 5.2 Identities (keys) on children increase performance, but add boilerplate code in MobX. This applies to all state-of-the-art solutions.

```
enum TodoActionKeys{ TOGGLE_TODO = "TOGGLE_TODO"}
interface ToggleTodoAction{
  type: TodoActionKeys.TOGGLE_TODO,
  todoId: string
}
type TodoAction = ToggleTodoAction

function toggleTodo(todoId: string) {
  return {
    type: TodoActionKeys.TOGGLE_TODO,
    todoId: todoId
  }
}
function reducer(todos:Todo[],action:TodoAction){
  switch(action.type){
    case TodoActionKeys.TOGGLE_TODO:
      return todos.map(todo =>
        todo.id == action.todoId
          ? {finished: !todo.finished, ...}
          : todo
      );
  }
}
```

Figure 5.3 Boilerplate code needed to dispatch a state update in Redux. The action is encoded as a plain javascript object which gets passed to a pure function by the runtime that processes all possible actions.

When children of a node are reordered, a linear diff algorithm cannot determine the new position of children. This means that instead of reordering the children, the children are replaced by each other. This can be very inefficient, for example when a child is added as first child (Figure 5.1). Adding identities to children enables reordering in linear time (Figure 5.2). However, it is the responsibility of the programmer to find suitable identities for the data structures that are being used and bind them to their sub-trees.

5.2.2 *Identifying which parts of the DOM-tree need updating*

It is unnecessary to diff the entire tree structure when entire parts of the tree do not depend on the changes that were made. If a sub-tree is parameterized by the set of values it depends on, that information can be used to only diff when these values changed. There are multiple approaches to achieve this.

The first approach is to use immutable data. Elm [Czaplicki and Chong, 2013] and Redux [url, 2017c] use this approach. With immutable data structures and pure view functions, reference equality can be used to determine whether a sub-tree needs to be rerendered. When a value changes, only the node where that value is displayed, and the spine to the root of the tree are recalculated. Since immutable data structures cannot contain cycles, programmers need to use a tree structured data model. Since immutable data cannot be updated in place, solutions with immutable data use message passing to encode updates. These messages are dispatched to a pure function calculating the new state based on the previous state. This optimization does come with a lot of boilerplate: each action needs to be encoded in a data structure, and when these actions are decoded, the relevant part of the state needs to be looked up and modified (Figure 5.3).

An alternative approach to localize DOM diffing is to construct a dependency graph for views. That way views can observe writes that are made to their dependencies to trigger a rerender. Hence, calls to setters on data are automatically reflected in the user interface. MobX [url, 2017a] is a framework that constructs the dependency graph dynamically while rendering. To achieve this at runtime, MobX relies on wrapping get and set operations of data. However, this can lead to subtle bugs where a child component is passed a value instead of the getter for that value.

5.2.3 *Summary*

In conclusion, state-of-the-art solutions induce error-prone boilerplate code. All solutions require identity annotations on lists. The immutable data solutions (Elm and Redux) require encoding of data modifications into action objects, and the mutable data solution (MobX) traps getters and setters (which can accidentally be circumvented in JavaScript).

5.3 STATIC DEPENDENCY TRACKING

State-of-the-art client-side application frameworks induce error-prone boilerplate code and their assumptions can be accidentally violated leading to subtle bugs. We propose to use static dependency tracking as a solution to these issues. *Static* dependency tracking does not trap getters and setters at runtime (such as MobX), but instead (over)approximates the dependency structure at compile-time. View definitions reference parts of the data model. These references can be statically determined, and this can be used to decide which views should be rerendered after a data modification.

To illustrate how to statically derive dependencies, we consider a miniature ToDo application (Figure 5.4). A `TodoList` holds zero or more `Todo`s. A `Todo` has a `description` and a `finished` flag. The view for a `TodoList` is a `div` containing a `ul` with a `li` for every item. Every `Todo` is rendered as a checkbox for the `finished` status and a `span` for the `description`.

```
model
  entity TodoList {
    todos : Todo* (inverse = Todo.list)
  }
  entity Todo {
    description : String
    finished    : Boolean
  }

view
  TodoList.view = div { ul { todos.itemView } }

  Todo.itemView = li {
    input[type="checkbox", value=finished]
    span { description }
  }
```

Figure 5.4 Miniature ToDo application data model and view

By analyzing the body of the views, we can collect all referenced paths. The `itemView` references both fields of the `Todo`. The `TodoList.view` references the `itemView` of its `todos`. This means that this view needs to be updated both when a referenced `itemView` changes and when the `todos` list changes itself. Together, the application contains the following dependencies:

```
Todo.itemView <- finished
Todo.itemView <- description
TodoList.view <- todos
TodoList.view <- todos.itemView
```

These dependencies can be inverted to get the data flow of the application. To be able to invert dependencies that reference `todos`, we need an *inverse*. Figure 5.4 defines the inverse of `todos` as `Todo.list`. When we invert dependencies we obtain the following data flow:

```
Todo.finished    -> itemView
Todo.description -> itemView
TodoList.todos   -> view
Todo.itemView    -> list.view
```

This data flow can be used to trigger rerendering of views on data modifications. Moreover, since views are parameterized by an entity, we can automatically assign keys to collections, without unnecessary boilerplate code.

We have designed and implemented PixieDust, a new language for declarative definition of user interfaces in the browser based on this dependency analysis. We will formalize this dependency analysis in Section 5.5, but first we will discuss the design of PixieDust.

5.4 PIXIEDUST

PixieDust is a language for specifying data models and browser-based user interfaces that separates the concerns of model and view, literally by keywords (for example Figure 5.4). Everything defined in the data model is visible in the view, but not vice versa.

5.4.1 *Data Model*

For the data model we use the IceDust data modeling language [Harkes et al., 2016; Harkes and Visser, 2017]. In IceDust, a data model consists of *entities* with *fields*. All fields have a type and a *multiplicity*. The multiplicities in IceDust are 1, ?, $*$, and $+$ (similar to regular expressions and highlighted in red in examples). If multiplicities are omitted, they default to 1. Fields with an entity-type have an *inverse*. Whenever an object refers from such a field to another object, the other object refers back from its inverse field. Lastly, IceDust features *derived value* fields: fields for which the value is calculated. For example, we can extend the `TodoList` in Figure 5.4 with a field indicating whether all todos are finished and how many are left:

```
entity TodoList {
  allFinished : Boolean = conj(todos.finished)
  todosLeft   : Int = countFalse(todos.finished)
}
```

5.4.2 *View*

In PixieDust we define views in the context of an entity (Figure 5.5). The `View` type is a (virtual) DOM node. Inside a view, the fields of the context entity can be concisely accessed by referring to them. Other views of the same entity also can be referenced directly, and views of other entities can be referenced by member access. This makes for concise definitions of views in PixieDust.

The view of a model might contain state. In our example we have the state of the input field for adding new todos. To separate the concerns of data model and view, we do not add this state to the data model, but introduce view state. View state fields can be of any type and are scoped by a context entity (Figure 5.6). View state supports the same kind of derived values as the data model. For example we derive the visible todos collection in Figure 5.6.

User interfaces should support user interaction with the application. In PixieDust, actions declaratively describe data modification (Figure 5.7). Actions are also scoped by an entity, this makes for concise definitions. Both the data model and view state can be accessed within actions. Moreover, new objects can be created (see `addTodo`) and old objects can be left for garbage collection (see `deleteTodo`).

Often an input element reads and writes to a specific field of an entity. One could program an action for each field, but that is tedious. For concise UI specifications, a language should support bidirectional mappings between user interface and data model. PixieDust provides built-in bidirectional mappings for primitive data types (`BooleanInput` and `StringInput` in Figure 5.5). In future work we would like to explore user-defined bidirectional mappings.

5.4.3 *Example*

Figures 5.4–5.7 contain an almost complete specification of a full ToDo application in PixieDust. The only thing missing is the definition of two functions

```
view
  TodoList {
    view : View = div {
      header
      ul { visibleTodos.itemView }
      footer
    }

    header : View = div {
      h1 { "Todos" }
      input[type="checkbox", value = allFinished,
                             onClick = toggleAll]
      StringInput[onClick = addTodo](input)
    }

    footer : View = div {
      todosLeft "items left"
      ul{
        visibilityButton(this, "All")
        visibilityButton(this, "Finished")
        visibilityButton(this, "Not finished")
      }
      if(count(finishedTodos) > 0)
        button[onClick = clearFinished]
    }
  }
  Todo {
    itemView : View = li { div {
      BooleanInput(finished)
      span { task }
      button[onClick=deleteTodo] { "X" }
    }}
  }
```

Figure 5.5 TodoList views: the TodoList view has a header with an input field for adding new todos; a list of all todos; and a footer with the number of todos left, a filter for which todos to show, and a button for removing finished todos.

```
view
  TodoList {
    input : String = (init = ""    )
    show  : String = (init = "All")

    finishedTodos : Todo* =
      todos.filter(todo => todo.finished)
      (inverse = Todo.inverseFinishedTodos?)

    visibleTodos : Todo* =
      switch {
        case show == "All"      => todos
        case show == "Finished" => finishedTodos
        default => todos \ finishedTodos
      }
      (inverse = Todo.inverseVisibleTodos?)
  }
```

Figure 5.6 TodoList view state: the view state contains fields for storing the input (to add a new item), filtering visible items, and computing visible items.

```
view
  Todo {
    actions {
      toggleTodo: finished := !finished
      deleteTodo: list := null
    }
  }
  TodoList {
    actions {
      addTodo:
        todos += {description = input
                  finished = false}
        input := ""

      toggleAll: todos.finished := !allFinished
      clearFinished: todos -= finishedTodos
      setVisibility(to: String): show := to
    }
  }
```

Figure 5.7 ToDo application actions: items can be toggled, deleted and added; and for a list all items can be toggled, all finished items can be deleted, and the filter can be changed.

```
functions
  visibilityButton(l:TodoList, to:String):View =
    li[onClick = l.setVisibility(to)] { to }

  countFalse(bs : Boolean*) : Int =
    count(bs.filter(b => !b))
```

Figure 5.8 Functions in ToDo application facilitate reuse

(Figure 5.8). Together, these figures form a concise specification of a complete ToDo application. Moreover, this application is incremental: derived values are only recalculated and views are only rerendered when needed.

5.5 DEPENDENCY AND DATA-FLOW ANALYSIS

In Section 5.3 we introduced static dependency tracking as a way to get rid of error-prone dynamic dependency boilerplate code. In this section we formalize this static dependency analysis. The analysis is based on the dependency analysis of IceDust [Harkes et al., 2016]. In this paper we extend it with analysis for functions and views.

5.5.1 *Dependencies between Fields in Data Model*

First, we recap the analysis of dependencies between fields from IceDust. To illustrate the analysis we extend Figure 5.4 with `allFinished` which is the `conj`unction of the finished fields:

```
allFinished : Boolean = conj(todos.finished)
```

The dependencies of a field are all fields which are needed to compute the derived value of that field. The dependencies are reachable from the entity containing the field via a path. A dependency is denoted by $(Ent.Field \leftarrow \pi)$, where $Ent.Field$ is a field and π is the path to a field.

Computing the dependencies requires extracting paths from expressions defining field values. The *path-based abstract interpretation* relation (Figure 5.9) defines the dependency paths of an expression. We use the notation $(Expr \searrow \{\pi\}\{\rho\})$, where $Expr$ is the expression that is abstractly interpreted, and $\{\pi\}$ and $\{\rho\}$ are the sets of paths defined by the abstract interpretation. The paths in $\{\pi\}$ are extended by surrounding expressions, while the paths in $\{\rho\}$ are not. The `if` only extends paths in the second and third operand, so Π_1 is passed to $\{\rho\}$. All paths start with `this` [This] or with navigation [NavStart]. When navigating by means of `e.m` all dependency paths in $\{\pi\}$ are extended with `.m` [Nav]. Operators just pass on all paths [UnOp, BinOp], and literals do not contain any paths [Literal]. Path-based abstract interpretation of the expression defining `allFinished` produces a set with a single path:

```
{ todos.finished }
```

The *dependencies* relation (Figure 5.9) defines the dependencies of a field and a full program. We use the notation $Field|Prog \searrow\searrow \{(Ent.Field \leftarrow \pi)\}$ where $Field|Prog$ is a field or full program, and $\{(Ent.Field \leftarrow \pi)\}$ is a set of dependencies. When a field depends on the value at the end of a path, it also depends on the relations en route. So the rule for fields [Field] takes the transitive prefix of the paths of its expression. As paths are concatenated later, the `this` is removed from paths. The paths for our example are:

```
(TodoList.allFinished ← todos.finished)
(TodoList.allFinished ← todos)
```

The data flow from a field is the set of all fields that depend on it to compute their value. The data flow relation is the inverse of the dependency relation. We use $(Ent.Field \rightarrow \pi)$ to denote the data flow relation from the source, $Ent.Field$, to the target, the end of the path π.

The *dependency inversion* relation, $(Ent.Field \leftarrow \pi) \nearrow (Ent\ .Field \rightarrow \pi)$, in Figure 5.10 defines the inverse of a dependency. A dependency is inversed by swapping source and target, and inverting the path π to get the path from target to source. The function `inv-path(`π`)` inverts the names on the path, and inverts their order. Name inversion is selecting the name on the opposing side of a bidirectional relation. The resulting data flow in our example is:

```
(TodoList.todos → allFinished)
(Todo.finished  → list.allFinished)
```

5.5.2 *Dependencies with Filter, Find, and OrderBy*

Note that IceDust 2 [Harkes and Visser, 2017] introduced `filter`, `find`, and `orderBy`, but did not document the dependency analysis for these. To illustrate the analysis of these, consider adding the following to `TodoList`:

```
numLeft:Int = count(todos.filter(x=>!x.finished))
```

Path-based abstract interpretation $\boxed{Expr \searrow \{\pi\}\{\rho\}}$

$$\frac{}{m \searrow \{m\}\{\}} \text{[NavStart]} \qquad \frac{}{\text{this} \searrow \{\text{this}\}\{\}} \text{[This]}$$

$$\frac{e \searrow \Pi\ \mathrm{P}}{e\ .\ m \searrow \{\pi\,.m \mid \pi \in \Pi\}\ \mathrm{P}} \text{[Nav]} \qquad \frac{e \in \mathit{Literal}}{e \searrow \{\}\{\}} \text{[Literal]}$$

$$\frac{\oplus \in \mathit{UnOp} \quad e \searrow \Pi\ \mathrm{P}}{\oplus\, e \searrow \Pi\ \mathrm{P}} \text{[UnOp]}$$

$$\frac{\oplus \in \mathit{BinOp} \quad e_1 \searrow \Pi_1\ \mathrm{P}_1 \quad e_2 \searrow \Pi_2\ \mathrm{P}_2}{e_1 \oplus e_2 \searrow \Pi_1 \cup \Pi_2 \quad \mathrm{P}_1 \cup \mathrm{P}_2} \text{[BinOp]}$$

$$\frac{e_1 \searrow \Pi_1\ \mathrm{P}_1 \quad e_2 \searrow \Pi_2\ \mathrm{P}_2 \quad e_3 \searrow \Pi_3\ \mathrm{P}_3}{e_1\,?\,e_2 : e_3 \searrow \Pi_2 \cup \Pi_3 \quad \Pi_1 \cup \mathrm{P}_1 \cup \mathrm{P}_2 \cup \mathrm{P}_3} \text{[If]}$$

$$\frac{\begin{array}{l}\oplus \in \{\text{filter}, \text{find}, \text{orderBy}\} \quad e_1 \searrow \Pi_1\ \mathrm{P}_1 \quad e_2 \searrow \Pi_2\ \mathrm{P}_2 \\ \Pi_2' = \text{replace-id*}(\Pi_2, x, \Pi_1) \quad \mathrm{P}_2' = \text{replace-id*}(\mathrm{P}_2, x, \Pi_1)\end{array}}{e_1.\oplus(x => e_2) \searrow \Pi_1 \quad \Pi_2' \cup \mathrm{P}_1 \cup \mathrm{P}_2'} \text{[Col]}$$

$$\frac{\begin{array}{l}f.\text{expr} \searrow \Pi_f\ \mathrm{P}_f \quad e_i \searrow \Pi_i\ \mathrm{P}_i \\ \mathrm{P}_e = \bigcup_{i=1..n} \mathrm{P}_i \quad \Pi_e^{\text{named}} = \{(f.\text{args}[i], \Pi_i) \mid i \in 1..n\} \\ \Pi_f' = \text{replace-ids}(\Pi_f, \Pi_e^{\text{named}}) \quad \mathrm{P}_f' = \text{replace-ids}(\mathrm{P}_f, \Pi_e^{\text{named}})\end{array}}{f(e_1, \ldots, e_n) \searrow \Pi_f' \quad \mathrm{P}_f' \cup \mathrm{P}_e} \text{[Fun]}$$

$$\begin{array}{l}\text{replace-id}(x.\pi,\ x,\ \pi_2) = \pi_2.\pi \\ \text{replace-id}(\pi, x,\ \pi_2) = \pi \\ \text{replace-id*}(\Pi_1,\ x,\ \Pi_2) = \{\text{replace-id}(\pi_1, x, \pi_2) \mid \pi_1 \in \Pi_1,\ \pi_2 \in \Pi_2\} \\ \text{replace-ids}(\Pi_1, []) = \Pi_1 \\ \text{replace-ids}(\Pi_1, [(x, \Pi_2)|t]) = \text{replace-ids}(\text{replace-id*}(\Pi_1, x, \Pi_2), t)\end{array}$$

Dependencies $\boxed{\mathit{Field}|\mathit{Prog} \searrow\!\searrow \{(\mathit{Ent.Field} \leftarrow \pi)\}}$

$$\frac{m.\text{expr} \searrow \Pi\ \mathrm{P} \quad e = m.\text{entity} \quad \Pi_2 = \bigcup \{\text{trans-pref}(\text{remove-this}(\pi)) \mid \pi \in \Pi \cup \mathrm{P}\}}{m \in \mathit{Field} \searrow\!\searrow \{(e.m \leftarrow \pi) \mid \pi \in \Pi_2\}} \text{[Field]}$$

$$\frac{\Pi = \bigcup \{\mathit{dep} \mid m \searrow\!\searrow \mathit{dep},\ m \in e.\text{fields},\ e \in p.\text{entities}\}}{p \in \mathit{Prog} \searrow\!\searrow \Pi} \text{[Prog]}$$

$$\begin{array}{l}\text{remove-this}(\text{this}\ .\ \pi) = \pi \\ \text{remove-this}(m\ .\ \pi) = m\ .\ \pi \\ \text{trans-pref}(\pi\ .\ m) = \{\pi\ .\ m\}\ \cup\ \text{trans-pref}(\pi) \\ \text{trans-pref}(m) = \{m\}\end{array}$$

Figure 5.9 Dependency relation by path extraction

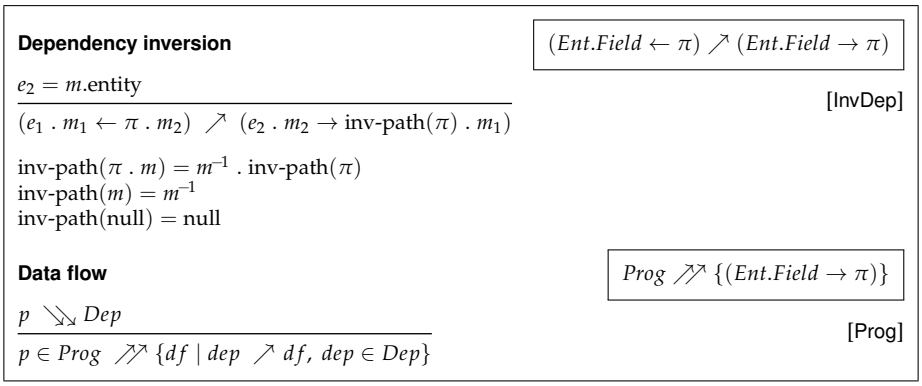


Figure 5.10 Data flow relation by inverting dependencies

The rule [Col] (Figure 5.9) covers these expressions containing a lambda. The occurrence of the parameter `x` in the paths of the body of the lambda are replaced with the paths of the argument. For our example replacing the `x` in `x.finished` with {`todos`} yields:

```
{ todos.finished }
```

5.5.3 *Dependencies with Functions*

In this paper we extend the dependency analysis with support for functions. As an example for functions we use our specification of `todosLeft` by using a function for counting the number of elements equal to `false`:

```
todosLeft : Int = countFalse(todos.finished)

function countFalse(bs : Boolean*) : Int =
  count(bs.filter(b => !b))
```

Rule [Fun] in Figure 5.9 covers user-defined functions. The dependency paths of a function call are defined as the dependency paths of the function definition expression, with all occurrences of argument names replaced by the paths of the arguments at the call site. Note that these are all sets of paths, so functions `replace-id*` and `replace-ids` operate on sets. If we apply the analysis to our example, the paths of the function body are:

```
{ bs }
```

The call from `numLeft` has the following *named* paths:

```
( bs => { todos.finished } )
```

Applying replacement yields the dependencies for `numLeft`:

```
{ todos.finished }
```

Note that this is identical to our original definition of `numLeft`.

PixieDust does not support direct recursive functions. In order to provide incremental behavior each recursive step should be cached. So recursion is

$$
\begin{aligned}
\Sigma \in \mathit{Entity} &: \mathit{EntityRef} \times \mathit{Field} \mapsto \\
&\quad (\text{val} \mapsto [\mathit{Value}],\ \text{cache} \mapsto [\mathit{Value}],\ \text{dirty} \mapsto \mathit{Boolean},\ \text{subs} \mapsto [\mathit{ComponentRef}]) \\
C \in \mathit{Component} &: \mathit{ComponentRef} \mapsto (\ o \mapsto \mathit{EntityRef},\ f \mapsto \mathit{Field},\ \text{mounted} \mapsto \mathit{Boolean}\ ) \\
Q \in \mathit{Queue} &: \{\mathit{ComponentRef}\} \\
F \in \mathit{Frame} &: \mathit{ObjectRef} \\
\mathit{Value} &: \mathit{EntityRef} \mid \mathit{PrimitiveValue} \mid \mathit{VirtualDOMElem}
\end{aligned}
$$

Figure 5.11 The PixieDust runtime has four stores. The entity store (Σ) maps object fields to user value, cached value, dirty flag, and subscribed components. The component store (C) maps components to object fields and a mounted flag. The queue (Q) is a global list of elements that need to be rerendered, and the frame (F) is a reference to a requested animation frame.

supported through materializing the intermediate results in a field. For example,

```
entity Node {
  children : Node* (inverse = Node.parent?)
  cnt : Int = count_descendants(this)
}

function count_descendants(n : Node) : Int =
  count(n.children) + sum(n.children.cnt)
```

5.5.4 *Dependencies between Views*

In Section 5.4 we introduced view state and `Views` as a new data type for fields in the view state. The dependency analysis treats view state fields equal to data model fields. However, views (fields of type `View`) that are related through containment, do not depend on each other. Views are updated *in place* inside the DOM, so 'parent' views do not have to be notified of change. We will cover this in more detail in the next section.

5.6 OPERATIONAL SEMANTICS

In this section we describe the dynamic semantics of rendering PixieDust applications. Our compiler (PixieDust-to-JavaScript) uses the React rendering framework. Analogously, our semantics use semantic functions which correspond to the React and browser APIs calls and callbacks at runtime. Our semantics extend the IceDust 2 semantics for incremental calculation [Harkes and Visser, 2017]. (Semantic functions are typeset in bold, and IceDust 2 calls are typeset in italic.)

We specify the operational semantics of PixieDust using big-step semantics. The reduction rules modify four stores (Figure 5.11). The first store (Σ) is the IceDust data store. We extend this store to include a list of components which should be notified of change per field: subscriptions. Note that we also store view state and rendered virtual DOMs in this store. The second store (C) contains meta data for React Components: which view-state field

Data modifications $\boxed{\mathit{Statement}/\Sigma, C, Q, F \Downarrow /\Sigma, C, Q, F}$

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

$$\frac{Q_2 = Q \cup \{c\} \quad \text{subscribe}() \Downarrow}{\text{schedule}(c)/Q \Downarrow /Q_2} \text{ [Schedule]}$$

$$\frac{F = \text{null} \quad F_2 = \textbf{requestAnimationFrame}(s))}{\text{subscribe}()/F \Downarrow /F_2} \text{ [SubFrame1]}$$

$$\frac{F \neq \text{null}}{\text{subscribe}() \Downarrow} \text{ [SubFrame2]}$$

Figure 5.12 Evaluation rules for modifications to data

Rendering $\boxed{\mathit{Statement}/\Sigma, C, Q, F \Downarrow /\Sigma, C, Q, F}$

$$\frac{[c.\textbf{forceUpdate}() \Downarrow \mid c \in Q, C[c, \text{mounted}] = \text{true}] \quad Q_2 = \emptyset \quad \text{unsubscribe}() \Downarrow}{\textbf{onAnimationFrame}(s)/Q \Downarrow /Q_2} \text{ [Render]}$$

$$\frac{F \neq \text{null} \quad \textbf{cancelAnimationFrame}(F) \quad F_2 = \text{null}}{\text{unsubscribe}()/F \Downarrow /F_2} \text{ [UnsubFrame1]}$$

$$\frac{F = \text{null}}{\text{unsubscribe}() \Downarrow} \text{ [UnsubFrame2]}$$

Figure 5.13 Evaluation rules for render

contains the rendered virtual DOM, and whether the component is currently mounted. The third store (Q) is a queue of views scheduled for rerendering, and the fourth store (F) refers to the next requested animation frame. In our rules we omit stores if they are not modified. When a store is omitted, it is implicitly threaded from left to right.

The evaluation rules are designed such that we only rerender views when needed, and only rerender them at most once per data modification. The rules in Figure 5.12 define what to do on data modifications. We override IceDust's [FlagDirty] rule to schedule renders on all subscriptions as soon as a field is marked as dirty. This does not rerender those views directly, but schedules them in the queue [Schedule]. Moreover, if this was the first view to be scheduled for rerender, we schedule a browser rerender with **requestAnimationFrame**. This method tells the browser that we want to perform an action before the next frame will be painted. In this way we can batch all effects of data modifications on the UI, avoiding double rerendering.

The rules in Figure 5.13 define what to do on a render. When the browser wants to display the next frame, it will call **onAnimationFrame**. On this call, the PixieDust runtime forces all mounted React components to be rerendered

Component life cycle $\quad \textit{Statement}/\Sigma, C, Q, F \Downarrow /\Sigma, C, Q, F$

$$\frac{C_2 = C[c, \text{mounted} \mapsto \text{true}] \quad c.o.\text{subDirty}(c.f, c) \Downarrow}{c.\textbf{componentDidMount}()/C \Downarrow /C_2} \quad \text{[Mount]}$$

$$\frac{C_2 = C[c, \text{mounted} \mapsto \text{false}] \quad c.o.\text{unsubDirty}(c.f, c) \Downarrow}{c.\textbf{componentWillUnmount}()/C \Downarrow /C_2} \quad \text{[Unmt]}$$

$$\frac{c.o \neq o_2 \quad c.o.\text{unsubDirty}(c.f, c) \Downarrow \quad C_2 = C[c, o \mapsto o_2] \quad o_2.\text{subDirty}(c.f, c) \Downarrow}{c.\textbf{componentWillReceiveProps}(o_2)/C \Downarrow /C_2} \quad \text{[Props1]}$$

$$\frac{c.o = o_2}{c.\textbf{componentWillReceiveProps}(o_2) \Downarrow} \quad \text{[Props2]}$$

$$\frac{[v.\text{addSubscriber}(f_2, c) \Downarrow | v \in V, o \vdash expr \Downarrow V, \\ expr.f_2 \in f.\text{depends}, \neg\text{isView}(f_2)]}{o.\text{subDirty}(f, c) \Downarrow} \quad \text{[Subscribe]}$$

$$\frac{\Sigma_2 = \Sigma[o, f, \text{subs} \mapsto \Sigma[o, f].\text{subs} \cup [c]]}{o.\text{addSubscriber}(f, c)/\Sigma \Downarrow /\Sigma_2} \quad \text{[AddSub]}$$

$$\frac{[v.\text{removeSubscriber}(f_2, c) \Downarrow | v \in V, o \vdash expr \Downarrow V, \\ expr.f_2 \in f.\text{depends}, \neg\text{isView}(f_2)]}{o.\text{unsubDirty}(f, c) \Downarrow} \quad \text{[UnSub]}$$

$$\frac{\Sigma_2 = \Sigma[o, f, \text{subs} \mapsto \Sigma[o, f].\text{subs} \setminus [c]]}{o.\text{removeSubscriber}(f, c)/\Sigma \Downarrow /\Sigma_2} \quad \text{[RemoveSub]}$$

$$\frac{V = c.o.\text{get}(c.f)}{c.\textbf{render}() \Downarrow V} \quad \text{[Render]}$$

Figure 5.14 Evaluation rules for component life cycle

with **forceUpdate** [Render]. React then updates the browser DOM with the diffs from the virtual DOM, before the next frame is rendered.

In this process, React will call various life cycle callbacks on components. Figure 5.14 defines what happens on various life cycle callbacks. The goal of these rules is to maintain a precise list of which data from the entity store is visible through views. First, rules [Mount, Unmt] keep track of whether components are currently mounted in the browser-DOM. Non-mounted components are not forced to update on a render [Render]. Second, the rules in Figure 5.14 maintain the *subs* fields in the entity store. The *subs* fields only contain components which depend on the field, and which are mounted. Note that we never have subscribers for view-typed fields [AddSub], since views are updated in place in the DOM (as discussed in Section 5.5). This way, only the minimal number of components is scheduled for rerendering when data is modified.

	PixieDust	MobX/React	React	React/Redux	Elm
LOC	74	193	259	276	300

Table 5.1 Lines of code for different todo list implementations. Implementations are stripped of features that are not shared between other implementations.

Finally, when React wants to update a view it calls **render**. This call is forwarded to IceDust's incremental evaluation for derived values which computes the virtual DOM for that view [Render].

Together, these evaluation rules minimize the amount of rerendering. In the next section we will evaluate the performance of our implementation. In this semantics we did not cover how actions work. However, the execution of actions is fairly straightforward, and we want to focus this paper on incremental rendering.

5.7 EVALUATION

We evaluate PixieDust with respect to two criteria: (1) reduction of error-prone boilerplate code, and (2) performance relative to state-of-the-art approaches. Our running example in this paper has been a ToDo application. More precisely, it is exactly the application from todomvc.com. TodoMVC compares frameworks through implementations of this ToDo application. We use this application to compare conciseness and performance.

5.7.1 *Conciseness*

The goal of PixieDust is to remove error-prone boilerplate code. To asses this, we look at the number of lines of code of the todo application in different approaches. We have taken the reference implementations for TodoMVC of MobX and vanilla React from todomvc.com, the implementation for Redux from their repository[2], and the implementation for Elm from the author of Elm[3]. Since not all implementations have the same features, we stripped off features that are not shared between all todo implementations. We used `cloc` for counting the lines of code, except for PixieDust which we had to count by hand.

The results are compiled in Table 5.1. Indeed, the PixieDust programs are more concise than the same programs in the state-of-the-art approaches. This is expected, as PixieDust is a domain-specific language with tailored syntax, while the state-of-the-art approaches are JavaScript libraries or general purpose languages.

[2] https://github.com/reactjs/redux/tree/master/examples/todomvc
[3] https://github.com/evancz/elm-todomvc

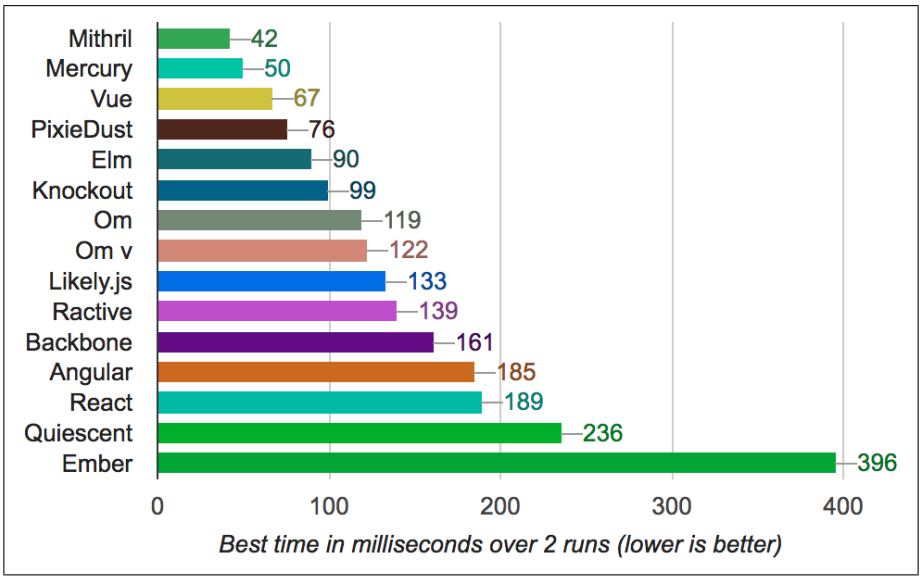


Figure 5.15 TodoMVC online performance benchmark shows PixieDust performs comparable.

5.7.2 *Performance*

The *todomvc performance benchmarks* is an existing online benchmark suite for TodoMVC[4]. This benchmark adds 50 tasks to a single todo list, marks all of them completed one by one and deletes them afterwards. We added an entry for a PixieDust implementation of the Todo application. The results of this benchmark can be seen in Figure 5.15. PixieDust has on-par performance according to this benchmark.

Unfortunately, the TodoMVC benchmark does not benchmark all features. Moreover, the implementations of the various state-of-the-art systems have not been kept up to date (last commit November 2015). So, we created a new benchmark that considers more features[5]. To make the benchmark more representative for larger applications we extended the ToDo application to support todo items which are lists themselves. A list is finished if all child lists and items are finished:

```
entity TodoList {
  children : TodoList*(inverse=TodoList.parent?)
  allFinished : Boolean =
    conj(children.allFinished) and
    conj(todos.finished)
}
```

None of the TodoMVC entries featured nested todo lists. Since MobX is closest in conciseness and also based on mutable data structures, we've extended its implementation with nested lists to compare against. Our test can be pa-

[4] https://github.com/featurist/todomvc-perf-comparison
[5] https://github.com/besuikerd/rendering-options

Framework	Depth	Children	Todos	#Actions
Balanced	4	3	5	1120
Deep	10	1	5	280
Deeper	25	1	5	700
Wide	2	100	2	1414
Leaves	1	1	100	475

Table 5.2 Test properties for benchmarking (depth, degree, and number of leaves of nested ToDo tree) and total number of user interactions performed during execution trace.

rameterized by several properties that influences the size and shape of the nested todo list:

- **Depth** defines the depth of nested todo lists from the root.
- **Children** defines how many child lists are added per list.
- **Todos** defines how many todos are added per list.

We run the benchmark on five data sets (Table 5.2). A test trace executes the following steps. The input field of the root list is selected. For each todo that needs to be added, three alphabetic characters are entered and the enter key is pressed to add it to the list. Next, the toggle all button is pressed twice to select and deselect all todos of the list and its children. After that, half of the todos of the list are finished one by one, and then one third of the todos are deleted individually. After this, all the filters are selected once, and the "Clear finished todos" button is pressed. Finally, if we have not yet reached the required depth, the specified amount of child lists is added to the list and this procedure is recursively repeated for each child.

To ensure that no renders are skipped, each action awaits the next animation frame before executing. The timings are recorded with the Chrome runtime performance recorder which reports scripting (executing JavaScript), rendering (the browser painting), and other (not categorized). During a test, the number of times a specific view component is rendered is counted. The todo list application has four components: **Header**, **Footer**, **List**, and **Todo**. The benchmarks were performed on a 2017 Macbook Pro laptop with Intel Core i7 2,6Ghz, 4 cores (8 threads), and 16 GB memory.

The results of the benchmark are compiled in Figures 5.16 and 5.17. In general, most tests have the same total execution time between frameworks, but the rerenders counts vary.

First, MobX renders the Todo view significantly more often than PixieDust. Whenever a new task is added to a list, all todo items are rendered again. This is caused by the fact that while rendering the list header, the derived value `allFinished` is calculated, which calls the getter on the finished field of each todo through the `finishedTodos` derived value. In the 'Leaves' test, MobX also spends significantly more time processing JavaScript, presumably for this very reason.

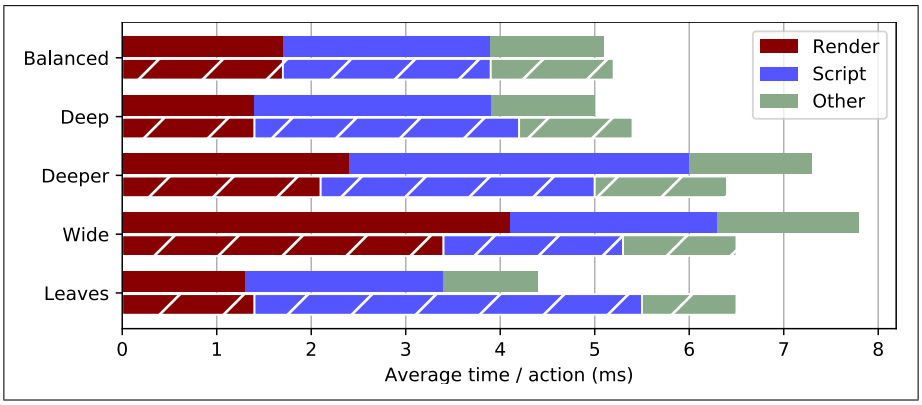


Figure 5.16 Average time per action on tests from Table 5.2. Solid bars are PixieDust, striped bars are MobX/React.

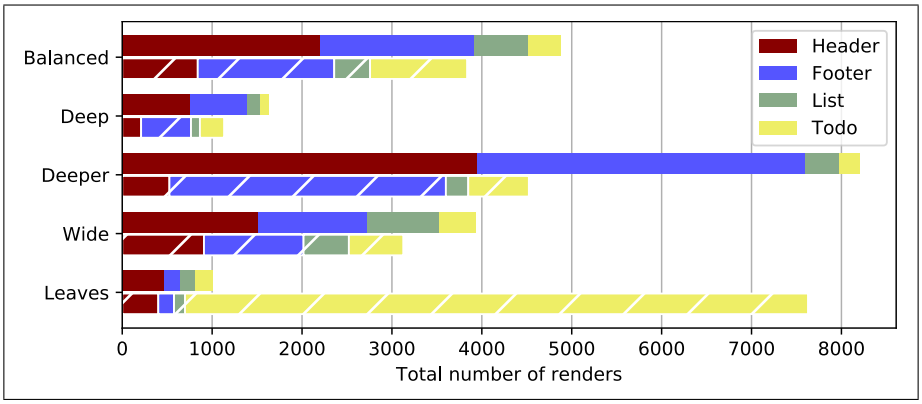


Figure 5.17 Total number of renders on tests from Table 5.2. Solid bars are PixieDust, striped bars are MobX/React.

Second, PixieDust renders the header component significantly more often when the depth is larger. This is caused by dirty flagging `allFinished` transitively along the spine of the tree whenever a modification is done in a todos list. Even when the value stays the same, a render is triggered. This is a limitation of lazy incrementality. In future work we might explore eager incremental evaluation which can detect if a dirty flagged value stays the same.

In conclusion, PixieDust outperforms MobX in some situations, and is outperformed by MobX in other situations. In general, PixieDust's performance is on-par with MobX while reducing lines of code.

5.8 RELATED WORK

The related work is organized in two groups: reactive user-interface languages and incremental computing. The first group we divided in functional (immutable data) and declarative approaches.

Functional Reactive UIs Elm is a functional reactive language for graphical user interfaces [Czaplicki and Chong, 2013]. Newer versions of Elm dropped the support for signals in favor of a simpler model. An application is split up in three parts: The model, the view and the update logic. The update logic takes events that might be triggered by the view or other sources and recompute the next state. While this model gives a clear separation of concerns, it does involve boilerplate code to achieve this.

Redux [url, 2017c] embraces the same pattern but integrates it in React and Javascript as a library. It has the same advantages and disadvantages as Elm. We covered the issues with these approaches in detail in Section 5.2.

Flapjax is a Javascript library for defining web applications using behaviors and event streams [Meyerovich et al., 2009]. In Flapjax data flow can be manually constructed by combining event sources and piping these to sinks. This model enables reactive programming, but hooking up reactive values to the DOM is still manual. Furthermore, the programmer is responsible for identifying where to hook up reactive values, which is error prone.

Reynders et.al. implemented a FRP library in Scala [Reynders et al., 2017]. They analyze different design trade-offs for FRP libraries that interact with the DOM. Based on these trade-offs, they implement a DOM UI library that uses push-pull FRP. Our approach also uses push-pull, push for marking things dirty, and pull for calculating by need. However, in our approach this behavior is hidden behind a declarative language.

UI.Next [Fowler et al., 2015] is a UI library in F#. It connects data sources to views by creating a dynamic data flow graph. The monoidal structure of its DOM elements enables composition of views. It requires higher-order functions to compose, which makes the code less declarative.

Declarative Reactive UIs MobX [url, 2017a] is a state management library. By annotating the variables in a data structure which change over time, MobX can construct a dependency graph at runtime. In contrast, our approach does static runtime dependency tracking. We covered MobX extensively in Section 5.2.

Reactive variables [Schuster and Flanagan, 2016] aim to reduce the boilerplate in programming with signals by adding syntactic sugar for reactive variables. These reactive variables are similar to our approach in the sense that they hide the fact that these variables have a `Signal<T>` type. Their approach is also compiled to JavaScript, but they do not detail how to interact with the DOM.

Mobl [Hemel and Visser, 2011] is a language to declaratively construct interactive mobile applications. The data model defines entities and bidirectional relations between entities, similar to the data model we use. Views can be parameterized by these entities which can be modified via input events. However, their interface language is geared toward phone screens, while ours is focused on browser-DOMs.

Incremental Computing IceDust [Harkes et al., 2016; Harkes and Visser, 2017] is a declarative data modeling language with derived values and bidirectional

relations. It features incremental calculation for derived values. However, it does not have any support for views. In this paper we have extended their approach for incremental computing to cover views in the browser.

Functional Reactive programming can be used for incremental computing. In FRP implementations, like REScala [Salvaneschi et al., 2014], signals propagate through their dependencies. That means that when a value changes, only relevant parts of the data flow are recalculated. However, this approach does not suffice for browser-based views. Because the DOM is a tree structure, composed views will propagate their signals up the spine of the tree, which triggers unnecessary rerenders.

5.9 CONCLUSION

In this paper we have presented PixieDust, a declarative user-interface language for browser-based applications. PixieDust uses static dependency analysis to incrementally update a browser-DOM at runtime. We have demonstrated that applications in PixieDust contain less boilerplate code than state-of-the-art approaches, while achieving on-par performance.

Our research also raises new research questions. First, can we refine our approach so it will perform better? Will eager incremental calculation of views, with the ability to short-circuit updates if values stay the same, perform better? And second, what would be a good language design for user-defined bidirectional mappings between data model and user interface?

WebLab Case Study

Migrating Business Logic to an Incremental Computing DSL: A Case Study [1]

6

To provide empirical evidence to what extent migration of business logic to an incremental computing language (ICL) is useful, we report on a case study on a learning management system. Our contribution is to analyze a real-life project, how migrating business logic to an ICL affects information system validatability, performance, and development effort.

We find that the migrated code has better validatability; it is straightforward to establish that a program 'does the right thing'. Moreover, the performance is better than the previous hand-written incremental computing solution. The effort spent on modeling business logic is reduced, but integrating that logic in the application and tuning performance takes considerable effort. Thus, the ICL separates the concerns of business logic and performance, but does not reduce total effort. Our case study relies on IceDust. IceDust builds on WebDSL and the Spoofax language workbench.

6.1 INTRODUCTION

Information systems are systems for the collection, organization, storage, and communication of information. Information systems aim to support operations, management, and decision-making. In order to do this, the data in information systems is filtered and processed to create new data: derived data. Often these information systems contain large amounts of data and receive frequent updates to this data. The derived data should be updated as base data is updated, and that should happen fast. However, realizing a high performance implementation typically requires invasive changes to the basic business logic in the form of cache and cache invalidation code. Unfortunately, this obfuscates the original intent of the business logic in an abundance of caching patterns. These programming patterns are an obstacle to understanding of programs by human readers [Felleisen, 1990], and thus reduce validatability. In other words, it is less straightforward to establish that a program 'does the right thing'.

Incremental computing languages (ICLs) aim to address this tension between performance and validatability by automatically incrementalizing non-incremental specifications. Since none of the existing ICLs could express business logic of our information systems concisely, we created IceDust, an ICL

[1] This chapter has appeared as Harkes, D. C., van Chastelet, E., and Visser, E. (2018). Migrating business logic to an incremental computing DSL: a case study. In Pearce, D. J., Mayerhofer, T. and Steimann F., editors, *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*.

with support for recursive aggregation and composition of multiple incremental computing strategies.

Contribution To provide empirical evidence to what extent migration of business logic to an ICL is useful, we report on a case study on a learning management system: WebLab. Our contribution is to analyze a real-life project, how migrating business logic to an ICL affects validatability and performance, and how much effort migration takes.

Audience We target language engineering researchers (interested in empirical data justifying their work or looking for new problems to solve) and information system developers (seeking to understand how ICLs can help them in practice).

Structure We organize this chapter according to the structure proposed for case studies by Runeson et al. [Runeson et al., 2012] and Yin [Yin, 2013], similar to a recent case study by Voelter et al. [Völter et al., 2015]. We start with the background on information systems, language engineering, and incremental computing languages in Section 6.2. Section 6.3 introduces our research questions and collected data. Section 6.4 describes the relevant context of the case study, including the architecture, the development timeline, and team composition as suggested by Dyba et al. [Dybå et al., 2012]. Section 6.5 provides an overview of the IceDust-based implementation of WebLab. Section 6.6 answers the research questions. Section 6.7 discusses validity. Section 6.8 contrasts our work to related work, and we conclude in Section 6.9.

6.2 BACKGROUND

In this section we cover the background information for this case study: information system engineering, language engineering, and incremental computing.

6.2.1 *Web-based Information System Engineering*

Nowadays, many applications — including information systems — are web applications, as these are easily accessible. This is illustrated by the fact that the most widely used programming, scripting, and markup languages by Stack Overflow users in 2018 were JavaScript, HTML, and CSS [url, 2018].

Many organizations have unique requirements for their information systems. Organizations differ on the exact structure of their data, who gets access to what data, what derived data is computed, and how many users concurrently use the system. Consequently, many organizations use custom-built information systems. Most of these organizations do not require a large-scale infrastructure for their web-based information system, usually a single-shard web-server suffices. While new storage technologies are on the rise, the predominant technology used for storage of data for information systems is relational databases. The code interacting with databases is usually object-

oriented, as illustrated by the Correlated Technologies in the same developers survey [url, 2018].

Developing information systems poses challenges. One of these challenges is bridging the gap between domain concepts and the encoding of these concepts in a programming language [Jackson, 2006; Felleisen, 1990; Visser, 2015]. The validatability of a program is a measure of the size of this gap. Better validatability, a smaller gap between intent and encoding, makes it straightforward to establish that a program 'does the right thing'. Another challenge is performance [Zeng et al., 2016; Green, 2015]. Realizing a high performance implementation typically requires invasive changes to a basic expression of intent, reducing validatability. The last challenge is reducing the effort spend on engineering information systems, as budget overruns and delays are a constant problem for information system engineering [Yeo, 2002].

6.2.2 *Incremental Computing Languages and IceDust*

Incremental computing is a software feature which, when a piece of data changes, attempts to save time by reusing previous results to compute new results [Carlsson, 2002; Acar, 2005; Demetrescu et al., 2011]. This can be orders of magnitude faster than computing new results from scratch. A programming language is an incremental computing language if all programs written in it use incremental computing. ICLs can be general purpose, for example self-adjusting computation [Acar, 2005] and Adapton [Hammer et al., 2014]; or domain-specific, such as type system languages [Szabó et al., 2016], array computation languages [McSherry et al., 2013; Zhao et al., 2017], and SQL materialized views [Gupta and Mumick, 1995; Koch et al., 2014].

IceDust [Harkes et al., 2016; Harkes and Visser, 2017] is a domain-specific ICL for the domain of information systems. It targets small to medium-sized information systems of small organizations that run on a single shard server and are programmed with a relational database and an object-oriented language. In IceDust, a data model and derived values can be specified. These derived values can be calculated by a variety of incremental calculation strategies [Harkes et al., 2016]. Moreover, these calculation strategies can also be composed [Harkes and Visser, 2017]. IceDust uses static dependency tracking so that for each page request only the relevant data needs to be loaded in memory (dynamic approaches require all data in memory or persistence of the dynamic dependency graph).

6.2.3 *Language Engineering with Spoofax*

Language engineering refers to building, extending and composing general-purpose and domain-specific languages [Völter et al., 2013]. Language workbenches [Fowler, 2005; Erdweg et al., 2013] are tools for efficiently implementing languages and their integrated development environments (IDEs). Spoofax is a language workbench for developing textual (domain-specific) programming languages [Kats and Visser, 2010]. Spoofax provides meta-

languages for high-level declarative language definition. It provides an interactive environment for developing languages using these meta-languages. Moreover, it produces parsers, type checkers, compilers, interpreters, and other tools from these language definitions.

6.3 CASE STUDY SETUP

Our goal is to find out the degree to which ICLs are useful for developing information systems. We adopt the case study method to investigate the use of IceDust in a mission critical project, as we believe that the true risks and benefits of DSLs can be observed only in such projects. Focusing on a single case allows us to provide significant details about that case.

To structure the case study, we introduce three specific research questions in Section 6.3.1. They are aligned with the general challenges for information systems discussed in Section 6.2.1. The data collected to evaluate these research questions is introduced in Section 6.3.2.

6.3.1 *Research Questions*

Encoding of domain concepts in programming language constructs makes it hard to validate that a program behaves as intended. To this end the domain-specific language and modeling communities aim to eliminate the gap between domain concepts and language constructs. IceDust is such an attempt, thus the first research question is as follows:

> *RQ-Validatability: Are the language features provided by IceDust beneficial for establishing that an information system 'does the right thing'?*

Performance for information systems is important as the amount of information and the amount of users tends to grow over time, and the filtering and processing to create new data can depend on a lot of data. We capture this in question two:

> *RQ-Performance: Do IceDust-based information systems perform well with real-world data and workloads?*

Independent of how useful an approach is in terms of the first two research questions, it must not require significant additional effort. Hence, our last research question is:

> *RQ-Effort: How much effort is required for developing information systems with IceDust?*

6.3.2 *Data Collected*

Below we list the data collected to answer the research questions. As this is a real project, with real users, some data is not available (see discussion in Section 6.7.5).

RQ-Validatability We look at the source code of the derived value calculations in the vanilla system (without IceDust), and the system with IceDust. We qualitatively asses their impact on the amount of encoding. Moreover, we quantitively asses their impact on the amount of encoding by looking at lines of code, where we assume fewer lines of code means less accidental complexity. (WebDSL and IceDust feature similar syntax and both organize code in entities.) The business logic in IceDust, and in vanilla, we can make available as artifact.

RQ-Performance We measure the original and migrated system performance under a variety of simulated workloads with real-world data sets, and analyze the achieved performance. In addition we measure when and how performance degrades under increasing workloads. The raw performance numbers we can make available as artifact. The private user data we benchmarked on, we cannot make available.

RQ-Effort We measure and discuss the effort required for migrating WebLab to IceDust, distinguishing expressing business logic, embedding it in the rest of the application, performance engineering, and benchmarking. Moreover, we measure and discuss the effort spent on the IceDust compiler triggered by the WebLab case study.

6.4 CASE STUDY CONTEXT

In order to better contextualize our case study, we describe the context as proposed by Dyba et al [Dybå et al., 2012].

6.4.1 *WebLab*

WebLab is a learning management system in which students can submit assignments that get graded semi-automatically. Students can submit answers to programming, essay, and multiple choice questions. Individual programming submissions are graded automatically based on (hidden) unit tests. Multiple choice questions also are graded automatically, and all types of submissions can be graded on checklists by teaching assistants. Moreover, WebLab provides many features for calculating the final grades of students: weighted averaging, pass-n-out-of-m assignments, deadlines with late penalties, personal deadline overrides, personal grade overrides, bonus assignments, optional assignments, minimum grade to pass, and calculation traces to explain the final grades. Finally, these grades are used in all kinds of statistics about the assignments and courses in WebLab.

The primary success criterion for WebLab is whether the system is reliable enough to use for course labs and exams. This reliability has two aspects. First, its availability should be high. During exams, or labs with deadlines, WebLab should not succumb to the peak loads, as this would invalidate exams or labs. This is reflected in *RQ-Performance*. Second, the computed final grades of students should respect all features which interact with grade calculation,

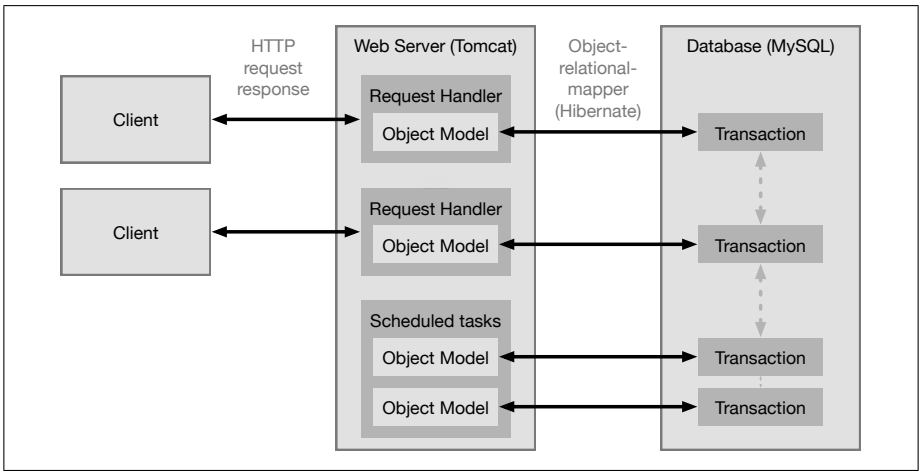


Figure 6.1 WebLab uses a standard stateless architecture for web servers. HTTP requests are serviced in isolation by a request handler. A request handler loads (saves) the data from (to) the database by means of an object-relational mapper. Request handlers interact concurrently with the database through transactions. The scheduled task executor handles periodic or asynchronous tasks.

otherwise the final grades are not reliable, and teachers will have to resort to spreadsheets again. WebLab has so many features interacting with grade calculation that this proved to be a non-trivial task the past few years while WebLab was evolving. This is reflected in *RQ-Validatability*.

6.4.2 *Software Architecture*

WebLab is a web-based information system that runs on a single shard server. WebLab uses a standard web architecture (Figure 6.1): a relational database for data persistence and transactions, a HTTP request handler which handles each request in isolation, and an object-relational mapper for loading and storing data every request. Its technology stack is the Tomcat web server, the Hibernate object-relational mapper [Neil, 2008], and MySQL. WebLab is built in the domain-specific language WebDSL [Groenewegen et al., 2008], which provides a typed integration between a Java-like object-oriented language, a SQL-like language (HQL), a custom UI templating language, AJAX interactions, and access control rules. While WebDSL is a domain-specific language, the code for grade and statistics calculation (which we migrate to IceDust) is written in the Java-like language of WebDSL. So for the purpose of this case study, we can regard this code as non-domain-specific.

6.4.3 *Server Setup*

WebLab runs on a large, but relatively old web server. It has 4 12-core 1.7 GHz AMD Opteron processors, 96 GB memory, and 4 500 GB conventional hard disks in RAID 10 configuration. In order to scale under a large parallel

workloads, the Tomcat server is configured to use up to 5 GB of memory. The disk bottleneck is mitigated by giving MySQL an InnoDB buffer pool size of 30 GB.

Our development machines, which we use both for development and for benchmarking before continuing to the acceptance stage, are mid-2014 MacBook Pro's. These feature 2.8 GHz Intel Core-i7 processors, 16 GB memory, and a fast PCI-e solid state drive. As memory is limited on these machines, both Tomcat and MySQL only get 2 GB of ram.

6.4.4 *Development Timeline*

Migration to IceDust started in September 2015, but stalled in October 2015 after 15 person days (PD) due to lack of expressiveness of IceDust. The migration restarted in September 2017. As of July 2018, WebLab-IceDust is in acceptance stage. Since the restart, 80 PDs have been spent. Full-time work was not feasible due to other pressing projects.

6.4.5 *Tools*

IceDust is available as a standalone Eclipse plugin. However, we used the language workbench Spoofax (also integrated in Eclipse) as IDE, as WebLab-IceDust used the nightly version of IceDust during development. WebDSL is also available as an Eclipse plugin.

6.4.6 *Organization and Team*

The developers working on WebLab are a team of two scientific programmers within a university. The projects built by these scientific programmers are funded by internal customers (within the university) or external scientific organizations. As such, decisions made about these projects are based on limited resources and potential sources of funding.

The team working on the WebLab migration was this team of scientific programmers plus the IceDust developer. These scientific programmers were not familiar with IceDust before, but were already familiar with Spoofax. As the scientific programmers are housed at the same floor as the IceDust developer, a lot of informal knowledge transfer happened 'at the coffee machine' [Donaldson and Siegel, 2001].

6.5 THE WEBLAB ICEDUST IMPLEMENTATION

This section provides an overview of WebLab-IceDust and illustrates its use of IceDust's features.

6.5.1 *Overall Structure and Migration*

The WebLab code is organized in components as detailed in Figure 6.2. In WebLab-IceDust we migrated the data model partially from WebDSL to Ice-

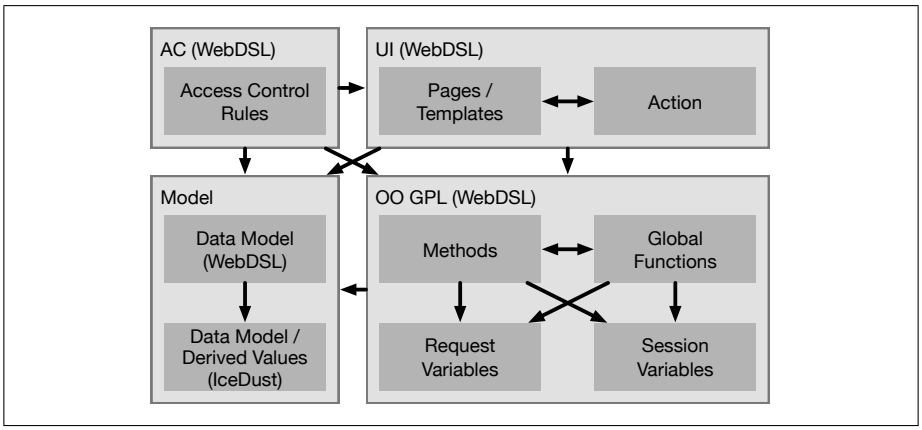


Figure 6.2 The WebLab code is organized in the following components. The base component is the data model which is partially defined in IceDust, and partially in WebDSL. The data model can be manipulated by the Java-like GPL base language of WebDSL. WebDSLs request variables are global per HTTP request, and its session variables are global per browser session. The user-interface is defined in two parts: actions which manipulate data by means of the GPL, and pages and templates which render information from the data model or GPL and define a navigation structure. Finally, the access control component provides or prevents access to pages and actions based on calls to the model or GPL.

Language	Files	LOC	Language	Files	LOC
IceDust	1	542	SQL (migration)	1	153
WebDSL	109	35696	CSS (mostly libs)	18	8513
Java	10	1688	JS (mostly libs)	1321	29055

Table 6.1 Number of files and lines of code in various languages in the WebLab-IceDust implementation.

Dust. Moreover, we migrated the calculation of derived values from object-oriented GPL code (methods and global functions) to IceDust derived value expressions. Finally, we refactored the rest of the code to use these derived values rather than the GPL methods.

6.5.2 *Size of the System*

The WebLab implementation consists of code written in multiple languages. Table 6.1 shows the size of the code; it is ca. 40,000 lines of code (LOC) when not counting external libraries. WebLab is a medium sized application with 61 different interactive pages using 549 UI-templates and displaying and modifying 98 different object-types through 4013 methods and functions.

Table 6.2 shows the number of instances of language concepts in IceDust. The business logic specified in IceDust is 542 LOC, where the hand-written in-

Concept	Count	Concept	Count
Attribute	*172*	*Entity*	*22*
abstract	3	base	14
user	27	sub type	8
derived	132	*Relation*	*23*
incremental	*2*	user	22
base	2	derived	1
eventual	*32*	*Function*	*17*
base	19		
overridden	13		
on-dem. eventual	*11*		
base	11		
inline	*87*		

Table 6.2 Number of instances of language concepts in the WebLab IceDust model.

```
progress        : Float = if(pass) 1.0 else 0.0
progressPercent : Float = round1(progress*100.0)
progressWeighted: Float = progress * weight
```

Figure 6.3 Business logic is expressed in IceDust through derived value expressions. These expressions in this example calculate the progress of students on assignments.

cremental calculation is over 800 LOC.[2] The generated WebDSL code from this IceDust code is over 20,000 LOC. This demonstrates that IceDust significantly cuts down on boilerplate for fine-grained incremental calculation strategies. In fact, it would be infeasible to write this by hand, let alone keep it correct while the model is evolving. Note that the hand-written solution provided only coarse-grained incrementality (checking a whole course for changes after a single change), while IceDust recomputes only derived values which are influenced by changes.

6.5.3 *Use of IceDust's Features*

Table 6.2 shows that WebLab makes use of many IceDust features, indicating their relevance for business logic in information systems. The rest of this subsection introduces these IceDust features and their use in WebLab in more detail.

Derived Values IceDust structures business logic into derived values. Derived values are calculated from base values or other derived values by means of derived value expressions. Figure 6.3 shows an example of derived value use in WebLab. These derived value expressions ensure that the definition of a derived value always is in one place, like a formula in a spreadsheet cell. This is good for traceability: the ability to verify why implemented business logic

[2] Line count obtained by manually listing the methods and fields which contribute to calculation of grades and statistics.

made certain decisions. When specifications are scattered, traceability tends to suffer [Walker and Viggers, 2004]. Moreover, these derived value expressions lend themselves well to incrementalization. WebLab-IceDust uses 133 derived value expressions, 132 for attributes and 1 for a bidirectional relation.

Incremental Computing Derived values can be computed incrementally by IceDust. Figure 6.4 shows an example of an incrementally computed derived value in WebLab. Computed values are read from cache, and when underlying values change only the cached values depending on these changes are recomputed. With these incremental derived values, information systems can provide fast reads. However, in the WebLab specification we only use two incremental derived values, as we mostly use eventual computing.

Eventual Computing Although incremental computing improves read performance, it makes writes to base values slower, as the writes to base values include recalculating all changed derived values. Eventual computing speeds up writes to base values by sacrificing consistency between base values and derived values. The updates to derived values are postponed, temporarily allowing reads to return outdated derived values (Figure 6.5). Many derived values in the WebLab specification are eventually calculated. With eventual calculation, WebLab can reliably service many concurrent users who interact with the same data.

On-demand Computing On-demand computing in IceDust means no caching at all. The on-demand calculation strategy is used when performance gains of caching do not outweigh the space cost. When an on-demand expression refers to an eventually calculated value, the on-demand value is also potentially outdated when read. We indicate this by calling this calculation strategy `on-demand eventual`. Figure 6.6 shows examples of on-demand computed derived values in WebLab.

Computing Strategy Composition In IceDust, the mentioned calculation strategies can be composed within a single specification (Figure 6.6). To safeguard against erroneous compositions, IceDust employs a static check (Figure 6.7). In WebLab-IceDust all derived value compositions are checked. These checks alert the developer when overlooking the impact of changing the calculation strategy of a derived value.

Inline Inline derived values enable breaking up a big expression into a set of smaller expressions, much like a let expression in functional programming languages (Figure 6.8). Toggling between inline and a calculation strategy controls the granularity of incremental computing. The majority of derived values in the WebLab specification are inline, favoring a somewhat larger granularity and smaller cache size.

Functions Functions enable reuse and abstraction in IceDust. Figure 6.9 shows examples of functions in the WebLab specification. The WebLab specification has 17 functions, 5 of these are reusable abstractions (such as the first function in Figure 6.9) while the rest is used to group together the markdown reporting scattered in the system.

```
deadline     : Datetime?
deadlineComp : Datetime? =
  deadline <+ parent.deadlineComp (incremental)
```

Figure 6.4 Derived value expressions can be calculated incrementally in IceDust. In this example snippet assignment deadlines are inherited from ancestors if not provided. When a `deadline` is set, IceDust automatically updates `deadlineComp` for that assignment on all its descendants. (The `<+` operator takes the left value if it is present, otherwise the right value.)

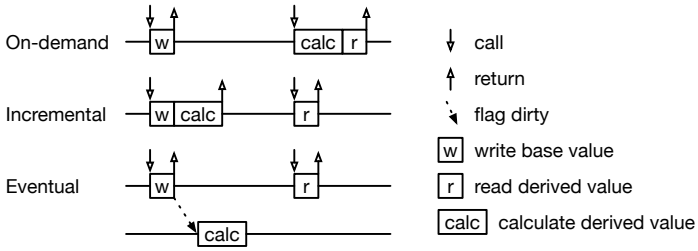


Figure 6.5 Thread activation diagrams for different calculation strategies in IceDust.

```
numAttempted  : Int = countTrue(subsInEval.attemptedComp)    (eventual)
numCompleted  : Int = countTrue(subsInEval.completedComp)    (eventual)
numPassed     : Int = countTrue(subsInEval.pass)             (eventual)
completedPerc : Int = numCompleted * 100 /. numAttempted <+ 0
                                                  (on-demand eventual)
passPerc      : Int = numPassed    * 100 /. numAttempted <+ 0
                                                  (on-demand eventual)
```

Figure 6.6 Derived values can be calculated eventually and on-demand in IceDust. On-demand values are not cached, they are recalculated on every read. In this example snippet the raw statistics of assignments are cached and eventually updated, while the percentages for presentation purposes are not cached.

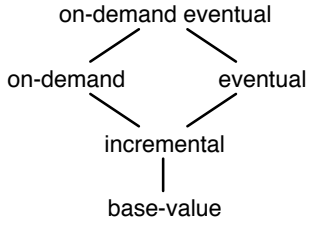


Figure 6.7 Calculation strategies guarantee their properties iff the derivation expressions refer to derived values with the same strategy or stronger strategies (lower in the lattice).

```
gradeWeighted: Float =
  if(weightCustom > 0.0) totalGrade / weightCustom <+ 0.0
  else totalGrade                                                    (inline)
gradeRounded : Float =
  max(gradeWeighted - (sub.penalty <+ 0.0) ++ 1.0).round1()          (inline)

gradeOnTime  : Float =
  if(sub.onTime <+ false) gradeRounded else 0.0                      (inline)

maxNotPassed : Float =
  max(0.0 ++ assignment.minimumToPass - 0.5).round1()                (inline)
passSub      : Boolean =
  sub.filter(:AssignmentCollectionSubmission).passSub <+ true        (inline)
maxNotPass   : Float =
  if(passSub) gradeOnTime else min(gradeOnTime++maxNotPassed)        (inline)

grade        : Float = min(maxNotPass ++ scheme.maxGrade)          (eventual)
```

Figure 6.8 Derived values in IceDust can be inlined on use site, this controls the granularity of incremental and eventual calculation. Here a submission grade is calculated based on various parameters, but only the final grade is cached.

```
hasF(d : Float*) : Boolean = count(d) > 0

gradeTracePassFail(s:BasicSubmission) : String =
  "Pass or Fail assignment, result: " +
  if(s.pass) "**PASS**" else "**FAIL**"
```

Figure 6.9 Functions in IceDust enable abstraction. Incremental and eventual computing inlines functions on use-sites.

```
entity CollectionSubmission extends Submission {
  progress : Float =
    if(pass)
      1.0
    else if(!isPassN)
      sum(subsForGrade.progressW) / totalWeight
    else
      sum(subsForGradeN.progressW)/ totalWeightN
    <+
      0.0
}
```

Figure 6.10 Inheritance with overriding enables modeling variation in IceDust. Progress in collection submissions takes the progress of the children (`subsForGrade`) into account. It overrides `progress` of `Submission` defined in Figure 6.3.

Inheritance and Overriding Inheritance and overriding enables the modeling of variation in IceDust. Figure 6.10 shows an example of this. The WebLab specification uses 13 derived value attribute overrides.

Native Multiplicities All derived value expressions make use of native multiplicities: the cardinality of values is part of the type system, and operators and functions are automatically lifted [Harkes et al., 2016]. The multiplicity

type system prevents null-pointer errors, and the automatic lifting prevents boiler-plate code dealing with collections and optional values.

6.5.4 *IceDust Feature Requests*

Apart from using the existing IceDust features, the WebLab implementation required features not previously supported in IceDust. For WebLab, two IceDust extensions have been developed; below we introduce these extensions and the specific rationales for developing them.

Multi-Threaded Eventual Calculation WebLab has high peak workloads concentrated on a small subset of all data: online exams with hundreds of students. Moreover, the derived values depend on many values and influence many other derived values. For example, the `deadline` in Figure 6.4 flows to all descendant assignments, and influences the grade calculation for all student submissions to these assignments. This can create a performance problem when students are submitting answers an exam while concurrently a teacher tries to change the deadline of that exam. Eventual calculation minimizes the number of transaction conflicts, such that these interactions can all succeed concurrently.

However, if availability is not important (for example recalculating all derived values in a course, after a migration), incremental computing is much faster than eventual computing. To mitigate this issue we made eventual calculation multi-threaded, and the number of threads configurable at runtime. This enables us to allocate more resources during migration such that it takes hours instead of days.

Manual override During development of WebLab-IceDust, a performance caveat was discovered. With some derived bidirectional relations, IceDust fails to capture the dependencies precisely with its path-analysis. This leads to a slowdown for the single derived relation in Table 6.2. The relational calculus captures the dependencies more precisely. This prompted a feature request for IceDust to ignore incremental updates for a specific derived value. Instead, we use the underlying relational database to manually incrementalize this relation.

6.6 ICEDUST EVALUATION

Many IceDust features are used to achieve a performant implementation with a validatable specification. However, achieving this also required work on the IceDust compiler. In this section we investigate these observations in more detail by evaluating the research questions introduced earlier.

6.6.1 *RQ-Validatability*

Migrating the business logic from WebDSL to IceDust reduced the line count by 32% (from 800+ to 542). As this is a real-world system, not a toy example

designed to showcase the DSL, we believe this result to be significant. It indicates that WebLab-IceDust contains less accidental complexity. We assess IceDust's effect on validatability qualitatively below.

Improved Traceability using Derived Value Attributes Derived value attributes have been used extensively, as illustrated by Table 6.2 and all code figures in Section 6.5. The derived value attributes make sure that derived values have one unique definition: the derivation expression. This helps traceability, developers never have to doubt whether a derived value in the system comes from a specific piece of code. This in turn simplifies reasoning about the code.

Figuring out where a derived value came from was complicated in WebLab-vanilla. When debugging, more time was spent making sure that the derived value did not originate from another piece of code, rather than trying to understand why a specific piece of code could have produced a specific value. Expressing the business logic in IceDust shifted the debugging conversation from tracing implementation details to domain discussions about the business logic.

Improved Readability with Native Multiplicities The WebLab-vanilla implementation in WebDSL suffered from the billion dollar mistake: null-pointers [Hoare, 2009]. The code-base is littered with non-`null` checks. Modern languages often adopt the `Option` Monad, with accompanying boiler-plate code containing `map`s and `flatMap`s or `do` notation. Native multiplicities solve the billion dollar mistake without introducing boiler-plate code. The business logic written in IceDust only mentions multiplicities when needed. This improves the readability of WebLab's business logic significantly.

Simpler Performance Engineering with Calculation Strategies In WebLab-vanilla it was very hard to validate that caching of derived values was correct. In fact, during migration we discovered inconsistencies in the data set from the live system. A grading parameter had been changed in a course, but the cached final student grades were never updated. With IceDust, cache invalidation is correct by construction. Moreover, multiple calculation strategies can only be soundly composed in IceDust. This means developers can stop worrying about the correctness of incremental computing and end users get correct out-of-date indicators for eventual calculation.

Another benefit of IceDust is that it is easy to see which calculation strategies are used. This makes it easier to understand and discuss performance trade-offs.

Separation of Concerns IceDust's design forces a separation of concerns between business logic and performance. Both can be edited separately in IceDust. Since we adopted IceDust, we noticed that business modeling and performance engineering have become two separate activities. The business logic specification is stable during performance engineering.

Remaining Intrinsic Complexity While IceDust reduces the accidental complexity of WebLab's business logic considerably, the business logic can still be complicated to understand. For example, Figure 6.8 takes some effort to

understand, as many variables contribute to the grade of a student (the full specification is even longer). Note that in WebLab-vanilla it was not even possible to see that the business logic is inherently complicated. In future work we might explore how to better organize the remaining intrinsic complexity. We summarize as follows regarding RQ-Validatability:

> *Derived value expressions, as a single source of computation, give developers confidence that they understand what the business logic specification means.*
>
> *During performance engineering developers can reason about what the system is going to do based on calculation strategies, without worrying about inconsistencies.*

6.6.2 *RQ-Performance*

To assess whether WebLab-IceDust performs well with real-world data sets and real-world workloads, we asked the main WebLab developer to describe all scenarios that could be performance bottlenecks. We identified three categories of interactions. Lightweight actions that hundreds of actors do concurrently. For example, students submitting new answers. Mediumweight actions have a larger effect and are performed by a single actor, while concurrently lightweight actions are performed. For example, a teacher postponing the deadline of the exam by 10 minutes, during the exam. And finally, heavyweight actions with a huge effect, performed by a single actor. For example, an administrator recalculating all derived values in a course after a migration.

For all these examples we used real-world data from the live database. (Which we cannot make available for privacy reasons.) Unfortunately, WebLab does not save all HTTP requests, so we could not replay real-world workloads. Fortunately, WebLab saves the history of programming submissions, so we could estimate the workload based on that.

We report on our final configuration of calculation strategies: mainly eventual computing. We experimented with other configurations, but none provided adequate availability under concurrent workloads. We vary the number of eventual computing threads to assess WebLab-IceDust's scalability. We report both the performance on a MacBook (our development machine) and the web server. Our baseline performance is the WebLab-vanilla implementation.

We identified three lightweight actions: random submission reads (browsing), random submission creations (first-time browsing), and random submission edits (working). Many students perform these actions concurrently. For these actions we are interested in the maximum workload WebLab can handle. Moreover, we also want to know under what workload derived value calculation starts to lag behind. If the workload is below that threshold, a teacher can see live statistics of exam progression during the exam. During a representative exam which lasted 3 hours and 30 minutes, we had 31836 code edits by 278 students. This is on average 3 edits per second. Peak load was 15 edits per second, and the 99th percentile is 8 edits per second.

We identified two mediumweight actions: change deadline and change checklist weight. The checklist is a grading tool for teaching assistants, and the checklist weight determines the ratio between other means of (automated) grading and the checklist. These two actions are performed by teachers, possibly while students are submitting answers. For these actions we are interested in how long it takes for all derived values depending on the change to be computed. Changing a deadline changes all deadlines lower in the assignment tree, but only influences grades if submissions are late. On the other hand, checklist weights are sure to influence the grades, but only of the assignment and its ancestors.

We identified only one heavyweight action: recalculate a course. This action is performed by administrators after a migration. For this action we are also interested in how long it takes to compute all derived values.

Results Table 6.3 shows the results of our benchmarks. IceDust enables live statistics during exams (create, edit, and read submissions), which was not possible with vanilla due to availability issues. Moreover, it can provide live statistics for well above 3 edits per second. IceDust speeds up the medium actions (change deadline and checklist weight) significantly, as IceDust's fine-grained incrementality does not have to visit the whole course. Also, with enough worker-threads, IceDust improves the recalculation speed of whole courses.

However, WebLab-IceDust can handle less peak load. With more IceDust worker-threads running, less processing power is available for request-handler threads. Moreover, object creation (in create submission) is more costly with IceDust. All relations in IceDust are bidirectional, opposed to many unidirectional relations in vanilla. Unfortunately, WebDSL and the ORM unnecessarily load objects in memory for keeping relations bidirectionally consistent, even when the other side of the relation is never used. We tried fixing this, but after 4 person days we concluded that it was not worth the effort. Thus, IceDust provides up-to-date statistics at the cost of slower object creation in this case study.

In terms of scalability, more parallelizable workloads benefit more from more worker threads. Recalculate course scales better than the mediumweight actions (change deadline and checklist weight). And the mediumweight actions performed on larger courses benefit more from extra threads than the same actions performed on smaller courses. Surprisingly, 2 threads for recalculating courses on the laptop consistently takes less than half the time of 1 thread. We cannot explain this, but we think it might be due to the CPU speed-step algorithm. Also, scalability on the laptop is hampered by throttling (up to 4ghz for single thread down to 2.6ghz for 4+ threads). The server does not have throttling, so it scales better. However, the server has slower CPUs in general, so it needs more threads to achieve live statistics. As the server has many more CPUs, its throughput (req/sec) is higher; but as the CPUs are slower the latency is also slightly higher (sec/req). In terms of performance, we summarize:

		Machine	MacBook									Server										
		Impl.	Vanilla		IceDust							Vanilla		IceDust								
		Threads	0	1	0	1	2	3	4	5	6	0	1	0	2	4	6	8	10	12	14	16
Action	**Unit**	**Course**																				
read	req/sec	Small	90.99	-	84.76	80.67	76.15	72.37	69.77	65.90	62.37	136.04	-	129.65	112.53	109.58	103.11	92.68	84.39	74.66	67.29	61.18
submission		Medium	89.68	-	90.85	84.65	80.86	77.49	73.83	71.00	68.01	152.95	-	145.04	125.37	119.33	112.02	104.82	93.88	88.26	80.14	71.66
		Large	95.96	-	94.56	89.42	84.60	80.70	77.04	74.01	70.64	132.45	-	119.62	141.99	130.21	115.44	105.22	97.49	96.07	95.98	96.19
edit	req/sec	Small	34.17	-	84.41	79.21	75.40	71.77	68.27	64.42	61.07	31.06	-	120.49	115.52	115.42	114.88	114.48	114.18	114.18	113.55	113.25
submission		Medium	63.65	-	75.53	64.24	61.87	59.40	57.03	54.36	52.24	111.91	-	96.13	90.92	87.08	82.02	69.74	64.73	62.30	63.18	58.96
		Large	60.28	-	75.94	71.57	68.37	65.11	62.28	59.46	57.24	104.17	-	107.84	103.64	96.30	89.56	80.08	75.68	71.30	66.63	62.63
create	req/sec	Small	75.72	-	55.72	50.04	44.71	40.25	37.25	34.47	32.28	100.72	-	67.70	64.39	61.95	60.58	57.31	54.49	51.91	48.94	46.85
submission		Medium	101.18	-	56.61	50.17	43.71	40.95	37.37	34.96	32.24	137.07	-	67.91	64.45	65.55	62.12	58.37	58.17	55.65	55.80	49.03
		Large	101.12	-	30.60	25.85	22.99	21.45	19.58	18.15	16.61	139.34	-	35.93	35.73	34.02	33.58	32.52	31.73	31.40	29.47	28.79
edit	req/sec	Small	-	-	-	22.52	34.46	40.07	42.32	44.16	45.16	-	-	-	117.45	114.38	113.85	113.31	112.85	112.41	111.61	113.68
submission		Medium	-	-	-	6.94	13.12	15.78	16.86	19.60	19.87	-	-	-	6.59	11.29	16.49	20.93	23.95	26.75	29.15	30.72
(live stats)		Large	-	-	-	7.94	15.01	17.88	19.62	20.95	22.26	-	-	-	6.12	12.47	16.62	19.69	20.86	21.68	25.3	24.66
create	req/sec	Small	-	-	-	9.74	16.25	18.95	20.34	21.27	21.48	-	-	-	9.02	15.73	20.56	22.86	24.92	26.85	27.83	29.30
submission		Medium	-	-	-	4.73	8.34	10.55	11.38	12.41	13.18	-	-	-	4.49	8.31	11.26	13.40	14.82	15.77	16.45	17.30
(live stats)		Large	-	-	-	3.95	6.86	8.20	8.68	8.99	9.20	-	-	-	3.60	6.26	8.29	9.54	10.54	11.81	12.35	12.68
change	sec/	Small	-	120	-	145	79	63	58	51	45	-	303	-	195	110	78	64	56	51	48	47
deadline	10 reqs	Medium	-	210	-	186	96	75	70	62	55	-	571	-	252	140	106	79	68	62	59	57
		Large	-	777	-	296	156	122	115	103	92	-	2324	-	408	232	172	143	128	120	117	102
change	sec/	Small	-	600	-	24	18	17	16	16	14	-	1220	-	21	15	14	12	12	11	11	12
checklist w.	100 reqs	Large	-	6430	-	106	64	54	50	46	43	-	19130	-	123	71	55	46	41	39	41	45
recalculate	sec/req	Small	-	21	-	125	57	44	34	40	30	-	49	-	147	76	53	42	36	32	30	28
course		Medium	-	530	-	1295	598	472	425	373	327	-	1718	-	1557	795	549	434	373	333	318	302
		Large	-	5508	-	10368	4786	3670	3244	2910	2653	-	18494	-	12624	6833	5078	4365	4049	3821	3797	3748

Table 6.3 Benchmark results. The first three benchmarks are maximum system throughput under concurrent student actions. We report the average requests per second over 30 second runs, higher is better. More IceDust threads decrease performance, as less processing power is available for requests. (Vanilla calculation cannot run concurrently with load, hence no measurements for one thread.) The next two benchmarks are the system throughput under which live statistics can be maintained by IceDust. Also these we run for 30 seconds. More IceDust threads increase performance, as derived values are calculated faster. The final three benchmarks are the medium- and heavyweight teacher and administrative actions. For these we measure time to completion in seconds, lower is better. More IceDust threads increase performance, as derived values are calculated faster. All benchmarks have been performed three times. We report the median. All measurements lie within $\pm 10\%$ of the median.

Development Task	Effort	% Total
Modeling	9 PD	11%
Integration / Migration	24 PD	30%
IceDust Compiler	11 PD	14%
Benchmarking / Performance Engineering	36 PD	45%

Table 6.4 WebLab migration to IceDust effort

The WebLab implementation in IceDust enables live statistics, which was infeasible manually.

WebLab-IceDust performs similar or better compared to WebLab-vanilla, except for object creation.

6.6.3 *RQ-Effort*

Migrating the WebLab business logic to IceDust took 80 PDs in total. Table 6.4 shows the different development tasks.

We spent 11% of the total effort (9 PDs) on reverse engineering WebLab's business logic and modeling it in IceDust.

We spent 30% (24 PDs) on integrating that business logic into the rest of the application. This included writing migration code to port the data from vanilla's calculation to IceDust's calculation, new user-interface (UI) elements to indicate calculation progress, and retro-fitting unit tests. Also new functionality was added during the integration: student grades finalization (after a course is over and grades are final). Modeling finalization in IceDust was a matter of minutes, creating UI elements took more effort.

We spent 14% of the total effort (11 PDs) on the IceDust compiler to add new features. Both the IceDust developer and the WebLab developers made changes to the IceDust compiler. (Remember that the WebLab developers are familiar with Spoofax and WebDSL.) The added language features enabled us to keep the separation of concerns between business logic and performance, effort well spent.

We spent 45% of the total effort (36 PDs) on benchmarks and performance engineering. As WebLab is used for exams at our university, it is of paramount importance that we can trust its performance. Designing benchmarks, setting up a benchmark infrastructure, and performing the benchmarks took the most time. While it is technically not part of the migration, it was required to give the responsible developers the confidence in WebLab-IceDust. A benefit of the calculation strategies is that is easy to switch between them. Proper benchmarking requires time, but getting a correctly functioning variant implementation to benchmark was a matter of minutes. Concerning RQ-Effort we conclude:

The effort for additional business logic is significantly lower in the ICL, but the total effort is not reduced.

While IceDust does not lead to an overall effort reduction or increase, it does increase separation of concerns.

6.7 DISCUSSION

The preceding sections show how IceDust affects validatability, performance, and effort of a real-life information system. We put our results in a broader perspective in this section.

6.7.1 *Internal Validity*

Internal validity concerns whether our results can be trusted.

Bias One factor that affects this question is the bias because of the involvement of the authors in this case study itself. The authors are the developers of IceDust and WebLab. To counter this bias, we focused on aspects that can be objectively measured (size, concept counts, performance, effort).

Team Expertise To clarify the potential impact of the team on the case study outcomes, we describe the team's background and expertise. The scientific developers both have 5+ years experience in developing information systems on the WebDSL technology stack. The IceDust developer had little experience with the WebDSL technology stack, but 5+ years experience with developing web applications with object-oriented languages and relational databases. When the project started, the scientific developers understood the business logic written in IceDust, but had little understanding of IceDust's calculation strategies. During the migration they gained understanding of these strategies by inquiring the IceDust developer and through experimentation.

Benchmark Internal Validity The benchmark results depend on full stack of technologies: MySQL, Java, Hibernate, WebDSL, and IceDust. Moreover, the results also depend on the hardware used and the settings for MySQL and the JVM. We verified that we actually measured the impact IceDust by benchmarking vanilla, and that we did not measure noise by benchmarking multiple times with a low standard deviation.

In this chapter, our benchmarks focus on external validity. As suggested by Vitek et al., we have benchmarks focusing on external and on internal validity [Vitek and Kalibera, 2012]. Benchmarks focusing on internal validity are described in previous work [Harkes et al., 2016].

6.7.2 *Conclusion Validity*

Our findings favor the using an ICL for separation of the business logic and performance concerns. Conclusion validity raises the question whether these findings can be explained.

Design of IceDust IceDust has been specifically designed to achieve the benefits reported in this case study. So the design rationale of IceDust forms the theoretical explanation of the case study outcomes. For an extensive description of this design rationale we refer to [Harkes and Visser, 2014; Harkes et al., 2016; Harkes and Visser, 2017].

Cognitive Dimensions of Notations The specification of business logic in IceDust improves over the specification in an object-oriented language according to the cognitive dimensions of notations, a set of established language evaluation criteria [Blackwell et al., 2001]. Four dimensions are specifically improved.

IceDust greatly reduces *Error-Proneness* with regard to incremental and eventual computing. Developers can rely on the IceDust runtime to keep derived values consistent with their defining expressions. IceDust also removes *Hidden Dependencies* in derived values, as all derived values have a single definition (unlike fields in object-oriented language which can be assigned to in all methods). IceDust greatly reduces the *Viscosity* of calculation strategies, changing a strategy is changing a keyword. IceDust also reduces the *Verbosity of Language* by adopting native multiplicities.

Experience vs. Notation A rival explanation of the success for validatability we measured might be that it is easier to understand the code as we spent considerable time at the white-board trying to reverse engineer the original code. The WebLab team is skeptical, a lot of human working memory is required to fully grasp all details of WebLab's business logic (even though it is only 500 LOC in IceDust). Every time a developer has worked on another project, and comes back to WebLab, this business logic needs to be rediscovered. This rediscovering is much easier in the IceDust specification.

6.7.3 *Construct Validity*

When describing our case study setup (Section 6.3), we explained how the three aspects studied (validatability, performance, and effort) relate to our overall goal of assessing the usefulness of ICLs for developing information systems. From a construct validity point of view, there are additional aspects (constructs) that we could have studied. Unfortunately, our migration did not yield any data on these constructs. However, we do think that our study is still useful, as these constructs are largely orthogonal to aspects we did study.

Interactive Development Information system development requires experimenting to design and understand some of the business logic specifications. Validatable specifications and extensive testing can reduce, but not avoid this need.

Unfortunately, WebDSL impairs interactive development due to long compilation times (over 3 minutes for WebLab). While WebDSL features incremental compilation, it only applies to non-invasive WebDSL features (such as UI components). The IceDust to WebDSL compilation takes a couple of seconds, and the extra generated WebDSL code lengthens the WebDSL compilation by a minute. This extra minute can be explained by the huge amount of fine-grained incrementality code generated. We do not believe IceDust itself inherently impairs interactive development, but properly incrementalizing the WebDSL and IceDust compilers is separate project.

Maintainability In *Little Languages: Little Maintenance?* [van Deursen and Klint, 1998] van Deursen and Klint conclude that a DSL designed for a well-chosen domain and implemented with adequate tools may drastically reduce the costs for building new applications as well as for maintaining existing ones. While we have no experience with long-term maintainability, we did make observations which confirm their conclusion.

During the migration we added new functionality: grade finalization. Modeling finalization of grades, and finalization statistics of courses in IceDust was easy. The intended behavior could be expressed concisely in IceDust.

Another part of the effort in software maintenance is re-understanding the existing code. As grade finalization interacts with grade calculation in general, it needed to be 'hooked in'. Due to the derived value expressions it was easy to see what part of the specification needed to be modified. IceDust's emphasis on validatability suggests that re-comprehension of the system is simplified.

Business Logic Evolution The business logic of information system evolves over time. When decision policies change, different decisions can be made by the business logic based on the same data. The question is how to deal with this evolution. WebLab implements an ad hoc check and does not override previous decisions or derived values. The migration to IceDust did not address this question. However, a validatable specification, which only talks about business logic, might be a stepping stone for addressing this question.

6.7.4 *External Validity*

Here we discuss whether our results can be generalized.

Beyond WebLab IceDust is best suited for information systems with complex business logic and a considerable amount of concurrent interaction. As for these situations a validatable specification together with good performance is important. So far, WebLab is the only information system which we modeled with IceDust, integrated into the rest of the application, and benchmarked with user data. We did model other systems with IceDust, but did not integrate or benchmark them. The findings in this chapter with regard to validatability apply to these other information systems as well.

Beyond the Team To be successful with IceDust, a team should have experience with building small to medium scale information systems with object-oriented languages and relational databases. IceDust provides separation of concerns between business logic specification and performance engineering, but the latter still requires expertise. If the IceDust calculation strategies provide enough performance, this experience should suffice. However, to modify or add strategies, the team in addition requires language engineering expertise. The IceDust compiler is not overly complicated, both the WebLab team and a master student were able to extend it independently. Note that they did use Spoofax before.

Beyond WebDSL IceDust probably generalizes beyond WebDSL, any object-oriented language with an object-relational mapper should do. IceDust provides an interface to the rest of the application with the getters and setters of fields and the constructors of objects. At this moment we have not implemented any other backends that persist their data. We have not explored targeting non object-oriented languages.

Beyond Spoofax The IceDust implementation in Spoofax is a close translation from its grammar, static semantics, and dynamic semantics [Harkes and Visser, 2014; Harkes et al., 2016; Harkes and Visser, 2017]. IceDust does not feature exotic constructs in its semantics. Thus, with considerable effort, IceDust should be implementable in any language workbench or general purpose programming language.

6.7.5 *Repeatability*

This case study reports on the development of a real-world information system. WebLab was not specifically set up as a case study. This has advantages and drawbacks. The advantages include a realistic system, realistic performance constraints, realistic data sets, and an experienced team of developers. The drawback is the unavailability of the source code and user data. (The business logic in IceDust and vanilla, as well as the raw performance numbers, we can make available as artifact.) In McGraths's terms [McGrath, 1995], this is a field study, it emphasizes realism over repeatability.

6.7.6 *Research Implications*

This chapter provides an in-depth case study of the use of IceDust to implement the business logic of a learning management information system, focusing on validatability, performance, and effort. To corroborate and challenge our findings, additional studies are needed, both for IceDust-based systems as well as for other incremental computing approaches.

Furthermore, as this study detailed the need for new language features during application development, we propose future research on co-development of DSLs and applications.

6.8 RELATED WORK

We are not aware of any other case studies of ICL use for information systems. Instead, we compare our work to incremental computing and DSL case studies. Also, we contrast IceDust to other ICLs which we might have used instead.

6.8.1 *Case Studies in Incremental Computing*

Chan et al. investigate the trade-off between query performance, incremental maintenance cost, and storage space for materializing views [Chan et al.,

1999]. They find that the optimal solution is to materialize some views, not all. We did not report on how we select calculation strategies and what we materialize (`inline` is not materialized). However, we have observed a similar trade-off between query time and incremental maintenance cost while experimenting with various configurations.

Another case study in databases [Hoppe and Gryz, 2007] investigates various performance optimizations for maximal event throughput. Changes grouped together in a larger commit increases performance. Our experience is similar. One big commit (with `incremental`) can recalculate a full course much faster than many small commits (with `eventual`). However, large commits introduce concurrency conflicts, hurting availability. They report the highest performance with the business logic on the clients instead of in the database (by means of triggers). In IceDust, we also run the business logic in the GPL.

Behrend and Schueller did a case study adopting a new materialized view update technique [Behrend and Schüller, 2014]. They observe that their technique improves performance only in specific conditions. Similarly, IceDust's incrementalization improves over vanilla in specific conditions: small updates are processed much faster, while object creation is not.

These studies only report performance, not validatability or effort. So we cannot compare our validatability and effort findings with other incremental computing case studies.

6.8.2 *Case Studies with DSLs*

Adopting a state machine DSL in a case study [Batory et al., 2002] led to findings similar to ours. They estimate a 10-fold effort reduction in modeling new scenarios in the DSL. In contrast to us, they do not report any required effort for improving their DSL compiler. They report decreased complexity by a code size reduction of 2-3x. Similarly, we also have a 1.5x reduction.

Adopting the Risla DSL for financial applications [van Deursen, 1997] led to similar findings. The effort for new products was reduced 5-fold. They mention extending the DSL is not easy, but do not quantify this in effort. Like us, they say *"it has become much easier to validate the correctness of the software"*. Unfortunately, they do not provide evidence for this claim.

Adoption of the Pheasant DSL [Barisic et al., 2011] was studied in a lab experiment setting. They report an effort reduction of roughly 1.5x. However, in this lab setting, all tasks were expressible with the DSL, and DSL compiler effort is excluded. They report fewer errors and higher confidence for inexperienced users, but no difference for experienced users. We find better validatability for experienced users. This might be explained by the much higher complexity of our multi-month migration, opposed to one hour lab experiment setting.

Ericsson adopted model-driven (similar to DSL-driven) software engineering [Staron, 2006]. They conclude that coding is always necessary, including coding the code-generators. Our case study agrees, we added features to the

IceDust compiler during migration as well. They report adoption needs to be broken up in phases. We did not have to. This could be explained by the fact that WebLab is a smaller system. One of their goals was to increase productivity, but they did not measure it.

Adoption of the mbeddr extensible language [Völter et al., 2015] also led to similar findings. mbeddr reduced complexity and improved readability while code size stayed the same. Similarly, IceDust increases validatability, but our code-size did reduce. 5% of their effort was spent on new language extensions, while we spent 14% on new IceDust features. This might be explained by being able to express everything in C when a DSL feature is lacking in mbeddr, while IceDust needs to cover everything. Like us, they report an effort reduction for adding additional functionality, but not for the total effort.

6.8.3 *ICLs for Information Systems*

Various ICLs target information systems or present one as running example. Here, we cover these ICLs. ICLs targeted at different domains, but which are similar to IceDust's mechanics are covered in previous work [Harkes et al., 2016; Harkes and Visser, 2017].

Object-Set Queries (OSQ) [Rothamel and Liu, 2008] brings relational incremental updates to an object-oriented setting. This might be a viable approach for incrementalizing the bidirectional relation which we had to hand-optimize in this case study. However, OSQ only supports set comprehensions, not complex expressions for calculating primitive values. Moreover, OSQ works in-memory, it is not clear whether it would work with an object-relational mapper and concurrent interaction.

IncOQ [Liu et al., 2016] improves over OSQ by tracking dependencies statically and incorporating demand. However, the same limitations apply: only set comprehensions and only in-memory.

Complex Object Queries [Nakamura, 2001] is a predecessor of these. However, this early approach incurs a considerable performance penalty by translating everything into sets and tuples.

MOVIE [Ali et al., 2003] also incrementalizes OQL queries, and it does persist its data. However, it does not support recursion, which this case study requires.

OR-SQL has also been incrementalized [Liu et al., 2003]. However, it only supports additions and removals (as most relational approaches) which significantly impair its efficiency when calculating complex primitive value expressions.

idIVM [Katsis et al., 2015] incorporates diffs based on ids in a relational database. idIVM persists its data, but it is used by submitting queries to a database rather than by an object-relational mapper. Moreover, idIVM does not support recursion.

LogiQL [Green, 2015] also supports incremental relational updates [Veldhuizen, 2013]. In contrast to many relational approaches its implementation supports recursive aggregation, be it behind a compiler flag. However,

also this database requires interaction through queries, rather than an object-relational mapper.

6.9 CONCLUSION

In this chapter we present a case study that evaluates the use of the ICL IceDust for specification of the business logic of an information system. We conclude that the migrated code has better validatability, similar or better performance, and that the effort involved in modeling decreases, but total effort does not. The ICL creates a separation of concerns between business logic specification and performance engineering. Performance engineering still takes considerable effort, including pull requests to the ICL compiler.

In future work we would like to investigate the phenomenon of co-development of DSLs with their applications. What factors influence whether a DSL is co-developed with its application? And is this co-development similar to co-development of frameworks or libraries with applications?

Another direction for future work is composing incrementalization techniques. IceDust's path-based incrementalization works well for complex expressions, while the relational calculus works well for bidirectional relations. Can these be combined in a unified incremental computing approach?

Conclusion

7

How far did we get in addressing the vision outlined at the beginning of this dissertation? In this final chapter, we revisit our original goals, summarize the main contributions presented in this dissertation, reflect on our methodology, and discuss future work.

7.1 INFORMATION SYSTEM ENGINEERING REVISITED

In the introduction of this dissertation, we described six concerns for information system engineering which are challenging to tackle simultaneously:

- *validatability:* how easy is it for information system developers to see whether the system does the right thing?
- *traceability:* can the origin of decisions made by the system be verified?
- *reliability:* can we trust the system to consistently make decisions and not lose our data?
- *performance:* can the system handle the load of its users?
- *availability:* can we trust that the system performs its functionality all (or most) of the time?
- *modifiability:* how easy is it to change the system specification when requirements change?

Our vision was to address these concerns for developing information systems all at the same time, as it would improve information system development and use tremendously. The domain-specific languages introduced in this dissertation address these six concerns. Every individual DSL feature introduced in this dissertation improves at least one of these concerns over the existing state-of-the-art.

In the introduction, we hypothesized that declarative specification of information system data models and business logic is feasible and useful. This dissertation shows it is feasible to specify an information system in an incremental computing DSL by describing the design of IceDust and detailing the use of IceDust in the WebLab information system. Moreover, our experience with the WebLab case study also shows the usefulness of specifying an information system in an incremental computing DSL as validatability, traceability, reliability, and modifiability were considerably improved while retaining similar performance and availability.

7.2 SUMMARY OF CONTRIBUTIONS

The first four core chapters in this thesis have contributions in terms of language features over state-of-the-art approaches, while Chapter 6 details the

usefulness of these features in practice. We summarize our contributions per chapter below:

- The Relations language (Chapter 2) is a data modeling language that features first-class relations, n-ary relations, native multiplicities, bidirectional relations, and concise navigation. These language features improve information system validatability and modifiability over object-oriented and relational approaches. Moreover, this language features good traceability by adopting attributes from attribute grammars.

- IceDust (Chapter 3) is a data modeling language for expressing derived attribute values with incremental and eventual computing based on path analysis. IceDust enables switching to a different calculation strategy (for example caching) without invasive code changes, and is expressive enough to support recursive aggregation. IceDust's language features improve information system modifiability without sacrificing performance and availability over object-oriented and relational approaches.

- IceDust2 (Chapter 4) is a generalization of IceDust with derived bidirectional relations with multiplicity bounds and support for statically checked composition of calculation strategies. IceDust2's language features improve information system validatability over relational approaches, and validatability, modifiability, and reliability over reactive programming approaches.

- PixieDust (Chapter 5) is an incremental computing user-interface language for browser-based applications. PixieDust's language features improve information system modifiability without sacrificing performance over existing reactive approaches.

- IceDustified WebLab (Chapter 6) is a case study demonstrating the usefulness of specifying an information system in an incremental computing DSL as validatability, traceability, reliability, and modifiability were considerably improved while retaining similar performance and availability.

With these contributions we improved information system engineering. We conclude that declarative specification of information system data models and business logic is feasible and useful.

7.3 REFLECTION ON METHODOLOGY

In order to achieve these results we used design oriented research which iterates four phases: analysis, design, evaluation, and diffusion. This iterative methodology worked well for us. In every iteration we analyzed information system development problems, we designed a new DSL (or DSL features) to

address these problems, we evaluated the new DSL by applying it in practice and subjecting it to scrutiny, and we diffused our knowledge through scholarly articles. The iterative approach enabled us to design, implement, evaluate, and publish new language features one by one. We were mildly successful in diffusing our knowledge: all our papers got accepted by our peer reviewers, but to our knowledge our vision of incremental computing DSLs has not yet been adopted or put into practice by others outside of our university.

We used a variety of specific methods to produce evidence supporting contributions, each method corresponding to the type of evidence produced (Table 1.1). Some of these methods worked out better than others for us. Language formalization through grammars and inference rules for static and dynamic semantics enabled us to get to the essence of DSLs and communicate the DSLs to other scientists. Language implementation in Spoofax [Kats and Visser, 2010] was at times smooth and at times unfeasible. This was mainly determined by whether the desired language semantics fitted in the meta languages provided by Spoofax. The informal arguments in standard logical constructs gave us (and our peer reviewers) a reasonable amount of confidence in the properties of our DSLs. We explored formalizing these arguments in proof checkers to increase the confidence (Appendix A), but for us the increased confidence did not outweigh the extra effort. The benchmarks and case studies took more effort than expected. The micro benchmarks and micro case studies gave us confidence that we were measuring improvements because of new DSL features. Due to the enormous effort involved, we did only one macro case study with corresponding benchmarks, showing that our approach works for a real-life information system. Doing more macro case studies remains future work.

7.4 FUTURE WORK

Although we made progress in simultaneously tackling validatability, traceability, reliability, performance, availability, and modifiability in information systems engineering, there is room for further improvement. The research in this dissertation raises further research questions in various directions regarding multiplicity-safety, data modeling, business logic specification, incremental computing and empirical evaluation.

Multiplicity Safety

As described in the postscript of Chapter 2, the multiplicity safety is only statically guaranteed for read operations. Update operations in IceDust are checked for multiplicity-safety at runtime. In Alloy [Jackson, 2006], multiplicity-safety can be checked by bounded model checking (e.g. before deployment of the application with actual data). This raises the question whether it would be possible to check multiplicity-safety at compile time for update operations.

In the postscript of Chapter 4 we described a performance caveat of expression-based derived bidirectional relations. This performance caveat can be avoided by using relational queries, but relational queries do not provide multiplicity bounds. This raises the question whether it is possible to automatically transform queries into expressions and vice versa to obtain both good performance and multiplicity bounds.

Data Modeling and Business Logic Specification

A recurring theme while developing information systems is migration of data between versions of the data model and versions of the business logic. While migration between data models has been studied in depth [Vermolen et al., 2011], it is interesting to know whether those migrations can be made multiplicity safe. An approach could be to specify both versions of the data model in different modules or name spaces, and write a migration correspondence between the two which can be checked for multiplicity-safetey.

A new theme for migration in information systems is evolving business logic. ICLs which support changes of the business logic at runtime will update all derived values immediately. However, derived values such as course grades (in Chapter 6) or invoices should not change after they are 'final', even though in the future business logic might change, such as a tax percentage on the invoice. An approach could be to keep all versions of the business logic in separate modules, and refer to these modules from the data. However, this would blow up the code size significantly.

Relational databases do not support recursive aggregation as termination cannot be guaranteed. IceDust supports recursive aggregation, but does not guarantee termination. However, with IceDust it should be possible to guarantee termination when the paths used in the recursion are not cyclic in the runtime data (for example with a tree). In AlanLight a static checker is implemented for recursion over hierarchical data [Kunst, 2018]. This raises the question whether such a static checker can work for graph-based data as well.

Incremental Calculation

IceDust provides multiple calculation strategies and sound composition between these strategies. However, all these strategies are based on the technique of analyzing paths. Some programs perform much better when another incrementalization technique is used, for example the bidirectional relations mentioned in the postscript of Chapter 4. This raises the question whether it would be possible to support multiple incrementalization techniques in a single language.

One challenge is that different techniques operate on different basic data structures. For example, materialized views for relational databases operate on relational tables, while our work operates on object graphs. In order to combine two techniques, they need to be retargeted to a common data structure, or data structures need to be efficiently transformable into each other. A candidate for addition to IceDust would be Demand-driven incremental object

queries [Liu et al., 2016], as these are relational materialized views retargeted to object graphs.

Empirical Evaluation

The last research direction is evaluating the impact of DSL usage on software engineering empirically. In this dissertation we have concluded that the use of an incremental computing DSL is useful for information system engineering in a single in-depth case study on a learning management information system developed and maintained by scientific programmers within our university. More research is needed to verify DSL usefulness in general. This empirical research would need to be performed in many different flavors to better understand the usefulness of DSLs: in-depth case studies versus many small programming tasks, experienced DSL developers and users versus inexperienced ones, DSL developers communicating with DSL users versus no communication, etc. This research would help to understand how exactly software engineering benefits from using DSLs.

Bibliography

(2017a). Mobx. https://web.archive.org/web/20171008145333/https://mobx.js.org/. Accessed: 2017-11-04. (Cited on pages 106 and 121.)

(2017b). React. https://web.archive.org/web/20171104234320/https://reactjs.org/. Accessed: 2017-11-04. (Cited on page 104.)

(2017c). Redux. http://web.archive.org/web/20171104000918/https://redux.js.org/. Accessed: 2017-11-04. (Cited on pages 106 and 121.)

(2018). Stack overflow developer survey 2018. https://insights.stackoverflow.com/survey/2018. Accessed: 2018-03-21. (Cited on pages 124 and 125.)

(2019). Javafx. https://openjfx.io/. Accessed: 2019-01-26. (Cited on page 19.)

Acar, U. A. (2005). *Self-adjusting computation*. PhD thesis, Princeton University. (Cited on page 125.)

Acar, U. A. (2009). Self-adjusting computation: (an overview). In Puebla, G. and Vidal, G., 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. (Cited on pages 66 and 99.)

Albano, A., Ghelli, G., and Orsini, R. (1991). A relationship mechanism for a strongly typed object-oriented database programming language. In Lohman, G. M., Sernadas, A., and Camps, R., editors, *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings*, pages 565–575. Morgan Kaufmann. (Cited on page 34.)

Ali, M. A., Fernandes, A. A. A., and Paton, N. W. (2003). Movie: An incremental maintenance system for materialized object views. *Data & Knowledge Engineering*, 47(2):131–166. (Cited on pages 65, 98, and 146.)

Apt, K. R., Blair, H. A., and Walker, A. (1986). *Towards a theory of declarative knowledge*. IBM Thomas J. Watson Research Division. (Cited on pages 50, 52, and 54.)

Balzer, S. (2011). *Rumer: a Programming Language and Modular Verification Technique Based on Relationships*. PhD thesis, ETH, Zürich. (Cited on pages 22, 33, 43, 63, and 97.)

Balzer, S., Gross, T. R., and Eugster, P. (2007). A relational model of object collaborations and its use in reasoning about relationships. In Ernst, E., editor, *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*, volume 4609 of *Lecture*

Notes in Computer Science, pages 323–346. Springer. (Cited on pages 15, 22, and 33.)

Barisic, A., Amaral, V., Goulão, M., and Barroca, B. (2011). Quality in use of domain-specific languages: a case study. In Anslow, C., Markstrum, S., and Murphy-Hill, E. R., editors, *Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools, PLATEAU 2011, Portland, OR, USA, October 24, 2011*, pages 65–72. ACM. (Cited on page 145.)

Barras, B., Boutin, S., Cornes, C., Courant, J., Filliatre, J.-C., Gimenez, E., Herbelin, H., Huet, G., Munoz, C., Murthy, C., et al. (1997). *The Coq proof assistant reference manual: Version 6.1*. PhD thesis, Inria. (Cited on page 12.)

Batory, D. S., Johnson, C., MacDonald, B., and von Heeder, D. (2002). Achieving extensibility through product-lines and domain-specific languages: a case study. *ACM Transactions on Software Engineering Methodology*, 11(2):191–214. (Cited on page 145.)

Becker, J., vom Brocke, J., Heddier, M., and Seidel, S. (2015). In search of information systems (grand) challenges - a community of inquirers perspective. *Business amp; Information Systems Engineering*, 57(6):377–390. (Cited on page 7.)

Behrend, A. and Schüller, G. (2014). A case study in optimizing continuous queries using the magic update technique. In Jensen, C. S., Lu, H., Pedersen, T. B., Thomsen, C., and Torp, K., editors, *Conference on Scientific and Statistical Database Management, SSDBM '14, Aalborg, Denmark, June 30 - July 02, 2014*, page 31. ACM. (Cited on page 145.)

Bernstein, P. A., Hadzilacos, V., and Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems*. Addison-Wesley. (Cited on page 5.)

Bharati, P. and Chaudhury, A. (2004). An empirical investigation of decision-making satisfaction in web-based decision support systems. *Decision Support Systems*, 37(2):187–197. (Cited on page 3.)

Bierman, G. M. and Wren, A. (2005). First-class relationships in an object-oriented language. In Black, A. P., 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. (Cited on pages 22, 34, 43, 63, and 97.)

Blackwell, A. F., Britton, C., Cox, A. L., Green, T. R. G., Gurr, C. A., Kadoda, G. F., Kutar, M., Loomes, M., Nehaniv, C. L., Petre, M., Roast, C., Roe, C., Wong, A., and Young, R. M. (2001). Cognitive dimensions of notations: Design tools for cognitive technology. In Beynon, M., Nehaniv, C. L., and Dautenhahn, K., editors, *Cognitive Technology: Instruments of Mind, 4th International Conference, CT 2001, Warwick, UK, August 6-9, 2001, Proceedings*, volume 2117 of *Lecture Notes in Computer Science*, pages 325–341. Springer. (Cited on page 142.)

Boehm, B. W. (1988). A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72. (Cited on page 6.)

Boral, H. and DeWitt, D. J. (1984). A methodology for database system performance evaluation. In Yormark, B., editor, *SIGMOD 84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984*, pages 176–185. ACM Press. (Cited on page 12.)

Buneman, P., Libkin, L., Suciu, D., Tannen, V., and Wong, L. (1994). Comprehension syntax. *SIGMOD Record*, 23(1):87–96. (Cited on page 32.)

Burckhardt, S. (2014). Principles of eventual consistency. *Foundations and Trends in Programming Languages*, 1(1-2):1–150. (Cited on page 67.)

Carlsson, M. (2002). Monads for incremental computing. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional Programming (ICFP 2002)*, pages 26–35. (Cited on page 125.)

Chan, G. K. Y., Li, Q., and Feng, L. (1999). Design and selection of materialized views in a data warehousing environment: A case study. In *DOLAP 99, ACM Second International Workshop on Data Warehousing and OLAP, November 6, 1999, Kansas City, Missouri, USA, Proceedings*, pages 42–47. ACM. (Cited on page 144.)

Chen, P. P. (1976). The entity-relationship model - toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36. (Cited on page 15.)

Czaplicki, E. and Chong, S. (2013). Asynchronous functional reactive programming for guis. In Boehm, H.-J. and Flanagan, C., editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 411–422. ACM. (Cited on pages 106 and 121.)

Davies, J., Welch, J., Cavarra, A., and Crichton, E. (2006). 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. (Cited on pages 78 and 98.)

Demaine, E. D., Mozes, S., Rossman, B., and Weimann, O. (2009). An optimal decomposition algorithm for tree edit distance. *ACM Transactions on Algorithms*, 6(1). (Cited on page 103.)

Demers, A. J., Reps, T. W., and Teitelbaum, T. (1981). Incremental evaluation for attribute grammars with application to syntax-directed editors. In *POPL*, pages 105–116. (Cited on pages 66 and 99.)

Demetrescu, C., Finocchi, I., and Ribichini, A. (2011). Reactive imperative programming with dataflow constraints. In Lopes, C. V. and Fisher, K.,

editors, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 407–426. ACM. (Cited on page 125.)

Donaldson, S. E. and Siegel, S. G. (2001). *Successful software development*. Prentice Hall Professional. (Cited on page 129.)

Dybå, T., Sjøberg, D. I. K., and Cruzes, D. S. (2012). What works for whom, where, when, and why?: on the role of context in empirical software engineering. In Runeson, P., Höst, M., Mendes, E., Andrews, A. A., and Harrison, R., editors, *2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '12, Lund, Sweden - September 19 - 20, 2012*, pages 19–28. ACM. (Cited on pages 124 and 127.)

Elliott, C. M. (2009). Push-pull functional reactive programming. In Weirich, S., editor, *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, Haskell 2009, Edinburgh, Scotland, UK, 3 September 2009*, pages 25–36. ACM. (Cited on pages 65, 70, and 98.)

Erdweg, S., van der Storm, T., Völter, M., Boersma, M., Bosman, R., Cook, W. R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, G., Molina, P. J., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V. A., Visser, E., van der Vlist, K., Wachsmuth, G., and van der Woning, J. (2013). The state of the art in language workbenches - conclusions from the language workbench challenge. In Erwig, M., Paige, R. F., and Wyk, E. V., editors, *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*, volume 8225 of *Lecture Notes in Computer Science*, pages 197–217. Springer. (Cited on page 125.)

Felleisen, M. (1990). On the expressive power of programming languages. In Jones, N. D., editor, *ESOP 90, 3rd European Symposium on Programming, Copenhagen, Denmark, May 15-18, 1990, Proceedings*, volume 432 of *Lecture Notes in Computer Science*, pages 134–151. Springer. (Cited on pages 2, 123, and 125.)

Fowler, M. (2005). Language workbenches: The killer-app for domain specific languages? (Cited on page 125.)

Fowler, M. (2010). *Domain-Specific Languages*. Addison Wesley. (Cited on page 7.)

Fowler, S., Denuzière, L., and Granicz, A. (2015). Reactive single-page applications with dynamic dataflow. In Pontelli, E. and Son, T. C., editors, *Practical Aspects of Declarative Languages - 17th International Symposium, PADL 2015, Portland, OR, USA, June 18-19, 2015. Proceedings*, volume 9131 of *Lecture Notes in Computer Science*, pages 58–73. Springer. (Cited on page 121.)

Giannakopoulos, T., Dougherty, D. J., Fisler, K., and Krishnamurthi, S. (2009). Towards an operational semantics for alloy. In Cavalcanti, A. and Dams,

D., editors, *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, volume 5850 of *Lecture Notes in Computer Science*, pages 483–498. Springer. (Cited on page 64.)

Glinz, M. (2005). Rethinking the notion of non-functional requirements. In *Proc. Third World Congress for Software Quality*, volume 2, pages 55–64. (Cited on page 61.)

Gluche, D., Grust, T., Mainberger, C., and Scholl, M. H. (1997). Incremental updates for materialized oql views. In Bry, F., Ramakrishnan, R., and Ramamohanarao, K., 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. (Cited on pages 65 and 98.)

Green, T. J. (2015). Logiql: A declarative language for enterprise applications. In Milo, T. and Calvanese, D., 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. (Cited on pages 65, 97, 125, and 146.)

Green, T. J., Huang, S. S., Loo, B. T., and Zhou, W. (2013). Datalog and recursive query processing. *Foundations and Trends in Databases*, 5(2):105–195. (Cited on pages 50, 52, 65, and 97.)

Groenewegen, D. M., Hemel, Z., Kats, L. C. L., and Visser, E. (2008). Webdsl: a domain-specific language for dynamic web applications. In Harris, G. E., editor, *Companion to the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-13, 2007, Nashville, TN, USA*, pages 779–780. ACM. (Cited on page 128.)

Gupta, A. and Mumick, I. S. (1995). Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18. (Cited on pages 4, 52, 65, 97, and 125.)

Gupta, A., Mumick, I. S., and Subrahmanian, V. S. (1993). Maintaining views incrementally. In Buneman, P. and Jajodia, S., 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. (Cited on pages 65, 69, and 97.)

Gupta, M. (2012). *Akka essentials*. Packt Publishing Ltd. (Cited on page 66.)

Hadzilacos, V. (1988). A theory of reliability in database systems. *Journal of the ACM*, 35(1):121–145. (Cited on page 3.)

Halpin, T. (2006). Object-role modeling (orm/niam). In *Handbook on architectures of information systems*, pages 81–103. Springer. (Cited on pages 15, 19, and 41.)

Hammer, M. A., Dunfield, J., Headley, K., Labich, N., Foster, J. S., Hicks, M. W., and Horn, D. V. (2015). Incremental computation with names. In Aldrich, J. and Eugster, P., editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 748–766. ACM. (Cited on pages 69, 87, and 99.)

Hammer, M. A., Khoo, Y. P., Hicks, M., and Foster, J. S. (2014). Adapton: composable, demand-driven incremental computation. In O'Boyle, M. F. P. and Pingali, K., editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 18. ACM. (Cited on pages 69, 87, 99, and 125.)

Harkes, D. (2014). Relations: a first class relationship and first class derivations programming language. In Binder, W., Ernst, E., Peternier, A., and Hirschfeld, R., editors, *13th International Conference on Modularity, MODULARITY '14, Lugano, Switzerland, April 22-26, 2014*, pages 9–10. ACM. (Cited on page 35.)

Harkes, D., Groenewegen, D. M., and Visser, E. (2016). Icedust: Incremental and eventual computation of derived values in persistent object graphs. In Krishnamurthi, S. and Lerner, B. S., 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. (Cited on pages 13, 69, 70, 84, 96, 97, 108, 110, 121, 125, 134, 141, 144, and 146.)

Harkes, D., van Chastelet, E., and Visser, E. (2018). Migrating business logic to an incremental computing dsl: A case study. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*. ACM. (Cited on page 13.)

Harkes, D. and Visser, E. (2014). Unifying and generalizing relations in role-based data modeling and navigation. In Combemale, B., Pearce, D. J., Barais, O., and Vinju, J. J., 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. (Cited on pages 13, 42, 43, 44, 64, 74, 96, 97, 141, and 144.)

Harkes, D. and Visser, E. (2017). Icedust 2: Derived bidirectional relations and calculation strategy composition. In Müller, P., editor, *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, volume 74 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. (Cited on pages 13, 108, 111, 114, 121, 125, 141, 144, and 146.)

Hemel, Z., Groenewegen, D. M., Kats, L. C. L., and Visser, E. (2011). Static consistency checking of web applications with WebDSL. *Journal of Symbolic Computation*, 46(2):150–182. (Cited on page 23.)

Hemel, Z. and Visser, E. (2011). Declaratively programming the mobile web with Mobl. In Lopes, C. V. and Fisher, K., editors, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 695–712. ACM. (Cited on page 121.)

Hermans, F., Pinzger, M., and van Deursen, A. (2010). Automatically extracting class diagrams from spreadsheets. In Hondt, T. D., editor, *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, volume 6183 of *Lecture Notes in Computer Science*, pages 52–75. Springer. (Cited on page 65.)

Hoare, T. (2009). Null references: The billion dollar mistake. *Presentation at QCon London*, 298. (Cited on page 136.)

Hoppe, A. and Gryz, J. (2007). Stream processing in a relational database: a case study. In *Eleventh International Database Engineering and Applications Symposium (IDEAS 2007), September 6-8, 2007, Banff, Alberta, Canada*, pages 216–224. IEEE Computer Society. (Cited on page 145.)

Huang, S. S. and Smaragdakis, Y. (2008). Expressive and safe static reflection with morphj. In Gupta, R. and Amarasinghe, S. P., editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 79–89. ACM. (Cited on page 51.)

Humble, J. and Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education. (Cited on page 6.)

Jackson, D. (2002). Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering Methodology*, 11(2):256–290. (Cited on pages 64 and 98.)

Jackson, D. (2006). *Software Abstractions - Logic, Language, and Analysis*. MIT Press. (Cited on pages 37, 125, and 151.)

Jacobson, I., Booch, G., and Rumbaugh, J. E. (1999). *The unified software development process - the complete guide to the unified process from the original designers*. Addison-Wesley object technology series. Addison-Wesley. (Cited on page 15.)

Jouault, F. (2005). Loosely coupled traceability for atl. In *Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability, Nuremberg, Germany*, volume 91, page 2. (Cited on page 3.)

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. (1990). Feature-oriented domain analysis (foda) feasibility study. Technical report, DTIC Document. (Cited on page 70.)

Kats, L. C. L., Vermaas, R., and Visser, E. (2011). Integrated language definition testing: enabling test-driven language development. In Lopes, C. V. and Fisher, K., editors, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 139–154. ACM. (Cited on page 12.)

Kats, L. C. L. and Visser, E. (2010). The Spoofax language workbench: rules for declarative specification of languages and IDEs. In Cook, W. R., Clarke, S., and Rinard, M. C., editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 444–463, Reno/Tahoe, Nevada. ACM. (Cited on pages 11, 25, 125, and 151.)

Katsis, Y., Ong, K. W., Papakonstantinou, Y., and Zhao, K. K. (2015). Utilizing ids to accelerate incremental view maintenance. In Sellis, T., Davidson, S. B., and Ives, Z. G., editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1985–2000. ACM. (Cited on page 146.)

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In Aksit, M. and Matsuoka, S., 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. (Cited on page 93.)

Koch, C., Ahmad, Y., Kennedy, O., Nikolic, M., Nötzli, A., Lupei, D., and Shaikhha, A. (2014). Dbtoaster: higher-order delta processing for dynamic, frequently fresh views. *VLDB J.*, 23(2):253–278. (Cited on page 125.)

Konat, G., Kats, L. C. L., Wachsmuth, G., and Visser, E. (2012). Declarative name binding and scope rules. In Czarnecki, K. and Hedin, G., editors, *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*, volume 7745 of *Lecture Notes in Computer Science*, pages 311–331. Springer. (Cited on page 12.)

Krishnamurthi, S., Fisler, K., Dougherty, D. J., and Yoo, D. (2008). Alchemy: transmuting base alloy specifications into implementations. In Harrold, M. J. and Murphy, G. C., 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. (Cited on page 98.)

Kunst, G. (2018). Alanlight: sound, functionally correct, bounded acyclic data flow modeling. (Cited on page 152.)

Lerner, B. S., Elberty, L., Li, J., and Krishnamurthi, S. (2013). Combining form and function: Static types for jquery programs. In Castagna, G., 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 79–103. Springer. (Cited on page 34.)

Liu, J., Vincent, M. W., and Mohania, M. K. (2003). Maintaining views in object-relational databases. *Knowl. Inf. Syst.*, 5(1):50–82. (Cited on page 146.)

Liu, Y. A., Brandvein, J., Stoller, S. D., and Lin, B. (2015). Demand-driven incremental object queries. *arXiv preprint arXiv:1511.04583*. (Cited on page 66.)

Liu, Y. A., Brandvein, J., Stoller, S. D., and Lin, B. (2016). Demand-driven incremental object queries. In Cheney, J. and Vidal, G., 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. (Cited on pages 99, 146, and 153.)

Liu, Y. D. and Smith, S. F. (2005). Interaction-based programming with classages. In Johnson, R. E. and Gabriel, R. P., editors, *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 191–209. ACM. (Cited on page 33.)

Maier, I. and Odersky, M. (2013). Higher-order reactive programming with incremental lists. In Castagna, G., 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. (Cited on pages 65, 70, 87, and 98.)

McGrath, J. E. (1995). Methodology matters: Doing research in the behavioral and social sciences. In *Readings in Human–Computer Interaction*, pages 152–169. Elsevier. (Cited on page 144.)

McSherry, F., Murray, D. G., Isaacs, R., and Isard, M. (2013). Differential dataflow. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org. (Cited on page 125.)

Meijer, E. (2010). Reactive extensions (rx): curing your asynchronous programming blues. In *ACM SIGPLAN Commercial Users of Functional Programming*, page 11. ACM. (Cited on pages 66, 70, and 99.)

Meijer, E., Beckman, B., and Bierman, G. M. (2006). Linq: reconciling object, relations and xml in the .net framework. In Chaudhuri, S., Hristidis, V., and Polyzotis, N., editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, page 706. ACM. (Cited on page 34.)

Meyerovich, L. A., Guha, A., Baskin, J. P., Cooper, G. H., Greenberg, M., Bromfield, A., and Krishnamurthi, S. (2009). Flapjax: a programming language for ajax applications. In Arora, S. and Leavens, G. T., editors, *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009*, pages 1–20. ACM. (Cited on page 121.)

Mitschke, R., Erdweg, S., Köhler, M., Mezini, M., and Salvaneschi, G. (2014). i3ql: language-integrated live data views. In Black, A. P. and Millstein, T. D., 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. (Cited on pages 65, 69, and 98.)

Mosses, P. D. and New, M. J. (2009). Implicit propagation in structural operational semantics. *Electronic Notes in Theoretical Computer Science*, 229(4):49–66. (Cited on pages 11 and 30.)

Mumick, I. S., Pirahesh, H., and Ramakrishnan, R. (1990). The magic of duplicates and aggregates. In McLeod, D., Sacks-Davis, R., and Schek, H.-J., editors, *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 264–277. Morgan Kaufmann. (Cited on pages 54 and 65.)

Nakamura, H. (2001). Incremental computation of complex objects queries. In *OOPSLA*, pages 156–165. (Cited on page 146.)

Neil, E. J. O. (2008). Object/relational mapping 2008: hibernate and the entity data model (edm). In Wang, J. T.-L., editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 1351–1356. ACM. (Cited on page 128.)

Nelson, S., Noble, J., and Pearce, D. J. (2008). Implementing first-class relationships in java. *Proceedings of RAOOL*, 8. (Cited on pages 22 and 34.)

Nilsson, H., Courtney, A., and Peterson, J. (2002). Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*. ACM. (Cited on page 52.)

Noble, J. (1997). Basic relationship patterns. *Pattern Languages of Program Design*, 4. (Cited on page 19.)

Oskarsson, Ö. (1982). *Mechanisms of modifiability in large software systems*. PhD thesis, VTT Grafiska. (Cited on page 6.)

Pearce, D. J. and Noble, J. (2006). Relationship aspects. In Filman, R. E., editor, *Proceedings of the 5th International Conference on Aspect-Oriented Software Development, AOSD 2006, Bonn, Germany, March 20-24, 2006*, pages 75–86. ACM. (Cited on page 34.)

Pierce, B. C. (2002). *Types and Programming Languages*. MIT Press, Cambridge, Massachusetts. (Cited on page 11.)

Porter, B., Grieves, M., Filho, R. V. R., and Leslie, D. (2016). Rex: A development platform and online learning approach for runtime emergent software systems. In Keeton, K. and Roscoe, T., editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 333–348. USENIX Association. (Cited on page 100.)

Ramakrishnan, R., Ross, K. A., Srivastava, D., and Sudarshan, S. (1994). Efficient incremental evaluation of queries with aggregation. In *Workshop on Design and Impl. of Parallel Logic Programming Systems*, pages 204–218. (Cited on pages 65 and 97.)

Reynders, B., Devriese, D., and Piessens, F. (2017). Experience report: Functional reactive programming and the dom. In Sartor, J. B., D'Hondt, T., and Meuter, W. D., editors, *Companion to the first International Conference on the Art, Science and Engineering of Programming, Programming 2017, Brussels, Belgium, April 3-6, 2017*. ACM. (Cited on page 121.)

Ross, K. A. and Sagiv, Y. (1992). 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. (Cited on pages 65 and 97.)

Rothamel, T. and Liu, Y. A. (2008). Generating incremental implementations of object-set queries. In Smaragdakis, Y. and Siek, J. G., editors, *Generative Programming and Component Engineering, 7th International Conference, GPCE 2008, Nashville, TN, USA, October 19-23, 2008, Proceedings*, pages 55–66. ACM. (Cited on page 146.)

Rumbaugh, J. E. (1987). Relations as semantic constructs in an object-oriented language. In *OOPSLA*, pages 466–481. (Cited on page 34.)

Runeson, P., Höst, M., Rainer, A., and Regnell, B. (2012). *Case Study Research in Software Engineering - Guidelines and Examples*. Wiley. (Cited on page 124.)

Salvaneschi, G., Hintz, G., and Mezini, M. (2014). Rescala: bridging between object-oriented and functional style in reactive applications. In Binder, W., Ernst, E., Peternier, A., and Hirschfeld, R., editors, *13th International Conference on Modularity, MODULARITY '14, Lugano, Switzerland, April 22-26, 2014*, pages 25–36. ACM. (Cited on pages 4, 65, 70, 76, 87, 98, and 122.)

Schuster, C. and Flanagan, C. (2016). Reactive programming with reactive variables. In Fuentes, L., Batory, D. S., and Czarnecki, K., editors, *Companion Proceedings of the 15th International Conference on Modularity, Málaga, Spain, March 14 - 18, 2016*, pages 29–33. ACM. (Cited on page 121.)

Shaw, M. (2003). Writing good software engineering research papers. In *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA*, pages 726–737. IEEE Computer Society. (Cited on pages 10, 11, and 12.)

Siegmund, J., Siegmund, N., and Apel, S. (2015). Views on internal and external validity in empirical software engineering. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 9–19. IEEE. (Cited on page 12.)

Slepak, J., Shivers, O., and Manolios, P. (2014). An array-oriented language with static rank polymorphism. In Shao, Z., editor, *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, pages 27–46. Springer. (Cited on page 35.)

Staron, M. (2006). Adopting model driven software development in industry - a case study at two companies. In Nierstrasz, O., Whittle, J., Harel, D., and Reggio, G., editors, *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings*, volume 4199 of *Lecture Notes in Computer Science*, pages 57–72. Springer. (Cited on page 145.)

Steimann, F. (2013). Content over container: object-oriented programming with multiplicities. In Hosking, A. L., Eugster, P. T., and Hirschfeld, R., 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. (Cited on pages 16, 34, 43, 74, and 96.)

Steimann, F. (2015). None, one, many - what's the difference, anyhow? In Ball, T., Bodík, R., Krishnamurthi, S., Lerner, B. S., and Morrisett, G., editors, *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA*, volume 32 of *LIPIcs*, pages 294–308. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. (Cited on page 43.)

Szabó, T., Erdweg, S., and Völter, M. (2016). Inca: a dsl for the definition of incremental program analyses. In Lo, D., Apel, S., and Khurshid, S., editors, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 320–331. ACM. (Cited on page 125.)

Szárnyas, G., Izsó, B., Ráth, I., Harmath, D., Bergmann, G., and Varró, D. (2014). Incquery-d: A distributed incremental model query framework in the cloud. In Dingel, J., Schulte, W., Ramos, I., Abrahão, S., and Insfrán, E., editors, *Model-Driven Engineering Languages and Systems - 17th International Conference, MODELS 2014, Valencia, Spain, September 28 - October 3, 2014. Proceedings*, volume 8767 of *Lecture Notes in Computer Science*, pages 653–669. Springer. (Cited on page 66.)

Söderberg, E. and Hedin, G. (2012). Incremental evaluation of reference attribute grammars using dynamic dependency tracking. Technical Report 98, Department of Computer Science, Lund University. (Cited on pages 66 and 99.)

Tarjan, R. E. (1972). Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160. (Cited on page 49.)

ten Veen, N., Harkes, D. C., and Visser, E. (2018). Pixiedust: Declarative incremental user interface rendering through static dependency tracking. In Champin, P.-A., Gandon, F. L., Lalmas, M., and Ipeirotis, P. G., editors, *Companion of the The Web Conference 2018 on The Web Conference 2018, WWW 2018, Lyon , France, April 23-27, 2018*, pages 721–729. ACM. (Cited on page 13.)

Ujhelyi, Z., Bergmann, G., Ábel Hegedüs, Ákos Horváth, Izsó, B., Ráth, I., Szatmári, Z., and Varró, D. (2015). Emf-incquery: An integrated development environment for live model queries. *Science of Computer Programming*, 98:80–99. (Cited on pages 69 and 98.)

van Antwerpen, H., Néron, P., Tolmach, A. P., Visser, E., and Wachsmuth, G. (2016). A constraint language for static semantic analysis based on scope graphs. In Erwig, M. and Rompf, T., editors, *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 49–60. ACM. (Cited on page 12.)

van Deursen, A. (1997). Domain-specific languages versus object-oriented frameworks: A financial engineering case study. In *Proceedings Smalltalk and Java in Industry and Academia, STJA'97*. Ilmenau Technical University. (Cited on page 145.)

van Deursen, A. and Klint, P. (1998). Little languages: little maintenance? *Journal of Software Maintenance*, 10(2):75–92. (Cited on page 143.)

Veldhuizen, T. L. (2013). Incremental maintenance for leapfrog triejoin. *arXiv preprint arXiv:1303.5313*. (Cited on page 146.)

Vermolen, S., Wachsmuth, G., and Visser, E. (2011). Generating database migrations for evolving web applications. In Denney, E. and Schultz, U. P., editors, *Generative Programming And Component Engineering, Proceedings of the 10th International Conference on Generative Programming and Component Engineering, GPCE 2011, Portland, Oregon, USA, October 22-24, 2011*, pages 83–92. ACM. (Cited on page 152.)

Visser, E. (1997). *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam. (Cited on page 12.)

Visser, E. (2002). Meta-programming with concrete object syntax. In Batory, D. S., Consel, C., and Taha, W., editors, *Generative Programming and Component Engineering, ACM SIGPLAN/SIGSOFT Conference, GPCE 2002, Pittsburgh, PA, USA, October 6-8, 2002, Proceedings*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315. Springer. (Cited on page 12.)

Visser, E. (2003). Program transformation with Stratego/XT: Rules, strategies, tools, and systems in Stratego/XT 0.9. In Lengauer, C., Batory, D. S., Consel, C., and Odersky, M., editors, *Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers*, volume

3016 of *Lecture Notes in Computer Science*, pages 216–238. Springer. (Cited on page 12.)

Visser, E. (2007). WebDSL: A case study in domain-specific language engineering. In Lämmel, R., Visser, J., and Saraiva, J., 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. Springer. (Cited on pages 19, 23, 34, 43, 50, and 90.)

Visser, E. (2015). Understanding software through linguistic abstraction. *Science of Computer Programming*, 97:11–16. (Cited on pages 1 and 125.)

Vitek, J. and Kalibera, T. (2012). R3: Repeatability, reproducibility and rigor. *ACM SIGPLAN Notices*, 47(4a):30–36. (Cited on pages 12 and 141.)

Vollebregt, T., Kats, L. C. L., and Visser, E. (2012). Declarative specification of template-based textual editors. In Sloane, A. and Andova, S., editors, *International Workshop on Language Descriptions, Tools, and Applications, LDTA '12, Tallinn, Estonia, March 31 - April 1, 2012*, pages 1–7. ACM. (Cited on page 12.)

Völter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L. C. L., Visser, E., and Wachsmuth, G. (2013). *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org. (Cited on page 125.)

Völter, M., van Deursen, A., Kolb, B., and Eberle, S. (2015). Using c language extensions for developing embedded software: a case study. In Aldrich, J. and Eugster, P., editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 655–674. ACM. (Cited on pages 124 and 146.)

Völter, M. and Visser, E. (2011). Product line engineering using domain-specific languages. In de Almeida, E. S., Kishi, T., Schwanninger, C., John, I., and Schmid, K., editors, *Software Product Lines - 15th International Conference, SPLC 2011, Munich, Germany, August 22-26, 2011*, pages 70–79. IEEE. (Cited on page 100.)

Walker, R. J. and Viggers, K. (2004). Implementing protocols via declarative event patterns. In Taylor, R. N. and Dwyer, M. B., editors, *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2004, Newport Beach, CA, USA, October 31 - November 6, 2004*, pages 159–169. ACM. (Cited on pages 2 and 132.)

Wiedermann, B. and Cook, W. R. (2007). Extracting queries by static analysis of transparent persistence. In Hofmann, M. and Felleisen, M., editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 199–210. ACM. (Cited on pages 64 and 66.)

Willis, D., Pearce, D. J., and Noble, J. (2006). Efficient object querying for java. In Thomas, D., editor, *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings*, volume 4067 of *Lecture Notes in Computer Science*, pages 28–49. Springer. (Cited on pages 34 and 64.)

Willis, D., Pearce, D. J., and Noble, J. (2008). Caching and incrementalisation in the java query language. In Harris, G. E., 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. (Cited on pages 66 and 99.)

Wren, A. (2007). Relationships for object-oriented programming languages. *University of Cambridge, Computer Laboratory, Technical Report*, 702(UCAM-CL-TR-702). (Cited on pages 22, 23, and 34.)

Xi, H. and Pfenning, F. (1999). Dependent types in practical programming. In *POPL*, pages 214–227. (Cited on page 7.)

Yeo, K. T. (2002). Critical failure factors in information system projects. *International journal of project management*, 20(3):241–246. (Cited on page 125.)

Yin, R. K. (2013). Validity and generalization in future case study evaluations. *Evaluation*, 19(3):321–332. (Cited on page 124.)

Zeng, K., Agarwal, S., and Stoica, I. (2016). iolap: Managing uncertainty for efficient incremental olap. In Özcan, F., Koutrika, G., and Madden, S., editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1347–1361. ACM. (Cited on page 125.)

Zhao, W., Rusu, F., Dong, B., Wu, K., and Nugent, P. (2017). Incremental view maintenance over array data. In Salihoglu, S., Zhou, W., Chirkova, R., 0001, J. Y., and Suciu, D., editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 139–154. ACM. (Cited on page 125.)

Österle, H., Becker, J., Frank, U., Hess, T., Karagiannis, D., Krcmar, H., Loos, P., Mertens, P., Oberweis, A., and Sinz, E. J. (2011). Memorandum on design-oriented information systems research. *EJIS*, 20(1):7–10. (Cited on pages 1 and 10.)

Appendix: IceProof

A

In Chapter 2 we introduced the relations language with native multiplicities. We described the syntax, type system, multiplicity system, and dynamic semantics. Such a language description raises the question whether the type and multiplicity system is sound. In this appendix we prove type-preservation, termination, and multiplicity-preservation for a subset of the relations language. The proofs are mechanized in Coq and are available online.[1]

A.1 LANGUAGE SPECIFICATION

The subset of the language we consider in this appendix is expressions without an environment or store. We include the expressions added in Chapters 3 and 4 which do not require an environment or store. We call this sub language IceProof. Figure A.1 shows the grammar of IceProof. IceProof features just two types: integers and booleans (Figure A.2). IceProof tracks the multiplicity of expressions and their values (native multiplicities). It features four different multiplicities: exactly one, zero or one, one or more, and zero or

[1] https://github.com/MetaBorgCube/IceDust/blob/develop/icedust.proofs/multiplicities.v

$$
\begin{aligned}
e \in Expr ::=\ & nat \mid \texttt{true} \mid \texttt{false} \mid \texttt{null:Int} \mid \texttt{null:Boolean} \\
& \mid f\ (\ e\ ) \mid e_1 \oplus e_2 \mid e_1\ ?\ e_2 : e_3 \\
f \in UnOp ::=\ & \texttt{!} \mid \texttt{min} \mid \texttt{max} \mid \texttt{avg} \mid \texttt{sum} \mid \texttt{count} \mid \texttt{conj} \mid \texttt{disj} \mid \texttt{first} \\
\oplus \in BinOp ::=\ & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{\&\&} \mid \texttt{||} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{==} \mid \texttt{=!} \mid \texttt{<+} \mid \texttt{++} \\
& \mid \texttt{elemAt} \mid \texttt{indexOf}
\end{aligned}
$$

Figure A.1 The grammar of IceProof

$$
t \in Type ::= \texttt{Int} \mid \texttt{Boolean}
$$

Figure A.2 Types in IceProof

$$
m \in Multiplicity ::= \texttt{[1,1]} \mid \texttt{[0,1]} \mid \texttt{[1,n)} \mid \texttt{[0,n)}
$$

Figure A.3 Multiplicities in IceProof

$$
\begin{aligned}
v \in Value ::=\ & IntValue \mid BooleanValue \\
IntValue ::=\ & nat^* \\
BooleanValue ::=\ & Boolean^* \\
Boolean ::=\ & \texttt{true} \mid \texttt{false}
\end{aligned}
$$

Figure A.4 Values in IceProof

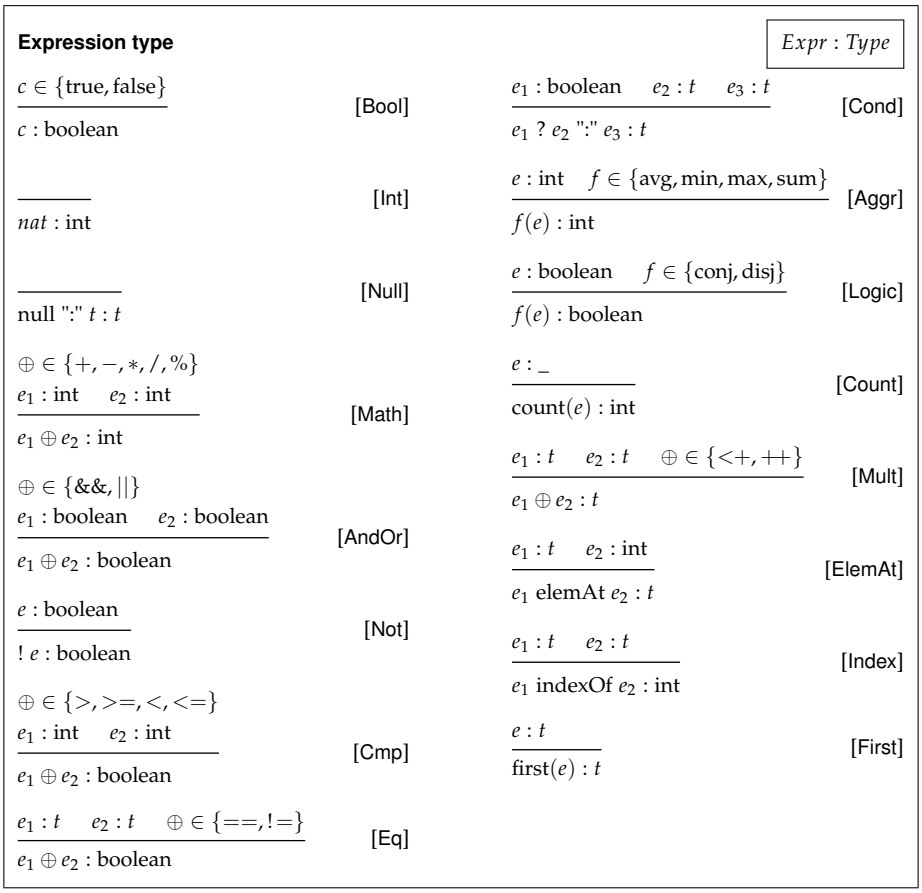


Figure A.5 Type rules of IceProof

more (Figure A.3). Consequently, the values in IceProof are lists of integers or lists of booleans (Figure A.4).

A.1.1 *Type System*

Figure A.5 shows the type system of IceProof. The type system features no type environment as none of the expressions introduce context. The type system is completely standard, except for the last four rules which deal with the fact that the values are lists. The choice and merge operator take two sub expressions of the same type are of this type as well and the `elemAt`, `indexOf`, and `first` operators are the standard list operations.

A.1.2 *Multiplicity System*

Figure A.6 shows the multiplicity system of IceProof. The basic idea is that expression with multiple operands take the Cartesian product of these operands. This means that binary operators mimic maybe-Monad behavior for zero or

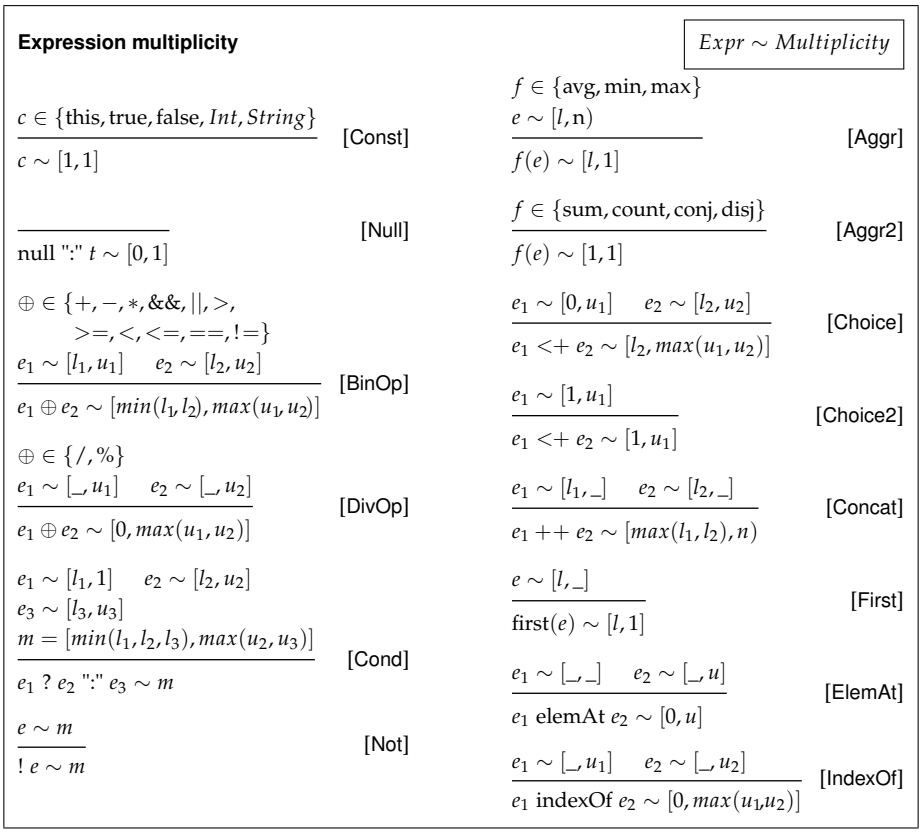


Figure A.6 Multiplicity rules of IceProof

one values: a maybe value as input for the computation returns a maybe value as output. The division and modulo operators exhibit slightly different behavior. Since dividing by zero has no result, at least one value in both operands might still result in no answer. Instead of throwing a division by zero exception zero answers are given for any denominator equal to zero. The Choice operator chooses at runtime the left expression if it has a result, and otherwise the right expression. This means its multiplicity is defined as the maximum of both upper and lower bound, except if the left lower bound is one. The Concat operator combines the results of both expressions. This means that at runtime it might always return more than one value; thus the upper bound is n. The `first` operator returns a value if its operand has one or more value, and the collection operations return at most as many values as their second operand.

A.1.3 *Dynamic Semantics*

Figure A.7 shows the dynamic semantics of IceProof. All the evaluation rules have a specific form: they operate on lists. A nice example of this is the

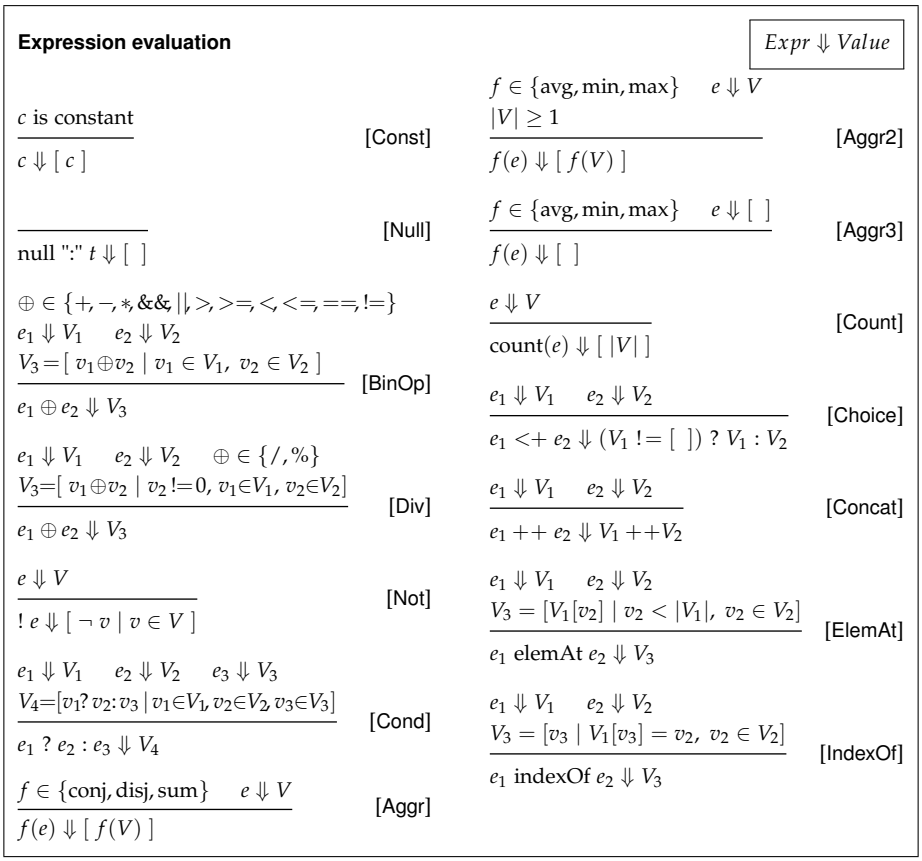


Figure A.7 Evaluation rules of IceProof

rule for binary operations. The left and right expressions evaluate to a list of values, the Cartesian product of these lists is taken, and on each pair of values the operator is applied. For single values a normal computation is performed, for maybe values a maybe computation and for many values a Cartesian product computation. Most evaluation rules follow this pattern. Some aggregation operations are defined for at least a single value (Aggr) while others are defined for all values (Aggr2). The choice operator returns the value of the left expression, if it has at least one value, otherwise the value of the right expression. The concat operator combines all values, regardless of how many there are. Finally, the list operations are also lifted: `elemAt` returns the elements at all provided indexes in all provided lists, and `indexOf` returns all indexes of all search elements in all lists.

A.2 TYPE PRESERVATION PROOF

With the language specified we can turn to the proofs. First we prove type preservation: if an expression has a type and evaluates to a value, then the

value is that type.

Theorem 6 (Type Preservation) $e : t \land e \Downarrow v \implies v : t$

Proof. Assume $e : t$ and $e \Downarrow v$. We need to show $v : t$. We do this by induction on the evaluation relation $e \Downarrow v$. This leads to 41 cases. By induction, v has a concrete structure in every case. We use this to refine or proof obligation to $t = Int$ or $t = Bool$. Also by induction, e has a concrete structure, we use this to get the concrete value of t as an assumption for all non-polymorphic expressions. This solves all non-polymorphic expressions. The polymorphic expressions (`if`, `<+`, `++`, `first`, and `elemAt`) need a proof for both types of values (integers and booleans), hence 10 cases. For these cases we use the types from the sub-expressions to derive the type of the overal expression. This solves the polymorphic cases. ∎

A.3 TERMINATION PROOF

Next, we prove termination for well-typed expressions. For the termination proof we use an executable function for the evaluation relation, `evalF`, such that the following is true.

Lemma 7 (EvalR equal evalF) $e \Downarrow v \iff evalF(e) = Some(v)$

Proof. The proof forward is induction over $e \Downarrow v$ with simple rewrites. The proof backward is induction over e, and destructs both `evalF` and $\Downarrow$ to show implication. ∎

We can use this lemma to execute the evaluation function in our termination proof.

Theorem 8 (Termination) $e : t \implies \exists v \ e \Downarrow v$

Proof. Proof by induction over $e : t$. The cases for literals are trivial. For all expressions with sub-expressions, use the induction hypotheses to get the values of the recursive calls. Next, we use type-preservation to derive the structure of these sub-expression values. Then we use the fact that we have evaluation as an executable function: we inline these values in the evaluation function body. At which point we have constructed the values that are returned from the evaluation function. These values are the $\exists v$ that we were looking for. ∎

A.4 MULTIPLICITY PRESERVATION PROOF

Last, we prove multiplicity preservation: if an expression has a type, a multiplicity, and evaluates to a value, then that value is of the right multiplicity.

Theorem 9 (Multiplicity Preservation) $e : t \land e \sim m \land e \Downarrow v \implies v \sim m$

Proof. Proof by induction over $e \Downarrow v$ with t and m independent. First, we derive all types and multiplicities of the sub-expressions. Then, we specialize our induction hypotheses to the types and multiplicities of the sub-expressions. At this point we let the proof assistant Coq mechanically examine all cases. We do case distinction on all sub-expression multiplicities and all sub-expression value lists (empty list, singleton list, or two-or-longer list). As IceProof has 4 multiplicites, and 3 interesting value lenghts, this generates $(4 * 3)^1$ cases for unary operators, $(4 * 3)^2$ cases for binary operators, and $(4 * 3)^3$ cases for the `if`. All these cases are trivial: either the assumptions are inconsistent (for example $e1 : [1, 1]$ with $e1 \Downarrow [true, false]$), or we can inline the values in the executable evaluation function (`evalF`), compute the output value, and conclude that it has the right multiplicity. ■

A.5 FUTURE WORK

There are several interesting properties to prove for IceDust. Especially the invariants listed in Chapter 4.

A.5.1 *Type- and Multiplicity-Safety*

The proofs in this Appendix only cover semantics without an environment or store. These proofs could be extended to cover expressions with environments and stores. When the IceDust object store is added, derived value expressions can be modeled as well. Though, at that point, the executable interpreter has to be changed to a non-terminating interpreter. This means the termination lemma has to be relaxed to progress, or to fuel-based interpretation.

A.5.2 *Preservation of bidirectionality*

If an IceDust interpreter with object store is created, it would also be interesting to prove the bidirectionality preservation invariant in Chapter 4 (Invariant 2). A simplified version of this proof could cover IceDust without derived values: just an object graph with getters and setters. This proof could then be extended with derived relations.

A.5.3 *Correctness of incremental calculation strategies*

For all programs that terminate with the non-incremental runtime, it holds that the incremental runtime returns the same value as the non-incremental runtime. It would be interesting to prove for these programs that incremental behavior results in the same values as "from scratch computation".

A simple version of this proof would cover a single IceDust object, or rather a language with no objects but just a set of global fields. This proof would include an invariant such as Invariant 1 from Chapter 4: either reevaluating all expressions results in exactly the cached values, or some up-to-date flag is false. Moreover, this proof would need include a simple version of the

path-based abstract interpretation from Chapter 3. The simplest proof can completely ignore multiplicities and only support a handful of expressions.

This proof could then be extended to cover IceDust (Chapter 3), with derived attributes and bidirectional relations. This would require modeling all features that interact with incremental updates: including multiplicities (to support path-inversion), and the full path-based abstract interpretation (to support conditionals).

Later, that proof could be extended to cover all IceDust features. Adding derived derived relations (Chapter 4) would mean that the dependency graph, which is used for dirty flagging, is not know statically, only when executing the interpreter. Covering multiple calculation strategies (eager and lazy incremental) would mean using different invariants for different strategies. The proof could then also be extended to cover calculation strategy composition (Chapter 4), proving that the strategy composition type system ensures the invariants of the various strategies under composition. Finally, the proof also could be extended with the inlining behavior of functions (Chapter 5).

Curriculum Vitae

Daco C. Harkes

1 March 1990
Born in Waddinxveen

2001-2007
VWO diploma
Driestar College in Gouda
Nature and Technology profile

2007-2010
B.Sc. in Computer Science
Delft University of Technology
Department of Mathematics and Computing Science
Cum laude (with honor)

2012-2014
M.Sc. in Computer Science
Delft University of Technology
Department of Mathematics and Computing Science
Cum laude (with honor)

2014-2018
Ph.D. in Computer Science
Delft University of Technology
Department of Software Technology

List of Publications

- Harkes, D. C. (2014). Relations: a first class relationship and first class derivations programming language. In Binder, W., Ernst, E., Peternier, A., and Hirschfeld, R., editors, *13th International Conference on Modularity, MODULARITY '14, Lugano, Switzerland, April 22-26, 2014*, pages 9–10. ACM.
- Harkes, D. C. and Visser, E. (2014). Unifying and generalizing relations in role- based data modeling and navigation. In Combemale, B., Pearce, D. J., Barais, O., and Vinju, J. J., 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.
- Harkes, D. C., Groenewegen, D. M., and Visser, E. (2016). Icedust: Incremental and eventual computation of derived values in persistent object graphs. In Krishnamurthi, S. and Lerner, B. S., editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.
- Harkes, D. C. and Visser, E. (2017). Icedust 2: Derived bidirectional relations and calculation strategy composition. In Müller, P., editor, *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, volume 74 of LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.
- ten Veen, N., Harkes, D. C., and Visser, E. (2018). Pixiedust: Declarative incremental user interface rendering through static dependency tracking. In *Companion of the The Web Conference 2018 on The Web Conference 2018*, pages 721–729. International World Wide Web Conferences Steering Committee.
- Harkes, D. C., van Chastelet, E., and Visser, E. (2018). Migrating business logic to an incremental computing DSL: a case study. In Pearce, D. J., Mayerhofer, T. and Steimann F., editors, *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*.
- Harkes, D. C. (2018). We should stop claiming generality in our domain-specific language papers (extended abstract). In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*.
- Harkes, D. C. (2018). We should stop claiming generality in our domain-specific language papers. In *The Art, Science, and Engineering of Programming*.

Daco Harkes was born in Waddinxveen, the Netherlands on March 1st, 1990. After he graduated from high school in 2007, he started a Bachelor in Computer Science and Technology at the Delft University of Technology, which he finished in 2010. In 2012 he started a Master in Computer Science at the Delft University of Technology, which he finished in 2014. From September 2014 to September 2018 he was a Ph.D. candidate with the Programming Languages group at the faculty of Electrical Engineering, Mathematics, and Computer Science of the Delft University of Technology, under supervision of Eelco Visser.

TU Delft