A graph-based approach to the automated discovery of data transformations

Douglas A. Creager
Oxford University Computing Laboratory
Linacre College
D.Phil. thesis

August 3, 2007

Supervisor: Andrew Simpson

A graph-based approach to the automated discovery of data transformations

Douglas A. Creager
Oxford University Computing Laboratory
Linacre College
D.Phil. thesis

Trinity 2007

Abstract

In recent years it has become much more common for software applications to communicate with each other directly. Internet connections have become a standard part of both office and home, and as more business processes and information move into the electronic realm, direct software communication will become even more prevalent. One of the largest deterrents to effective communication is the heterogeneous nature of the data and information involved. We cannot guarantee that two software systems that need to communicate will be running the same software or modeling their data in the same way. Obviously, the data must be somehow logically similar — otherwise, there would not be any meaningful communication possible. A key element of any modern communication protocol or framework must be a strategy for resolving any data mismatches that exist between the two sides.

The data mismatch problem is not new; unsurprisingly, there are many existing solutions to it. We would like to judge these solutions by two criteria: generality and automation. A *generic* solution will not needlessly limit the kinds of applications and data models that are supported. An *automated* solution will limit the amount of tedious, manual work needed to support a new application or data model. Unfortunately, none of the existing solutions are both sufficiently generic and sufficiently automated.

This thesis presents an automated solution to the data mismatch problem that is also fully generic: it makes absolutely no assumptions about the underlying data whatsoever. In order to achieve this generality, some automation must be sacrificed. Our approach requires that some *atomic* transformations be written manually. However, we can exploit the fact that transformations are composable — with a sufficient number of atomic transformations, a *compound* transformation can be automatically discovered between arbitrary datatypes. This approach is fully generic, since the transformation discovery algorithms require no knowledge of the structure or semantics of the datatypes involved; instead, the knowledge of a particular datatype is encapsulated into the atomic transformations that directly operate on it.

The contributions of this thesis are threefold. First, we present a graph-based model for transformations that has an efficient polynomial-time discovery algorithm. While efficient, this model is limited in that it can only represent *unary* transformations — those between one input and output datatype. This model is still surprisingly powerful; we present two case studies that show how this simple model can be used in the context of a real-world application, and what limitations it has.

Second, we present an extension to this graph-based model that supports *polyadic* transformations between multiple input and output datatypes, and show examples of how this increases the expressive power of the transformation graphs that we can create. Unfortunately, this expressiveness comes at a price: the naïve discovery algorithm for the new model runs in exponential space and time.

Finally, we show that polyadic transformation discovery is in fact worst-case *NP*-hard. Hopefully, the problem is only truly intractable for pathological inputs, and for real-world transformation graphs, compound transformations can be discovered with reasonable time and space requirements. We use a novel application of CSP to test this hypothesis, empirically exploring the complexity space of the problem and highlighting criteria for designing efficient transformation graphs.

Acknowledgments

This thesis is the product of three years of work here in Oxford, and unsurprisingly, it's gone through the usual phases you might expect: the exhilaration of making progress with a new idea, the tedium of "turning the crank" on the obvious bits that nevertheless need a thorough treatment, the "brilliant" new approaches that turn out to be useless in hindsight (and usually incorrect, to boot), and the annoyance and frustration that accompanies not knowing how to tackle a particularly nasty theorem in the math or bug in the code. The end result that you are holding would not be possible without the help and support of many, many people — so many that I am bound to forget a few due to my usual absentmindedness.

My love and gratitude obviously goes out to my family — Ken Creager, Marti Head, Dennis Yamashita, and Rachel Creager — whose support was not diminished by the distance of being back in the U.S.

My supervisor, Andy Simpson, has been more helpful than I could possibly have hoped for, and has become a solid friend in addition to being a colleague and mentor. (And also, as you can see from the date on the front, I won the bet!)

My office-mate and fellow teaching assistant, David Faitelson, kept me sane during the bad bits and helped celebrate during the good bits, and was integral in helping dream up distractions and grand plans — most of which, of course, we never got around to starting, let alone finishing.

My examiners, Jeremy Gibbons and Michael Butler, were very friendly and helpful. With the obvious nervousness involved in defending such a large piece of work, they set a great tone for my viva and gave me many great suggestions for improving the document that you're reading.

When I decided to become a teaching assistant as I started my degree, I couldn't have imagined how fulfilling and enjoyable it would turn out to be. My thanks to all of the students that I met for making it so. I must also thank my colleagues and partners in crime, both in the Software Engineering Programme: Ale Cavarra, Sarah Christou, Jim Davies, Jackie Jordan, Andrew Martin, Steve McKeever, Karen Roberts, and Shirley Sardar; and in the Computing Lab: Phil Armstrong, Bruno Oliveira, Ed Smith, and Nick Wu.

And finally, I must thank my many friends here in Oxford and back in Boston that made life outside of work so much fun: Sarah Boada-Momtahan, Mike Bonnet, Kate Constantino, Charlie Crichton, Tim Daoust, Max Devereaux, Martin Evans, Zinta Gercāns, Louise Glenn, Daniel Goodman, Dennis Gregorovic, Fran Griffin, Anat Grinfeld, Andy Hadcroft, Rich Hanna, Ele Hunter, Joss Knight, Betsy Krichten, Thomas Krülle, Mike Lilley, Jeff Mellen, Ruth Miller, Lee Momtahan, Marci Nantlova, Todd Nightingale, Douglas Russell, Sara Pierce, David Power, Simon Probert, Jon Salz, Kevin Schmidt, Clint Sieunarine, Mark Slaymaker, James Welch, and Tom Wrathmell.

Contents

List of Figures

List of Tables

1 Introduction

The World Wide Web [13, 12] represents one of the most significant recent advances in computer science and software engineering. The IP-based Internet [84, 83, 85] had already existed as a networking platform for nearly two decades before the advent of the Web; however, it was limited in scope to a small number of academic and government research laboratories. It was not until the Web that we had a content delivery platform able to exploit and drive the increasing reach of the Internet into the personal and business realms. The Internet is now ubiquitous: before, a connection to the global network was considered an extravagant luxury; now, it is a necessity.

This ever-present connection means that modern software applications are not the isolated systems of days past. Software systems must now exploit their connectivity by communicating with each other. This communication might be direct, with a network channel directly linking two applications, and messages passed between them according to some possibly sophisticated protocol. It might also be indirect and human-mediated, such as two colleagues exchanging spreadsheets via email.

A common theme is that the two applications must somehow understand each other's data models and encodings, at least partially, before any communication can take place. As an example, we can consider the simple case of two colleagues using different spreadsheet applications — such as one using Microsoft Excel and the other OpenOffice Calc. While this example obviously lacks the subtlety and sophistication of, for instance, a complex Web Service protocol, it still highlights the key underlying issue: these different applications will have different ways of modeling and encoding their data, and we must somehow reconcile these differences before communication can occur.

In fact, this issue, which we will call the *data mismatch problem*, is usually the major hurdle encountered when developing and integrating applications. An unfortunately high percentage of the work is spent on the "plumbing" between applications, and not on interesting new business logic enabled by the new communication capabilities. This has been mentioned as a problem in a wide range of fields, including medical informatics [21], industrial CAD software [51], the defense industry [90, 89], and integrated information systems both within [68] and between [99] enterprises. The problem has appeared under many names, including data interoperability, integration, translation, transformation, and conversion.

One obvious solution to the data mismatch problem is to agree on a standardized intermediary datatype that is independent of the different applications. Each application would then be responsible

for understanding only this extra datatype, instead of the datatypes of every other application. The OpenDocument group, for instance, has recently developed an open XML-based format [58] for storing spreadsheets and other office documents. This is used as the native format for the OpenOffice suite, and is supported by many competing office suites, as well. If all office suites were able to read and write this format, then communication between the applications would be trivial — data would be imported into each application from the standardized format, without requiring a separate translation routine for each supported datatype. However, there is not broad agreement on using the OpenDocument format as a global standard; Microsoft, which has by far the largest market share for office suites, has proposed an alternative standard based on its own Office Open XML format [38]. Both formats have been developed and standardized by industry consortia — by OASIS[1] for OpenDocument, and by Ecma[2] for Office Open XML — and both are in the process of becoming international standards recognized by the International Organization for Standardization (ISO). Each enjoys broad support within its own user community. Unfortunately, no consensus has developed between the communities as to which should be used as a single, global standard.

As another example, the Web Service stack of protocols [2] is an attempt to take the technologies developed for the World Wide Web and use them to provide a "service-oriented" paradigm for connecting heterogeneous enterprise applications. Following the design of the Internet protocol stack (Figure 1.1), the Web Service paradigm consists of a layered set of protocols, each of which provides a small, well-defined set of services to the other protocols and user applications.

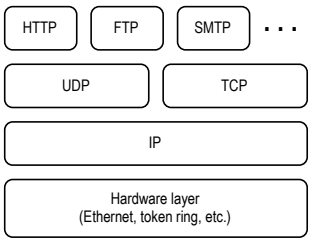


Figure 1.1: The Internet protocol stack

At the lowest end of this layered design, the Web Service stack simply uses existing World Wide Web protocols and specifications: HTTP [41] and the underlying IP stack [84, 85] for data transport, URIs [14] for resource identity and network addressing, and XML [18] for data serialization. Above this are the main "big three" Web Service standards: SOAP [49] for message encoding, WSDL [22] for service description, and UDDI [107] for service publishing and discovery.

Higher levels of the Web Service stack provide more sophisticated features: message-level security (such as encryption and digital signatures), application-level security (such as cross-domain authentication and authorization), multi-party transaction control, service orchestration and choreography, and many others. Solutions have not been fully developed for all of these areas, though the standards bodies are following a common, useful development strategy by fostering competition between many different prototype solutions, yielding a Web Service protocol stack that currently looks like that shown in Figure 1.2. Ideally, one would then distill an overall standardized solution by extracting the best features of each competing proposal. This has certainly happened, for instance,

[1] http://www.oasis-open.org/
[2] http://www.ecma-international.org/

with WS-Addressing [50] and WS-Security [75], which are now standards recognized by the W3C[3] and OASIS, respectively. One could expect that widely-accepted standards would also coalesce like this from competing solutions to other higher-level problems.

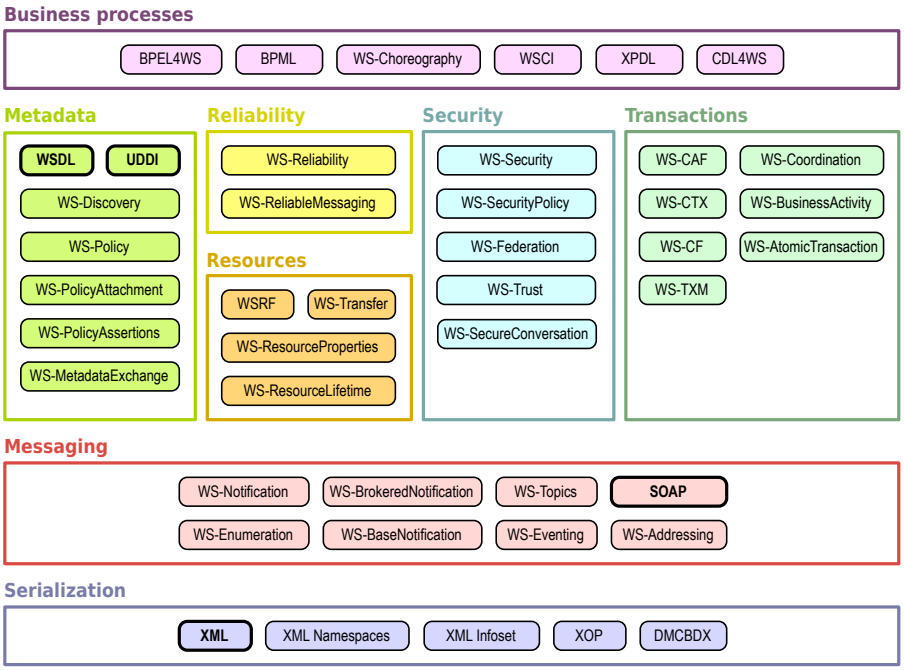


Figure 1.2: Current stack of Web Service protocols [53]

Unfortunately, the higher we look on the stack, the less consensus there is over which proposed solution should be adopted as "the" Web Service standard. Of course, much of this can be attributed to the complexity of the issues being addressed, and the relatively short amount of time spent so far on solving these problems. However, the competition-and-distillation process is slow, and is not guaranteed to produce an optimal solution. The companies and organizations that develop the original competing solutions invest a lot of time and money into their proposals, and have an understandable desire to have their proposal accepted as the standard, regardless of technical merit, so as not to waste that investment. This desire can run at odds with the need to produce an interoperable standard that is understandable and usable by many parties. This is true even at the lowest levels of the Web Service stack: SOAP and WSDL, for instance, are so extensible, and have so many corner cases, that it is possible to produce incompatible implementations of these relatively simple standards. This has led to the creation of a new "meta-standards" body, the Web Services Interoperability Organization[4] (WS-I), whose purpose is to define restricted subsets of the existing standards that better support interoperable implementations.

Even if we assume that the various Web Service stakeholders will eventually agree on noncompeting standards, there is not general consensus that the Web Service approach is the correct way to design a generic, Web-based distributed application platform. An alternative approach, based on Representational State Transfer (REST) [40], has been gaining popularity recently as a simpler paradigm

[3] Worldwide Web Consortium, http://www.w3.org/
[4] http://www.ws-i.org/

that more closely aligns with the original design criteria and strengths of the Web itself [106, 91].

These examples highlight an important assumption and limitation of the standardization process: standards only work if *everyone* involved is committed to developing, maintaining, and adhering to the same standard. For various commercial, political, and technical reasons, neither office format is likely to emerge as a single standard in the near future. Similarly, the debate over the relative merits of the Web Service and REST paradigms is not likely to be resolved soon, either.

In the case of the data mismatch problem, then, we must seek a solution that can gracefully handle the case when standardization efforts fail. From our point of view, the argument over the technical merits of the two office formats is irrelevant — it does not matter which side of the debate is "right". As software engineers, we find ourselves in a world that is heterogeneous in a fundamental way that was not the case before: there is no single authority that can *impose* a solution on all involved parties. We cannot use either office format as a single standard intermediary, since this would require a global consensus that does not exist.

This is not to say that open, independent standards are not needed. From a technical viewpoint, they obviously simplify the problem greatly. We should therefore *promote* and *desire* open standards whenever possible. Unfortunately, even if the standards are technically sound, we cannot *rely* on them as a solution to the data mismatch problem, because of the social nature of the standardization process. We must accept that we must deal with multiple datatypes, with mismatches that run the gamut from low-level syntax to high-level semantics, and which are developed by different stakeholders that might not agree with each other.

If we want to facilitate communication between applications, in the presence of this multitude of data formats and data models, we must provide a means of transforming or converting between them. We would like this solution to be both *automated* and *generic*. A naïve solution to the mismatch problem requires a manually-written translation for each pair of datatypes. Obviously, we prefer automated techniques that limit this manual analysis and coding as much as possible. Unfortunately, existing automated transformation techniques only work within specific contexts — requiring, for instance, that both datatypes be XML formats or relational database schemas. We prefer generic techniques that place no restrictions on the kinds of datatypes that are supported.

This thesis presents a graph-based solution to the data mismatch problem that is both highly automated and highly generic. Our approach is not *fully* automatic — we require that some *atomic transformations* be written manually. However, we can exploit the fact that transformations are composable: with a sufficient number of atomic transformations, a *compound transformation* can be automatically discovered between arbitrary datatypes. This approach is fully generic, though, since the transformation discovery algorithms require no detailed knowledge of the datatypes involved; instead, the knowledge about a particular datatype is encapsulated into the atomic transformations that directly operate on it.

The remainder of this thesis is organized as follows. Chapter 2 provides a more detailed overview of the data mismatch problem and existing solutions to it. Chapter 3 presents a generic theory of data that will allow us to formally reason about datatypes with differing syntax, structure, and semantics. We then show how several interesting problems, including data transformation, can be modeled even when the datatype definitions are fully opaque. Chapter 4 describes an initial version of our graph-based transformation framework; by exploiting opaque datatypes, this framework has a very efficient transformation discovery algorithm. Chapters 5 and 6 present case studies that show how this model

is useful in real-world scenarios. Chapter 7 then extends the model to support transformations with multiple inputs and outputs, and highlights the additional kinds of problems that can be solved with the increased expressiveness. Unfortunately, transformation discovery is provably *NP*-hard in this extended model; Chapter 8 uses a novel application of the CSP process algebra to explore the complexity space of the problem. This lets us show that the intractability is a worst-case bound, highlighting those kinds of transformation graphs that lend themselves to more efficient discovery. Finally, Chapter 9 summarizes the contribution of the thesis, and describes some potential areas of future work in this area.

2 Background

In this chapter, we describe the data mismatch problem in detail, showing how problems arise when dealing with heterogeneous datatypes — both those that are fairly static, and those that are more dynamic. Next, we provide an overview of existing solutions to the data mismatch problem, and show how none of them are both generic enough and automated enough for our purposes. Finally, we show how a classification from Ouksel and Sheth can be used to clarify and organize the different aspects of the data mismatch problem, including the descriptions of the datatypes that we must support, and possible solutions to the problem.

2.1 Data mismatch problem

The main hurdle to overcome when dealing with communicating software systems is the mismatch between their data models. This is true regardless of how the actual communication takes place. It is an obvious problem when the applications are linked directly by some kind of network channel, since the data sent by one application must be intelligible by the receiving application. It is also an issue when the communication is indirect and mediated by people. In this section we highlight the different problems that arise with heterogeneous datatypes: both when they are relatively static in nature, and when they are more dynamic, varying widely over time or use case.

2.1.1 Static datatypes

We first consider datatypes that are relatively well-defined and unchanging. As an example, we can consider in more detail the example, introduced in Chapter 1, of work colleagues exchanging spreadsheets via email.

The simplest case we can consider is when both colleagues are using the same version of the same application — Microsoft Excel, for instance. In this case, each application is just communicating with another copy of itself. Since both applications are exactly the same, it is trivial for them to understand each other's data models. Both colleagues' copies of Excel will make identical assumptions about the structure and representation of a spreadsheet, and will be able to read spreadsheets created by the other with no difficulty.

Of course, the communicating systems will not necessarily be exact copies of each other. For instance, if only one of the colleagues has upgraded to the latest version of Excel, a data mismatch occurs. Later versions usually need a new file format to support the new features introduced into the application. Within the context of a single application, this particular form of data mismatch is usually handled by ensuring that the newer versions are able to read and write the file formats of several previous versions. (Of course, more nefarious software developers might use this artificial incompatibility as a market pressure to incite users to upgrade to the newer, costlier version.) The developer will usually be able to reuse the file format code from previous versions, so within this limited scope, this is a perfectly acceptable solution. However, as we will see, this solution does not scale well at all.

The situation is slightly more complicated when we consider different applications, rather than different versions of the same application. For instance, one of the colleagues might use OpenOffice Calc instead of Excel. This means that the differences between the spreadsheet formats are more fundamental and harder to overcome. It is tempting to use the same solution as before, and somehow require both applications to read and write both spreadsheet formats. In certain cases, this is the solution used in real applications. OpenOffice Calc, for instance, has fairly robust support for reading Excel spreadsheets. Interestingly, this allows us to view "open" and "import" as the same operation: both translate a spreadsheet from some external format into the appropriate internal data structures. The "open" operation just happens to handle an external format that more closely corresponds to this internal representation. As before, in limited scopes, this solution is perfectly acceptable — as long as both colleagues explicitly instruct their applications to use a format common to both, they can exchange spreadsheets without difficulty.

Unfortunately, since the applications are developed by different organizations, it is less likely that code reuse can be used to amortize the cost of this solution. Worse, if we consider a third colleague using yet another spreadsheet application, such as KDE's KSpread, we see that requiring every application to support every format would quickly become cumbersome. One of our main goals in developing a solution to the data mismatch problem will be to prevent this.

2.1.2 Dynamic datatypes

All of the examples so far are fairly static — the new versions of the applications might require new versions of the office formats, but this will happen relatively infrequently. Therefore, the new revisions can just be considered distinct new datatypes. Another example that we want to support is when the internal structure of the datatypes can vary more rapidly. Most likely, this would be because the datatype is "extensible", and allows the user to add their own custom fields depending on their particular requirements and use cases.

A good example of this is the OME-XML microscope image format [46], developed as part of the Open Microscopy Environment project [104]. One of the assumptions of this project is that an image must be tightly coupled to its corresponding metadata, with both stored in the same file. This metadata is a mixture of static and dynamic: certain *curation metadata*, such as the image's owner, and the details of the microscope used to collect the image, have a static structure that will not change over time. The OME project has therefore standardized a useful set of these static metadata elements as part of the core OME-XML format.

On the other hand, other portions of an image's metadata will be more dynamic. This usually consists of the *computational results* that are generated while analyzing the image. These metadata elements are dynamic partly due to their time-dependent nature: as more analysis routines are run against an image over time, more computational results accumulate. More interesting, though, is that the metadata elements can vary by *use case*: different biological experiments will require completely different analysis routines for their images, which will generate completely different kinds of computational results. OME as a standardization group cannot anticipate all of the possible kinds of results that will be created by computational biologists — nor should it want to. Instead, the OME-XML format allows the user to define and include new metadata elements as needed.

To be truly useful, a transformation framework should be able to handle both purely static datatypes, such as the different office formats, and more dynamic ones, such as customized OME-XML. As we will see, this is not a trivial task; our design decisions when creating a framework will have a large impact on how well-supported dynamic datatypes will be.

2.2 Existing solutions

While the data mismatch problem is exacerbated by the heterogeneous nature of the Internet landscape, it is not a new problem. This section highlights many of the existing solutions to this problem. We will consider separately manual approaches and automated approaches.

2.2.1 Manual techniques

In the manual approach to the data mismatch problem, the software developer must write any translations that are needed by the application. This has the benefit that it is fully generic: since each translation routine is written manually, it can be perfectly tailored to the datatypes that it translates.

To illustrate this approach, we first assume that each application has its own specialized data format, as shown in Figure 2.1. In this figure, applications are shown as rounded rectangles, and data formats are shown as document icons. An application and a format are connected by a dashed line if the application can read and write that format. If two applications are connected to the same format, they can use that format to communicate. In Figure 2.1, Microsoft Excel and OpenOffice Calc cannot communicate with each other, since their associated formats are mutually unintelligible.

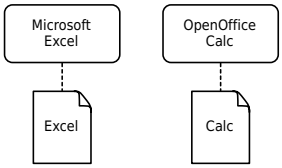


Figure 2.1: Applications with different data formats

The naïve solution is to allow each application to understand every datatype, as shown in Figure 2.2. In this case, either datatype can be used to facilitate communication, since both are understood by both applications.

Unfortunately, this approach does not scale well, as evidenced by Figure 2.3. Here we add two more applications, KSpread and WordPerfect, each with a specialized data format. We can easily see

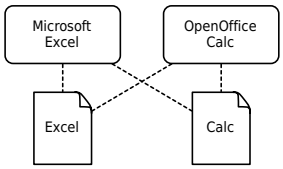


Figure 2.2: Applications that can understand many data formats

that $D \cdot A$ different translation routines are needed, where D is the number of data formats and A is the number of applications. In the special case where every application has its own data format, this simplifies to A^2 translation routines. Since each of these routines must be written manually, this approach is very intensive in terms of development time.

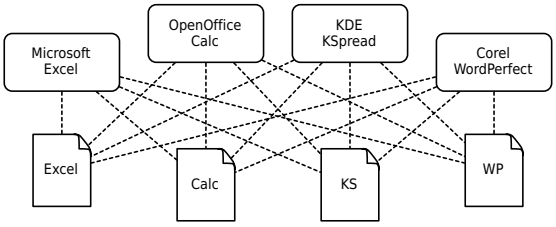


Figure 2.3: Many applications that can understand many formats

Since we cannot reduce the number of applications that we need to support, we must instead try to reduce the number of datatypes. Standardized formats do this very well: Figure 2.4 shows the hypothetical world where every spreadsheet application has agreed to use OpenDocument [58] as its file format. Each pair of applications can communicate, since they each understand the OpenDocument format. Since the standard mandates exactly one datatype, we now only have to manually write A translation routines.

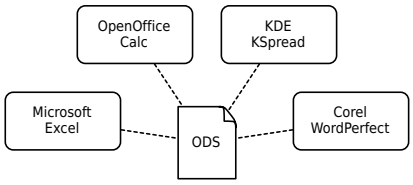


Figure 2.4: A standardized format reduces the number of translations needed

Of course, as we mentioned in the introduction, there is not universal support for OpenDocument as a single standard. Microsoft has developed the Office Open XML [38] format as a competing standard. Figure 2.5 shows a slightly more accurate version of the current support for these standards. Some office suites support both standards, while others only support one. We still have fewer translation routines to write than before, but we can no longer ensure that every pair of applications can communicate.

If we had a way to transform between the Office Open XML and OpenDocument formats, as shown in Figure 2.6, we would once again be able to ensure communication between every application. Unlike in Figure 2.5, we can now send a spreadsheet from Excel to KSpread. The transformation between the datatypes can occur in one of three places. It might be part of Excel's export routine, in

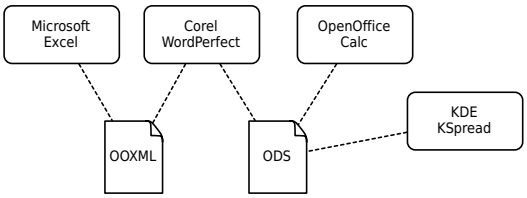


Figure 2.5: Competing standards can prevent communication

which case it appears to outside observers that Excel supports an additional export format. It might be part of KSpread's import routine, in which case it appears that KSpread supports an additional import format. Finally, it could be part of the communications channel itself. In our running example, this might correspond to one of the colleagues running the spreadsheet through a separate transformation tool before opening the document in their spreadsheet application. In this case, any transformation that occurs is hidden from both applications; instead, they read and write the data using their preferred data formats.

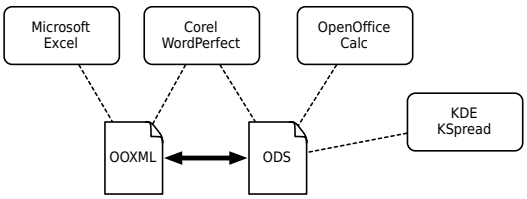


Figure 2.6: Transformations can overcome competing standards

In this example, we only have one transformation, so it seems reasonable to write it manually, like the import and export routines of the applications themselves. This does not work well if there are many transformations, though. We can apply the same idea to Figure 2.3, for instance; the result, shown in Figure 2.7, shows that we still require $O(D^2)$ transformations (instead of D^2 import/export routines) to ensure that each pair of applications can communicate. However, since we have decoupled the problem from the specific applications, we can try to *automatically* generate the transformations, once again reducing the amount of manual coding required.

2.2.2 Automated techniques

With a large number of possible data transformations to consider, we will want to automate the process of finding or creating them. There are several existing techniques for tackling this problem that can exploit the commonalities that will inevitably exist between the different datatypes. Two datatypes for recording a personnel record, for instance, will both contain some way to store the employee's name. With a detailed enough description of the particular datatypes (known as a *schema*), one can identify which elements map to each other. These mappings then provide a recipe for translating data from one schema to another.

In [87], Rahm and Bernstein present an overview of existing research in this area of *schema matching*. They first provide a generalized definition of a *Match* operator that, given two schemas, returns a set of mappings between elements of the schemas. They then identify several orthogonal

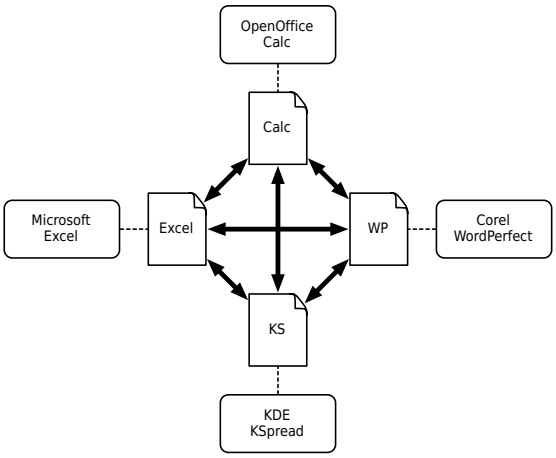


Figure 2.7: Using transformations without a single standardized format

criteria for classifying different schema matching techniques. Two criteria stand out as being most important. The first is whether the technique can find matches between complex compound structures in the schemas, or is limited to low-level atomic data elements; the second is whether mappings are found primarily by comparing the natural-language names of the data elements (*nominal typing*), or by analyzing the relationships between data elements and the constraints placed on them (*structural typing*). These criteria can be applied both to purely schema-based techniques, which consider only the explicit description of a data format, and to techniques that infer parts of the data's structure from the contents of actual data instances. (The latter can be used, for instance, to handle data from a "messy" or "noisy" source that does not conform to a strict, explicit schema description.) The authors then use this classification to compare and contrast several schema matching tools, including SemInt [64, 65, 66], the Learning Source Descriptions (LSD) framework [36, 35], the Semantic Knowledge Articulation Tool (SKAT) [74], TranScm [73], DIKE [81, 80, 79], ARTEMIS [20], and Cupid [67].

Most of these existing tools have been written with one or two specific modeling formalisms (e.g., the relational model [27], XML model [18], or object-oriented model [57]) in mind. This might seem to imply that these existing techniques do not exhibit the level of generality that we seek from a data transformation framework. It would rule out, for instance, the ability to handle proprietary binary formats and low-level differences in encoding. However, the underlying ideas usually work in the presence of any data model that has an appropriately detailed schema description, regardless of which formalism underlies the data. These techniques could then be extended, for instance, to binary datatypes described by an ASN.1 [56] schema. Extending an existing tool like this might be a time-consuming task, but it would be theoretically possible.

Another possible issue is that schema matching tools can only provide *candidate* mappings, since the nuanced semantics of the data elements cannot always be expressed in a form intelligible to the tool. Human intervention is therefore needed to verify or tweak the output of the matching tool. Nonetheless, this is still much less onerous than the alternative — having to manually examine each schema and derive the mappings by hand.

The main drawback to this approach is that the effectiveness of any particular schema matching technique on a particular set of data schemas can be highly variable. There is no one-size-fits-all

solution. The compositional approach described in this thesis recognizes this: it can incorporate data transformations created by different schema matching tools, in addition to those written manually. Thus we can exploit the strengths of schema matching in exactly those situations where it is most useful.

2.3 S classification

The spreadsheet example from Section 2.1 might seem trivial and contrived, but it allows us to consider certain aspects of the data mismatch problem without worrying about the actual differences between the data formats. The basic outline of the problem is the same regardless of whether we are considering two colleagues manually exchanging spreadsheets via email or two companies' intricate purchasing systems automatically sending each other electronic purchase orders.

However, at some point, we will obviously need to consider the datatypes themselves. Since we want a fully generic solution, we will need to consider a wide variety of data mismatches. Integers might be encoded in binary or as ASCII decimal strings. Dates might be expressed in an American "MM-DD-YYYY" format, a British "DD-MM-YYYY" format, or as the number of seconds elapsed since the Unix epoch on midnight, January 1, 1970. The size recorded in a compressed archive might refer to the compressed size or the uncompressed size. The dimensions of a manufactured product might be expressed in Imperial or metric units. If metric is used, the dimensions might be in meters, millimeters, or kilometers. A postal address might include a different set of fields depending on which country the user is in.

It will be useful to classify these different kinds of data mismatch to make them more manageable. In their study of information systems [78, 98], Ouksel and Sheth identify a useful classification, identifying mismatches as *systemic*, *syntactic*, *structural*, and *semantic*. We will call this the *S classification*. "System" refers to the particular combination of hardware and software used to implement an application. "Syntax" refers to the low-level representation of the data — usually in terms of a specific binary encoding. "Structure" refers to the underlying data primitives used to model an application domain — both which structures are available and how they are used. "Semantics" refers to the inherent meaning and interpretation of the data; the terms *information* and *knowledge* are often used instead of *data* to refer to semantic content. We will see several examples of how different aspects of a datatype fall into these four categories in the next chapter, when we develop a formalism for reasoning about datatypes.

The S classification has a natural ordering, from the "low-level" details of systems up to the "high-level" details of semantics. However, the boundaries between the four levels in the S classification are not necessarily crisp; depending on one's point of view, it can be a subjective decision whether to consider a certain property of a datatype syntactic rather than structural, for instance. However, for our purposes, crisp boundaries are not needed. We do not need the S classification to tell us whether integer endianness exists specifically at the system level or semantic level; instead, the classification is useful because it highlights a spectrum of properties, all of which we must consider.

It is interesting to point out that the data mismatch problem seems to be largely solved at the system level: it no longer matters what particular hardware and software is on each side of a network socket, as long as both correctly implement the underlying network protocol. This works because the standardization process has succeeded at this level: there is a universally agreed-upon suite of

standard network protocols, the most important being IP [84], ICMP [83], TCP [85], and UDP [82]. They provide an abstract view of the network connection as a "sequence of bytes". (In the case of TCP, this is even a "*reliable* sequence of bytes".)

One might hope, then, that similar universal standards can be agreed on to solve the data mismatch problem at higher layers of the S classification. XML [18], for instance, seems like an obvious choice as a universal syntax, while the Semantic Web's RDF [61, 8] and OWL [70, 101] have been touted as universal languages for structure and semantics. Unfortunately, we return to the same problem mentioned in the introduction — these must be truly *universal* if they are to be real solutions. The network socket abstraction is truly universal because our computers are, fundamentally, machines that operate on sequences of bytes. The syntax, structure, and semantics levels, on the other hand, are much more flexible. They are more dependent on the needs of an application, and even on the whims of the developers designing and implementing a system. XML cannot be a guaranteed universal syntax, for any number of reasons. Many systems that we must support are legacy applications with pre-existing non-XML syntaxes. Some data models do not map well to XML's hierarchical model. The markup overhead can make XML an inefficient syntax for certain use cases. And of course, we must consider the human element — some developers might choose not to use XML out of sheer stubbornness.

The mantra is the same: we must accept that there will be a variety of choices at each level of the S classification, and we need to support them all. The system level is the *only* place where it is safe to assume a single, universal solution. A rogue developer might decide to develop a new data format that is not based on a sequence of bytes, but they would quickly find themselves without a computer system to implement it on. The same is not true at any other level of the S classification.

A key feature of the transformation framework presented in this thesis is that it does not assume any universal language for syntax, structure, or semantics. In fact, it does not even make the system-level "sequence of bytes" assumption. The ideas behind the framework work equally well for datatypes with infinitely many elements. We feel that this generality is an important requirement for any application or theory that claims to embrace the heterogeneous world brought about by the global Internet.

Summary

In this chapter we have presented a more detailed overview of the data mismatch problem, and showed how the underlying issues are exacerbated when the datatypes in question are highly dynamic over time or use case. We can use the *S classification* from Ouksel and Sheth to examine many of these issues and solutions: for instance, by organizing a datatype's description into the differing levels of system, syntax, structure, and semantics.

The obvious manual techniques for solving the data mismatch problem are clearly unsuitable, due to the amount of cumbersome work that is required as the number of supported applications and datatypes grows. Intermediary datatypes, which are understood by many applications, are helpful at relieving this tedium; unfortunately, even when the perfect intermediary datatype exists, we cannot rely on any centralized authority to mandate its use.

An alternative solution is to rely on *translations* between datatypes, especially if the discovery and development of these translations can be automated. There are many existing attempts in the

literature to provide this functionality. Though many of them are not fully automated, requiring manual supervision and verification of the results, this still represents a dramatic improvement over fully manual techniques. Unfortunately, none of these techniques are fully generic, since each only works within a limited problem domain. We believe that these techniques lack full generality because they require a precise description of each datatype, from which translations between the datatypes are inferred. Any data description language that could truly describe *any* datatype would be far too complex for this translation inference to be tractable. In the following chapters, we will present a framework that abandons these precise datatype descriptions, making transformation discovery both fully general and more efficient.

3 Generic theory of data

As we have mentioned in the previous chapters, one of the main goals of our transformation framework is that it should be *fully generic*: there should be no restrictions on the kinds of datatypes that are supported. Existing transformation techniques are not fully generic, only supporting, for instance, relational or XML datatypes. This lack of generality is due to the fact that these techniques require a precise description of each datatype; even if it were possible to design a data description language that is fully generic, it would be exceptionally complex, and impossible to work with or reason about in practice.

To make our transformation framework fully generic, we abandon this notion of precisely described datatypes. Later chapters will describe *transformation graphs*, whose constituent *atomic transformations* will encapsulate all knowledge about each of the datatypes. Thus, the datatypes can be fully opaque from the point of view of the higher-level transformation discovery logic. This opacity will be the key feature that enables the efficiency and full generality of our approach. However, it does have some ramifications that we must remain cognizant of. In this chapter we develop a formalism that lets us reason about opaque datatypes; this is done by treating the equivalences between datatypes as first-class objects. We first provide conceptual examples of the equivalences and datatypes that we should support. We then formalize these notions, and show how datatypes can be defined by a set of *interpretations*, along with logical *constraints* that show how the interpretations are related. These interpretations and constraints can be defined at any appropriate level of detail; specifically, we show that two important issues — canonicalization and transformation — can be modeled and reasoned about even when the datatypes and equivalences are fully opaque. The contents of this chapter have been published previously in [30].

3.1 Overview

In this section we show how the seemingly simple ideas of “data” and “equivalence” are complicated when we consider the full range of the S classification, and present informal descriptions of several datatypes that we want our theory to support.

3.1.1 Data equivalences

A key feature to take into account when designing a data theory is *data equivalence*. What do we mean when we say that two data instances are "equivalent"? A naïve answer would be to define this based on binary equality — two program variables that both contain the 32-bit integer "73" are obviously equivalent. However, this does not capture the entire picture. We present a few obvious counterexamples.

First, we can consider low-level encoding details that can affect data equivalence. For instance, computer processors have a property called *endianness* that affects how multi-byte numbers are stored in memory. "Big-endian" processors store these numbers with their most-significant byte first, whereas "little-endian" processors store the number's least-significant byte first. As an example, consider the number 1,000, which can be encoded in hexadecimal as the 16-bit quantity `03E8`. When encoded on a big-endian machine, this number is represented by the byte string ⟪`03 E8`⟫. When encoded on a little-endian machine, however, the byte string becomes ⟪`E8 03`⟫. In one sense, that of binary equality, the data are not equivalent; in another, equally valid sense, that of integer equality, they are. This inconsistency holds in reverse, as well. Consider the byte string ⟪`03 E8`⟫. As before, on a big-endian machine, this evaluates to the integer 1,000. On the little-endian machine, however, this is interpreted as the hexadecimal number `E803`, or 59,395. In this case, the data are equivalent according to binary equality, but not according to integer equality.

To further complicate matters, both of the previous examples assumed that the integers were unsigned. Modern computers encode signed integers using *two's complement notation*, which has the beneficial property that the same binary addition operator can be used for both signed and unsigned numbers. This causes a further inconsistency in how a particular byte string can be interpreted as an integer. For example, on a big-endian machine, the byte string ⟪`E8 03`⟫ is interpreted differently as a signed integer (-6,141) than as an unsigned integer (59,395), whereas on a little-endian machine, it is interpreted as 1,000 regardless of signedness. This is another case where data can be equivalent according to binary equality, but not according to integer equality. These examples are summarized in Figure 3.1.

Figure 3.1: Differing semantic interpretations of binary integers

Similar inconsistencies can appear at higher abstraction levels. For instance, in the HTML markup language [86], it is possible to specify the background color of a Web page with the `bgcolor` attribute of the opening `body` tag. To give a Web page a white background, for instance, one could use the following:

```
<body bgcolor="white">
```

This example represents the color using one of the values in the list of named color strings specified by the HTML standard. It is also possible to specify the color by giving an explicit color value in the RGB color space, such as:

```
<body bgcolor="#ffffff">
```

This example specifies a background color that has the maximum value of 255 ("ff" in hexadecimal) for each of its red, green, and blue components; this color happens to be the color white. These two examples are not equivalent according to binary equality, or even according to character string equality (which can be different from binary equality due to character set issues). However, the semantics of the `bgcolor` attribute, as defined by the HTML standard, are such that the character strings "white" and "#ffffff" represent equivalent colors.

Thus, it is easy to see that a true notion of data equivalence is very application-dependent. It is also very dependent on the level of abstraction being used — two data that have different binary encodings might be semantically equivalent, and vice versa. Sometimes semantic equivalence will be more important; sometimes syntactic equivalence will. The S classification, described in Section 2.3, provides a useful way to classify these abstraction levels and the different forms of equivalence that we want to support.

3.1.2 Datatypes

Any study of data needs to think about datatypes. Broadly speaking, we define a *datatype* to be some set of data. Notionally, a datatype is different from an arbitrary set of data, because the data instances that constitute a type are supposed to be "similar" in some way. Exactly what form this similarity takes will be application-dependent, just like our notion of data equivalence. To illustrate this, we present several example datatypes, and show how the S classification helps classify them.

Integers

As our first example we can consider the integer types. This is a very low-level set of types; its syntax is a binary string, or sequence of bytes. As we have seen in previous sections, our interpretation of these bytes depends on several factors. At the system level, we must know the integer's endianness, as this affects the order of the bytes in the sequence. At the structural level, we must know the length (and therefore numeric range) of the integer; this is necessary, for instance, to know how much space to reserve in memory for the integer value. At the semantic level, we must know whether the application intends to use this integer as a signed or unsigned value.

Each of these levels can be seen as imposing constraints on which particular data instances can appear in the datatype's set: an integer datatype contains all of the instances that are encoded as a byte string of a particular length, and are interpreted as integers with a particular endianness and signedness. Taken together, this constraint-based definition of the datatype's set brings our original vague notion of "similarity" into focus — but only for this particular datatype.

Postal address (XML)

Next we look at a higher-level type — a postal address encoded in XML. This data type might be used, for example, to send "electronic business cards" between address book applications. An instance of this datatype is shown in Figure 3.2.

At the semantic level, this datatype represents a postal address. As people who have grown up with a postal system, we are able to encapsulate a lot of semantic meaning into this concept. This

```
<address>
  <name>Douglas Creager</name>
  <company>Oxford University Computing Lab</company>
  <line1>Wolfson Building</line1>
  <line2>Parks Road</line2>
  <city>Oxford</city>
  <postcode>OX1 3QD</postcode>
  <country>UK</country>
</address>
```

Figure 3.2: Example instance of the postal address XML type

datatype does not provide us with a way to directly encode this semantic meaning in the data, apart from the natural language names of the XML tags, but it will inform how we write applications that use this data.

At the syntax level, we are using the Extensible Markup Language (XML). Therefore, by extension, our datatype implicitly includes all of the syntactic assumptions and requirements of the XML standard [18]: for instance, a binary string that is not well-formed XML cannot be a valid instance of our datatype.

At the structural level, we have an XML schema (not shown) that specifies which XML tags must be used, the content of those tags, and the order in which the tags must appear. As at the syntax level, this implicitly includes into the datatype definition all of the structural assumptions and requirements of our XML schema: a well-formed XML document that does not match our schema is not a valid instance of our datatype.

At the system level, things are more vague, and will depend in part on the details of the application that is accessing the data. Further, the different aspects of the system interpretation of the datatype are interrelated with the interpretations of the other three levels. Our application will need to have some sort of XML parser, which will handle the syntax level. It will also need application-level logic for parsing the abstract document tree, taking care of the structural level. The application itself will be written with some intuitive notion of what an address actually *is*, taking care of the semantic level. In addition, there will be the low-level details of the application itself, such as the hardware and operating system that it is running on, and any shared libraries that it uses. However, none of these issues affect whether a particular binary string is an instance of our datatype, and can be safely ignored.

Again, we can look at these levels as imposing constraints on the members of the datatype's set: the set contains all of those data instances that are encoded in XML, conforming to this particular address schema, and that are used as "postal addresses" within the context of some application.

Postal address (database)

As another example, we might decide to store these postal addresses in a relational database. This could correspond to an address book application's internal state of the various business cards that someone has collected. An instance of this datatype is shown in Figure 3.3.

Semantically, this datatype represents a postal address, just as in the previous example. Specifically, this means that the semantic-level constraints imposed on the corresponding sets are the same for both of these datatypes.

ADDRESSES table

ADDRESS_ID	13
NAME	"Douglas Creager"
COMPANY	"Oxford University Computing Lab"
LINE_1	"Wolfson Building"
LINE_2	"Parks Road"
CITY	"Oxford"
POST_CODE	"OX1 3QD"
COUNTRY_ID	30

COUNTRIES table

COUNTRY_ID	30
NAME	"United Kingdom"
ABBREV	"UK"

Figure 3.3: Example instance of the postal address database type

Structurally, however, they are obviously quite different. The tables used in this example are based on the relational model, which is quite different from the hierarchical model of XML. Instead of using an XML schema to define which tags must appear in the tree of XML data, we have a database schema that defines which relational tables we use, and how the tables relate to each other.

The system and syntax levels of this example are rather blurred. Relational databases do provide an application-visible syntax in the SQL query language, but this does not provide a syntactic representation of the data itself. In fact, we have several similar datatypes that are equivalent semantically and structurally, but different syntactically. We could be referring to the internal representation used by a particular database management system, such as PostgreSQL or Oracle, to store the data on disk and in memory. We could be referring to the wire format used by the database server to send back to the application the results of a query that returns this exact set of data. We could be referring to the equivalent SQL `INSERT` statements that could be used to reconstruct the data. We could be referring to the abstract notion of a relational tuple, in which case there is no actual low-level syntax that can be represented in a computer. Often, these syntactic differences will not matter, and we can exploit data independence by ignoring them. At other times, they will be important, and must be included in the datatype definition.

Postal address (Semantic Web)

As one final example, we can describe a third postal address datatype, which uses the formalisms and notations of the Semantic Web [15]. The Semantic Web provides a data representation that is better able to express the semantics of the data involved. It does this by representing data using *subject-predicate-object* triples as defined by the Resource Description Framework (RDF) [61, 8]. One can envision these triples as edges in a graph, with the subject being a source node, the object being a destination node, and the predicate being a labeled edge connecting the two. This graph notation is used in Figure 3.4 to show how a postal address could be expressed in the Semantic Web. (Technically, we should give full URIs [14] for the labels of the edges and the address1 and uk nodes; we provide shorter labels for brevity.)

Semantically, this datatype once again represents a postal address; however, by using subject-

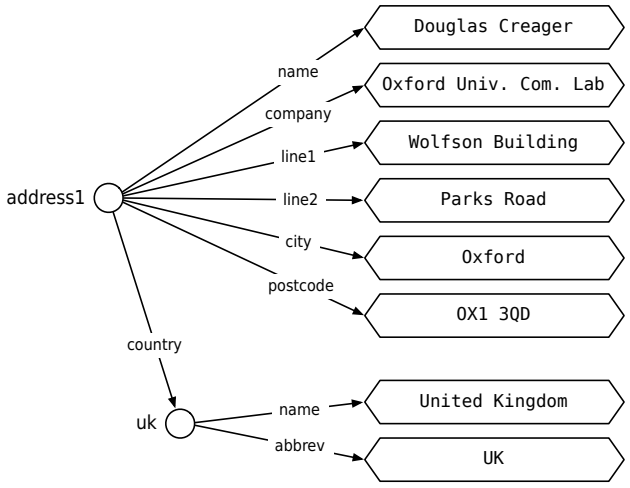


Figure 3.4: Example instance of the postal address Semantic Web type

predicate-object triples, we have encoded a version of these semantics into the data more directly. Of course, this leaves unanswered the question of how to use this semantic information in the design of an application.

Syntactically, the Semantic Web uses XML to encode these graphs of RDF triples, so in one very specific, low-level sense, this datatype is similar to the XML postal datatype described previously. Structurally, however, not just any XML data is allowed — Semantic Web data must exist in a well-formed RDF graph, encoded in XML in a specific way. So while the XML syntax is used for both datatypes, they differ greatly in structure. As with the previous examples, the Semantic Web provides a schema language — the Web Ontology Language (OWL) [70, 101] — for stating which particular semantic structures are used. Our datatype would include an OWL ontology describing the overall structure of the graph in Figure 3.4. RDF graphs that do not match this ontology would not be instances of this datatype.

3.1.3 Data with multiple interpretations

Our theory of data also needs to support *polysemantic* data instances, which have identical concrete representations, but different interpretations. In our framework, a particular data instance can belong to a different datatype depending on the use case; usually this is due to subtle semantic differences in how the data is interpreted.

As an example, we can consider a datatype used to store data collected from some piece of scientific equipment — for example, a digital microscope. Often, some form of preprocessing is required to "clean up" the raw data before it is used in later analyses. In the case of microscope images, this usually involves a *deconvolution* of the image pixels based on the transfer functions of each lens and filter that the light passed through in the microscope. The raw and deconvolved images are usually stored using the same binary file format. In some cases, it might be important to distinguish between raw and deconvolved images; to do this, we can treat them as separate datatypes. Syntactically and structurally, these datatypes are identical; the only difference between them is a semantic assumption about what preprocessing has (or has not) already been applied to the image.

In other cases, it might not matter whether an image has already gone through deconvolution; we can model this with a datatype where we make no assumption about what preprocessing has occurred.

In this example, then, we have three distinct datatypes: "raw" images, "deconvolved" images, and "unspecified" images. Any particular image file could be an instance of exactly two of these types: every image can be an instance of "unspecified", and can also be an instance of either "raw" or "deconvolved", depending on whether it has been preprocessed. This highlights one of the main ramifications of opaque datatypes: every important, distinguishable difference between data has to be modeled by separate datatypes, even if, in all other respects, the datatypes would be superficially identical.

Depending on the details that are recorded in the image file, it might be difficult to infer from an arbitrary image whether it has been deconvolved. If there is a "has been deconvolved" field, and we can trust that it is filled in correctly, then the decision is simple. If the field does not exist, or is untrustworthy, then we cannot easily decide. We must rely on some outside agent — such as the user, or the type signature of the code that creates the data — to explicitly provide the type of a particular data instance. In general, even when it is possible to infer a data instance's type automatically, the details of the deduction will be different for each example, and will require detailed internal knowledge of the datatypes, violating opacity. Therefore, we will always assume that the type of each data instance is provided explicitly. In those cases where automatic deduction is possible, this can be handled by another system, with the necessary specific knowledge of the datatypes in question; its answer will then be passed along to our generic transformation framework just like any other typing information.

3.2 Formalization

The example datatypes described in the previous section were not particularly complex. Even so, they were able to incorporate several formalisms that represent data in completely different ways. A fully generic theory of data must be able to incorporate all of these datatypes, regardless of the differences in the underlying formalisms. In this section we describe such a theory, using a simple running example to provide clarity.

The notation used is based on the Z notation [102, 103, 108]. However, in places where it makes our formalization more clear, we depart from strict adherence to the Z standard in three ways. First, we allow implicit universal quantification: when defining a function, for instance, the function's parameters will be implicitly quantified over their types. Second, we allow functions to be overloaded: two functions can have the same name as long as it is always clear from their types which function is being declared or used in any given context. Finally, we do not require explicit constructor names for free types. Implicit free type constructors might seem to lead to ambiguities; for instance, as we will see later in Definition 4.2, an instance i of type *Integer* can also be used as a *Value*. To do so, the implicit *Value* constructor must be used to "promote" the *Integer* into a *Value*. The worry is that ambiguities can arise, where a particular expression might have many possible promotions that are all type-correct. However, in all of our specifications, this will not occur; if an expression needs to be promoted to conform to the required type, there will only be one promotion that is possible.

We will revisit each of these extensions the first time they are used.

3.2.1 Datatypes and equivalences

In order to talk about data, we must first define it. Since we are aiming for full generality in this type theory, we cannot assume any kind of structure when referring to data — it must be considered completely opaque. We can also define an *equivalence*, denoted with a $\doteq$, which defines an equivalence relation (which is reflexive, symmetric, and transitive) between data. Note that an equivalence is only a *partial* equivalence relation on the entire universe of data; it only becomes a proper equivalence relation when restricted to a particular domain of data.

$$[Datum]$$

$$Equivalence : \mathbb{P}\,(Datum \leftrightarrow Datum)$$

$$\begin{aligned}
&\forall \doteq : Datum \leftrightarrow Datum \bullet \\
&\quad \doteq \in Equivalence \Leftrightarrow \\
&\qquad \forall\, d : \operatorname{dom} \doteq \bullet\ d \doteq d \land \\
&\qquad \forall\, d_1, d_2 : \operatorname{dom} \doteq \bullet\ (d_1 \doteq d_2) \Rightarrow (d_2 \doteq d_1) \land \\
&\qquad \forall\, d_1, d_2, d_3 : \operatorname{dom} \doteq \bullet\ (d_1 \doteq d_2 \land d_2 \doteq d_3) \Rightarrow (d_1 \doteq d_3)
\end{aligned}$$

As mentioned above, datatypes are represented as sets of data. We can define a simple is-a relation between data instances and datatypes. By defining this as a function, it follows that the *Datatype* sets are pairwise disjoint.

$$Datatype == \mathbb{P}\ Datum$$

$$\_ \text{ is-a } \_ : Datum \to Datatype$$

$$\forall\, d : Datum;\ t : Datatype \bullet d \text{ is-a } t \Leftrightarrow d \in t$$

However, we have also said that a datatype is not just any set of data; the data in question must be similar in some way. We will express this similarity by defining *interpretations* and *constraints* for each datatype. The interpretations and constraints can both be classified using the S classification. It might seem that disjoint *Datatype* sets prevent us from properly supporting the polysemantic datatypes from Section 3.1.3. However, interpretations allow us to do exactly that. The *Datum* instance is an abstract entity, and does not represent the data instance's concrete representation. Instead, the concrete representation is one of the *Datum*'s (possibly many) interpretations. Since we will allow multiple data instances (which might belong to different datatypes) to have identical interpretations, we can correctly support polysemantic data — they will have identical concrete interpretations and differing semantic interpretations.

We can apply these ideas to one of the integer types mentioned in Section 3.1.2. There are multiple integer datatypes, since bit length, endianness, and signedness all affect the integer interpretation. For simplicity, we will look at one integer datatype in particular: 16-bit, little-endian, and unsigned.

$$Integer_{16,L,U} : Datatype$$

Our first task is to specify the datatype's interpretations. In the case of the integer datatypes, there are two interpretations: an instance's binary encoding, and its integer value. We use the $_{\mathsf{Syn}}$ subscript to indicate that the binary interpretation is syntactic, and the $_{\mathsf{Sem}}$ subscript to indicate that

the integer interpretation is semantic. Both interpretations are defined as partial functions:

$$\begin{aligned} &\mathrm{binary}_{\mathsf{Syn}} : \mathit{Datum} \pfun \mathit{ByteString} \\ &\mathrm{integer}_{\mathsf{Sem}} : \mathit{Datum} \pfun \mathbb{Z} \end{aligned}$$

In the first case, we define the binary interpretation using the byte string type specified in Appendix A. Similarly, we define an integer interpretation in terms of Z's integer type ($\mathbb{Z}$). It is important to point out that this integer interpretation is not the same as any concrete representation of an integer — rather, it is an abstract mathematical concept that fully captures the semantics of an "integer". With these interpretations in place, we can formalize our notion of binary equivalence and integer equivalence. Two data are binary-equivalent if their binary interpretations are equal; similarly, they are integer-equivalent if their integer interpretations are equal.

$$\begin{aligned} &\_ \doteq_{\mathrm{bin}} \_ : \mathit{Equivalence} \\ &\forall\, d_1, d_2 : \mathrm{dom} \doteq_{\mathrm{bin}} \bullet \\ &\quad (d_1 \doteq_{\mathrm{bin}} d_2) \Leftrightarrow (\mathrm{binary}_{\mathsf{Syn}}\ d_1 = \mathrm{binary}_{\mathsf{Syn}}\ d_2) \end{aligned}$$

$$\begin{aligned} &\_ \doteq_{\mathrm{int}} \_ : \mathit{Equivalence} \\ &\forall\, d_1, d_2 : \mathrm{dom} \doteq_{\mathrm{int}} \bullet \\ &\quad (d_1 \doteq_{\mathrm{int}} d_2) \Leftrightarrow (\mathrm{integer}_{\mathsf{Sem}}\ d_1 = \mathrm{integer}_{\mathsf{Sem}}\ d_2) \end{aligned}$$

In both cases, we did not restrict the interpretation to the $\mathit{Integer}_{16,L,U}$ datatype. Instead, we defined it as a generic property that can be applied to any *Datum*, since there are many other datatypes that might be encoded in binary or interpreted as an integer. The interpretations are both partial functions, though, because not every *Datum* has a binary or integer interpretation. We must then apply these generic properties to our specific datatype:

$$\begin{aligned} &\mathit{Integer}_{16,L,U} \subseteq \mathrm{dom}\ \mathrm{binary}_{\mathsf{Syn}} \\ &\mathit{Integer}_{16,L,U} \subseteq \mathrm{dom}\ \mathrm{integer}_{\mathsf{Sem}} \end{aligned}$$

After defining the interpretations, we must also specify the datatype's constraints. Each of these constraints will depend in some way on at least one of the interpretations. First, we have the structural constraint that our integer type is 16 bits (or two bytes) long. This is defined in terms of the datatype's binary interpretation. Note that this is a two-way constraint; we must not only say that each of our integers is 16 bits long, but also that every 16-bit binary string corresponds to a valid instance of this type.

$$\forall\, i : \mathit{Integer}_{16,L,U} \bullet \#(\mathrm{binary}_{\mathsf{Syn}}\ i) = 2$$

$$\forall\, b : \mathrm{Bytes}\ 2 \bullet \exists_1\, i : \mathit{Integer}_{16,L,U} \bullet \mathrm{binary}_{\mathsf{Syn}}\ i = b$$

Our other constraint states how the binary and integer interpretations relate to each other, which we can calculate using the functions of Appendix A. This constraint is informed by both the system-level endianness property and the semantic-level signedness property. As before, the constraint is two-way: we must explicitly state that every integer interpretation in the correct numeric range has a corresponding instance of the $\mathit{Integer}_{16,L,U}$ datatype.

$$\forall\, i : \mathit{Integer}_{16,L,U} \bullet \mathrm{integer}_{\mathsf{Sem}}\ i = \mathrm{unsigned}\ \mathrm{littleEndian}\ \mathrm{binary}_{\mathsf{Syn}}\ i$$

$$\forall\, z : 0..(2^{16} - 1) \bullet \exists_1\, i : \mathit{Integer}_{16,L,U} \bullet \text{integer}_{\text{Sem}}\ i = z$$

This completes a formal specification of this particular integer type. The other integer types can be defined analogously.

3.2.2 Opaque datatypes

The amount of detail that went into the description of this integer datatype highlights an important distinction in our formalism. The $\mathit{Integer}_{16,L,U}$ datatype had a *full specification* — we provided a complete, formal description of both of the datatype's interpretations, and of the constraints that relate them. In this particular case, this full specification was not overly verbose. We were able to use Z's existing mathematical integer type ($\mathbb{Z}$) to model the semantics of an integer, and it was relatively straightforward to provide a formal definition of binary data (*ByteString*) in Appendix A.

Often a complete formal description is not readily available, and the effort involved in developing a precise definition might not be worth the benefit gained from doing so. In these cases, it is possible to provide a datatype with a *partial specification*, where we define some of the interpretations and constraints as abstract entities. This becomes especially useful when considering how multiple partially-specified datatypes relate to each other. If *every* interpretation and constraint is left abstract and undefined, whether by choice or necessity, we then have a *fully opaque* datatype. Fully opaque datatypes will be important in the remainder of this thesis, in that they will allow us to develop more efficient algorithms by purposely ignoring many details of the datatypes that we work with. Later parts of this chapter will show that this opacity, which is the key to the efficiency of our methods, does not affect the correctness of our algorithms.

As an example, we can revisit the postal address types, which have new interpretations that were not used by the integer datatype. However, whereas we provided (or were given) full definitions of the $\mathbb{Z}$ and *ByteString* types, we will leave these new interpretations abstract:

$$[\mathit{XMLDocument}, \mathit{XMLSchema}]$$
$$[\mathit{RelationalTuple}, \mathit{RelationalSchema}]$$
$$[\mathit{PostalAddress}]$$

XMLDocument represents the logical document tree of an XML document, while *RelationalTuple* represents a row from some relational table. In both cases, we have also mentioned a type that represents the schema that describes the data's structure. *PostalAddress* represents the semantic meaning of a postal address; this allows us to talk about the real-life concept of an address used by the postal service to identify a physical location, though we do not need to provide any details of how this semantic meaning is represented. We can now define interpretations and equivalences for the postal address datatypes, similarly to the integer example:

$$\begin{array}{l} \text{xml}_{\text{Struct}} : \mathit{Datum} \pfun \mathit{XMLDocument} \\ \text{relational}_{\text{Struct}} : \mathit{Datum} \pfun \mathit{RelationalTuple} \\ \text{address}_{\text{Sem}} : \mathit{Datum} \pfun \mathit{PostalAddress} \end{array}$$

$$\begin{array}{l} \_ \doteq_{\text{xml}} \_ : \mathit{Equivalence} \\ \hline \forall\, d_1, d_2 : \text{dom} \doteq_{\text{xml}} \bullet \\ \quad (d_1 \doteq_{\text{xml}} d_2) \Leftrightarrow (\text{xml}_{\text{Struct}}\ d_1 = \text{xml}_{\text{Struct}}\ d_2) \end{array}$$

$$\_ \doteq_{\text{rel}} \_ : \mathit{Equivalence}$$

$$\forall\, d_1, d_2 : \operatorname{dom} \doteq_{\text{rel}} \bullet \\ (d_1 \doteq_{\text{rel}} d_2) \Leftrightarrow (\text{relational}_{\text{Struct}}\ d_1 = \text{relational}_{\text{Struct}}\ d_2)$$

$$\_ \doteq_{\text{addr}} \_ : \mathit{Equivalence}$$

$$\forall\, d_1, d_2 : \operatorname{dom} \doteq_{\text{addr}} \bullet \\ (d_1 \doteq_{\text{addr}} d_2) \Leftrightarrow (\text{address}_{\text{Sem}}\ d_1 = \text{address}_{\text{Sem}}\ d_2)$$

XML equivalence implies that two data instances contain the same structure of XML elements and attributes. Relational equivalence implies that two data instances are identical tuples from identical relational tables. Address equivalence implies that two data instances are postal addresses that refer to the same physical location, regardless of how the instances are structured or encoded.

With these interpretations defined, we can define the datatypes themselves. The XML address datatype will have binary, XML, and address interpretations; the relational address datatype will have relational and address interpretations. (We ignore the syntax of the relational datatype to maintain data independence and simplify the definitions.)

$$\mathit{Address}_{\text{XML}} : \mathit{Datatype}$$

$$\mathit{Address}_{\text{XML}} \subseteq \operatorname{dom} \text{binary}_{\text{Syn}} \\ \mathit{Address}_{\text{XML}} \subseteq \operatorname{dom} \text{xml}_{\text{Struct}} \\ \mathit{Address}_{\text{XML}} \subseteq \operatorname{dom} \text{address}_{\text{Sem}}$$

$$\mathit{Address}_{\text{Rel}} : \mathit{Datatype}$$

$$\mathit{Address}_{\text{Rel}} \subseteq \operatorname{dom} \text{relational}_{\text{Struct}} \\ \mathit{Address}_{\text{Rel}} \subseteq \operatorname{dom} \text{address}_{\text{Sem}}$$

Next we specify the constraints, for which we will need several helper functions and relations, which, again, we do not provide full definitions for:

$$\text{encodes}: \mathit{ByteString} \pfun \mathit{XMLDocument}$$

$$\text{instanceof}: \mathit{XMLDocument} \rightarrow \mathit{XMLSchema} \\ \text{instanceof}: \mathit{RelationalTuple} \rightarrow \mathit{RelationalSchema} \\ \mathit{AddressSchema}_{\text{XML}} : \mathit{XMLSchema} \\ \mathit{AddressSchema}_{\text{Rel}} : \mathit{RelationalSchema}$$

$$\text{interpret}: \mathit{XMLDocument} \pfun \mathit{PostalAddress} \\ \text{interpret}: \mathit{RelationalTuple} \pfun \mathit{PostalAddress}$$

The encodes function maps a byte string to the XML document that it represents. (The function is partial since not all byte strings represent valid XML documents.) We have not said *how* to derive an XML document from the binary string; we have just said that we *can*. We could easily provide a definition for encodes; a simple one would be

$$\text{encodes} = (\text{binary}_{\text{Syn}}{}^{\sim}) \fatsemi \text{xml}_{\text{Struct}}$$

However, as mentioned previously, we will purposely ignore any such definition, since we want to show that we can consider fully opaque datatypes.

The instanceof relations and interpret functions are the first examples of overloaded Z definitions,

one of our extensions to the notation. The two flavors of instanceof allow us to verify that an XML document or relational tuple matches its corresponding schema. We also mention the particular schemas used by our XML and relational datatypes. The two flavors of interpret allow us to determine the semantic meaning of an XML document or relational tuple. These are then applied to the datatypes as constraints:

$$
\begin{aligned}
&\forall\, d : \mathit{Address}_{\mathsf{XML}} \bullet \\
&\quad (\mathsf{binary}_{\mathsf{Syn}}\ d)\ \mathsf{encodes}\ (\mathsf{xml}_{\mathsf{Struct}}\ d) \land \\
&\quad (\mathsf{xml}_{\mathsf{Struct}}\ d)\ \mathsf{instanceof}\ \mathit{AddressSchema}_{\mathsf{XML}} \land \\
&\quad \mathsf{interpret}\ (\mathsf{xml}_{\mathsf{Struct}}\ d) = \mathsf{address}_{\mathsf{Sem}}\ d
\end{aligned}
$$

$$
\begin{aligned}
&\forall\, d : \mathit{Address}_{\mathsf{Rel}} \bullet \\
&\quad (\mathsf{relational}_{\mathsf{Struct}}\ d)\ \mathsf{instanceof}\ \mathit{AddressSchema}_{\mathsf{Rel}} \land \\
&\quad \mathsf{interpret}\ (\mathsf{relational}_{\mathsf{Struct}}\ d) = \mathsf{address}_{\mathsf{Sem}}\ d
\end{aligned}
$$

This provides a formal, fully opaque rendering of the datatype definitions in Section 3.1.2. For an XML postal address, its binary encoding must match its logical XML document; this XML document must conform to the postal address schema; and the document must have some valid real-world interpretation as a postal address. Similar constraints apply to relational postal addresses. In the next sections we will show that we can still reason about interesting data-related problems when the datatypes in question are opaque.

3.2.3 Canonicalization

One example that highlights the importance of differing notions of equivalence is *data canonicalization*. A well-known current example of canonicalization involves XML documents and digital signatures [37, 16, 17].

The problem stems from the fact that every XML document has many different encodings as a concrete sequence of bytes. Three aspects of the XML syntax, in particular, affect the encoding of a document: attributes, namespaces, and whitespace. In most XML applications, these differences are not an issue, since the application works with a high-level view of the XML content; this is often in the form of the Document Object Model API [63], which represents an XML document by its abstract tree structure. However, one application area where these differences are important is *digital signatures*. Briefly, digital signatures are a more cryptographically-secure version of checksums and error-correcting codes. They provide a means of attesting that the content of a document has not been modified in transit between two parties. This is an important security feature in modern applications that helps prevent, among other things, man-in-the-middle attacks.

The algorithms used to implement digital signatures are not constrained to XML documents; they work on any binary payload. This can be problematic when signing XML documents. Alice can send an XML document to Bob, signing it before sending it along the communications channel. However, there might be communications gateways in between Alice and Bob that modify the binary representation of an XML document without modifying the document structure. When Bob receives the document, its binary representation will have changed, and Alice's signature will no longer match the document.

Looking at this in terms of our datatype formalism, we can define a function that can sign a byte string:

$[Signature]$

$$\mathsf{sign} : ByteString \rightarrow Signature$$
$$\forall\, b_1, b_2 : ByteString \bullet (\mathsf{sign}\ b_1 \neq \mathsf{sign}\ b_2) \Rightarrow (b_1 \neq b_2)$$

This captures the essence of a digital signature: if the signatures match, the byte strings most likely match as well; conversely, if the signatures do not match, the byte strings are different. Often, we consider this signature property to be a true equivalence. However, it is more accurate to say that $\doteq_{\mathsf{bin}}$ is a refinement of $\doteq_{\mathsf{xml}}$, since $\doteq_{\mathsf{xml}}$ defines more data instances to be equivalent than $\doteq_{\mathsf{bin}}$. The number of signatures is much smaller than the number of binary strings, so some overlap is inevitable. This refinement is "close enough" to an equivalence, though, to still be useful; even though we cannot provide a full guarantee, matching signatures *strongly imply* that the binary strings are the same.

We can define a similar function for signing data that simply signs a datum's binary interpretation; signatures then work for arbitrary data, too, *but only under binary equivalence*:

$$\mathsf{sign}_{\mathsf{bin}} : Datum \nrightarrow Signature$$
$$\forall\, d : \mathrm{dom} \doteq_{\mathsf{bin}} \bullet \mathsf{sign}_{\mathsf{bin}}\ d = \mathsf{sign}\ \mathsf{binary}_{\mathsf{Syn}}\ d$$
$$\forall\, d_1, d_2 : \mathrm{dom} \doteq_{\mathsf{bin}} \bullet (\mathsf{sign}_{\mathsf{bin}}\ d_1 \neq \mathsf{sign}_{\mathsf{bin}}\ d_2) \Rightarrow (d_1 \not\doteq_{\mathsf{bin}} d_2)$$

We run into a problem in the case of XML. Alice's and Bob's applications do not care about binary equivalence; they care about XML equivalence. The hope, then, is that the signature predicate holds for XML equivalence, too:

$$\forall\, d_1, d_2 : \mathrm{dom} \doteq_{\mathsf{xml}} \bullet (\mathsf{sign}_{\mathsf{bin}}\ d_1 \neq \mathsf{sign}_{\mathsf{bin}}\ d_2) \stackrel{?}{\Rightarrow} (d_1 \not\doteq_{\mathsf{xml}} d_2)$$

For this to be the case, we would need the following implication to hold:

$$\forall\, d_1, d_2 : \mathrm{dom} \doteq_{\mathsf{xml}} \bullet (d_1 \not\doteq_{\mathsf{bin}} d_2) \stackrel{?}{\Rightarrow} (d_1 \not\doteq_{\mathsf{xml}} d_2)$$

However, we know this is not true; two different byte strings *can* represent the same XML document.

What is needed is a *canonicalization function*. In the case of XML documents, we need to choose one particular binary encoding for each logical XML document. We would then define a $\mathsf{canon}_{\mathsf{xml}}$ function that maps an XML datum to its canonical binary encoding. The required property would then hold:

$$\forall\, d_1, d_2 : \mathrm{dom} \doteq_{\mathsf{xml}} \bullet (\mathsf{canon}_{\mathsf{xml}}\ d_1 \doteq_{\mathsf{bin}} \mathsf{canon}_{\mathsf{xml}}\ d_2) \Leftrightarrow (d_1 \doteq_{\mathsf{xml}} d_2)$$

Two XML documents that have the same logical structure, when canonicalized, would also have the same binary encoding. Expressed another way, two data that are XML-equivalent, when canonicalized, would also be binary-equivalent. When signing XML documents, we must then ensure that we sign the binary representation of the *canonicalized* XML:

$$\mathsf{sign}_{\mathsf{xml}} : Datum \nrightarrow Signature$$
$$\forall\, d : \mathrm{dom} \doteq_{\mathsf{xml}} \bullet \mathsf{sign}_{\mathsf{xml}}\ d = \mathsf{sign}\ \mathsf{binary}_{\mathsf{Syn}}\ \mathsf{canon}_{\mathsf{xml}}\ d$$
$$\forall\, d_1, d_2 : \doteq_{\mathsf{xml}} \bullet (\mathsf{sign}_{\mathsf{xml}}\ d_1 \neq \mathsf{sign}_{\mathsf{xml}}\ d_2) \Rightarrow (d_1 \not\doteq_{\mathsf{xml}} d_2)$$

In fact, we can define canonicalization as a generic property that a function might provide between any two equivalences:

$$DataFunction == Datum \pfun Datum$$

$$\begin{array}{|l} \_ \text{ canonicalizes } [\_/\_]: \\ \quad DataFunction \rightarrow (Equivalence \times Equivalence) \\ \hline \forall\, f : DataFunction;\ \doteq_1, \doteq_2 : Equivalence \bullet \\ \quad f \text{ canonicalizes } [\doteq_1 / \doteq_2] \Leftrightarrow \\ \qquad \mathrm{dom} \doteq_1 \subseteq \mathrm{dom}\, f \land \\ \qquad \mathrm{ran}\, f \subseteq \mathrm{dom} \doteq_2 \land \\ \qquad \forall\, d_1, d_2 : \mathrm{dom} \doteq_1 \bullet (d_1 \doteq_1 d_2) \Leftrightarrow (f\, d_1 \doteq_2 f\, d_2) \end{array}$$

With this generic property defined, we can easily state that the $\mathsf{canon}_{\mathrm{xml}}$ function canonicalizes XML equivalence in terms of binary equivalence:

$$\begin{array}{|l} \mathsf{canon}_{\mathrm{xml}} : DataFunction \\ \hline \mathsf{canon}_{\mathrm{xml}} \text{ canonicalizes } [\doteq_{\mathrm{xml}} / \doteq_{\mathrm{bin}}] \end{array}$$

It should be noted that this formalism does not help us find a detailed definition of the $\mathsf{canon}_{\mathrm{xml}}$ function. In general, the definition of a canonicalization function will be highly dependent on the details of the underlying data formalism and how this relates to its binary encodings. However, if desired (and beneficial), one could use this formalism to prove that a fully-defined function correctly canonicalizes two equivalences.

3.2.4 Transformations

This formalization of datatypes and equivalences also gives us a convenient way to describe transformations. They will be represented by data functions, just like the canonicalization example from the previous section. The key feature of a transformation that makes them useful as a solution to the data mismatch problem is that they *maintain* some equivalence between disparate datatypes.

We can return once again to the postal address example, and consider two address book applications: one which uses the relational datatype, and one which uses the XML datatype. Since the datatypes both refer to postal addresses, instances of these types can be semantically address-equivalent ($\doteq_{\mathrm{addr}}$). Therefore, in theory, the two applications can communicate. We can model this situation similarly to the canonicalization example, reusing the *DataFunction* type from that section. We need to introduce the notion of *typing* the data functions, however:

$$\begin{array}{|l} \mathsf{source} : DataFunction \rightarrow Datatype \\ \mathsf{dest} : DataFunction \rightarrow Datatype \\ \hline \forall\, f : DataFunction;\ t : Datatype \bullet \\ \quad \mathsf{source}\, f = t \Leftrightarrow \mathrm{dom}\, f \subseteq t \land \\ \quad \mathsf{dest}\, f = t \Leftrightarrow \mathrm{ran}\, f \subseteq t \end{array}$$

We can define the *source* and *destination* datatypes for a data function; this simply states that all of the function's input or output values come from the respective datatype. These are defined as functions, since datatype sets are disjoint.

A data function then *links* its source type to its destination type:

$$\_ \text{ links } [\_ \rightarrow \_] : \mathit{DataFunction} \rightarrow (\mathit{Datatype} \times \mathit{Datatype})$$

$$\forall f : \mathit{DataFunction};\ t_S, t_D : \mathit{Datatype} \bullet f \text{ links } [t_S \rightarrow t_D] \Leftrightarrow (\text{source } f = t_S) \land (\text{dest } f = t_D)$$

Lastly, a data function *maintains* an equivalence if that equivalence holds between each of the function's inputs and the corresponding output:

$$\_ \text{ maintains } \_ : \mathit{DataFunction} \leftrightarrow \mathit{Equivalence}$$

$$\forall f : \mathit{DataFunction};\ \doteq : \mathit{Equivalence} \bullet f \text{ maintains } \doteq \Leftrightarrow (f \subseteq \doteq)$$

With these definitions in place, we can state the existence of the required transformation: it links the XML and relational postal address datatypes ($\mathit{Address}_{\mathsf{XML}}$ and $\mathit{Address}_{\mathsf{Rel}}$), and maintains the postal address semantic equivalence ($\doteq_{\text{addr}}$).

$$\mathsf{xform}_{\text{Address}} : \mathit{DataFunction}$$

$$\mathsf{xform}_{\text{Address}} \text{ links } [\mathit{Address}_{\mathsf{XML}} \rightarrow \mathit{Address}_{\mathsf{Rel}}]$$
$$\mathsf{xform}_{\text{Address}} \text{ maintains } \doteq_{\text{addr}}$$

Note that once again, we have abstracted away a lot of unnecessary detail — we have said nothing about how $\mathsf{xform}_{\text{Address}}$ performs this transformation.

Since transformations are modeled as functions between data, they are also composable. This allows us to consider sequences of datatypes, and sequences of data functions:

$$\mathit{TypeSequence} == \text{seq}_1\ \mathit{Datatype}$$
$$\mathit{FunctionSequence} == \text{seq}_1\ \mathit{DataFunction}$$

$$\_ \text{ types } \_ : \mathit{TypeSequence} \leftrightarrow \mathit{FunctionSequence}$$
$$\text{source} : \mathit{FunctionSequence} \rightarrow \mathit{Datatype}$$
$$\text{dest} : \mathit{FunctionSequence} \rightarrow \mathit{Datatype}$$

$$\forall ts : \mathit{TypeSequence};\ fs : \mathit{FunctionSequence} \bullet$$
$$ts \text{ types } fs \Leftrightarrow$$
$$\#ts = \#fs + 1 \land$$
$$\forall i : 1 \mathinner{..} \#fs \bullet \text{source } fs\,(i) = ts\,(i) \land \text{dest } fs\,(i) = ts\,(i+1) \land$$
$$\text{source } fs = (\mathit{head}\ ts) \land$$
$$\text{dest } fs = (\mathit{last}\ ts)$$

A sequence of functions is *well-typed* if the destination type of each data function matches the source type of its successor. We can then define the source and dest operators for sequences, much as they are defined for individual functions: the source (destination) of a function sequence is the source (destination) of the first (last) function in the sequence.

With these definitions, we can define a compose operator on function sequences:

$$\text{compose} : \mathit{FunctionSequence} \rightarrow \mathit{DataFunction}$$

$$\forall fs : \mathit{FunctionSequence} \bullet \text{compose } fs = \fatsemi / (fs)$$

$$\forall\, fs : \mathit{FunctionSequence} \bullet \\ \quad \mathsf{source}\ fs = \mathsf{source}\ (\mathsf{compose}\ fs) \land \\ \quad \mathsf{dest}\ fs = \mathsf{dest}\ (\mathsf{compose}\ fs)$$

The operator is defined using distributed composition on the functions in the sequence.

Because of the transitivity of data equivalence, it is obvious that a compound transformation maintains an equivalence if each transformation in its sequence maintains it. Each atomic transformation, and the compound transformation that results, might be partial, so the compound transformation need not generate a transformed value for every instance of the source datatype. However, if it does, we can be sure that the original and transformed instances are equivalent according to the equivalence in question.

Summary

In this chapter, we have presented a theory of data that supports full generality; it does this by treating the equivalences between data instances as first-class objects, and by defining a datatype in terms of a number of *interpretations* and the logical *constraints* that relate them together. We can define these interpretations and constraints abstractly, stating that they exist without providing any structural details of their definitions. Datatypes defined in this abstract way are said to be *fully opaque*.

Importantly, many data-related problems, such as *canonicalization* and *transformation*, can still be expressed and reasoned about when the datatypes in question are fully opaque. The distinguishing feature of a canonicalization function is that it operates within the context of a single datatype, providing a "bridge" between two equivalences that disagree about instances of that datatype. Transformations, on the other hand, translate from one datatype to another, ensuring that some equivalence is maintained by this translation. Further, compound transformations, which consist of a sequence of transformations composed together, are able to maintain any equivalence that is maintained by each of its constituent transformations. In both cases, these properties are true regardless of the underlying structure of the datatypes and equivalences in question.

In the remainder of this thesis, we will show how to use a variety of graph-based structures to construct a framework for discovering compound transformations automatically. The fact that we can do this in the presence of fully opaque datatypes will be an important reason for the efficiency of our method.

4 Transformation graphs

In previous chapters, we presented the motivations behind our transformation framework, and its two major requirements: automation and full generality. In Chapter 3, we showed how we can still reason about many interesting properties of data when the datatype definitions are fully opaque. We will now exploit this fact to develop an efficient transformation discovery framework. The efficiency and correctness of our technique is a direct consequence of fully opaque datatypes: conceptually, since we are able to ignore many of the complex details of a datatype definition, we greatly reduce the amount of information we must consider when searching for a solution.

In this chapter, we present a graph-based framework that allows us to use opaque datatypes to efficiently discover transformations between arbitrary datatypes. We first present a conceptual overview of the transformation graph and describe its main elements: datatypes and atomic transformations, which form the graph's basic structure; properties, which allow the same graph to be used for multiple use cases, which will likely have different optimality criteria for the discovered transformations; and declaration patterns, which allow repetitive features of a transformation graph to be defined more concisely. We then present a language and file format that can be used to define transformation graphs, with support for each of these main elements. Finally, we present a formalization and denotational semantics for the graph language, serving two purposes. First, it shows that the graph language is well-founded and consistent with itself. Second, and more importantly, it can be seen, along with the transformation graph file format, as providing a language-independent specification for implementing this framework. A prototype implementation, written in Python[1], has been developed; its overall design and graph construction logic is identical to that of the denotational semantics. Other implementations (or a more robust version of the Python prototype) could be developed along the same lines. Parts of this chapter have previously appeared in [29].

4.1 Overview

In this section we present a conceptual overview of our graph model for describing and reasoning about transformations. A *transformation graph* consists of *datatypes* and *atomic transformations*. The datatypes are sets of similar data instances, as defined in Chapter 3; a datatype will specify the

[1] http://www.python.org/

instances at each level of the S classification. Atomic transformations represent the ability to translate directly from one datatype to another, while maintaining some equivalence between them. Atomic transformations are composable; this allows a directed path in a transformation graph to represent a sequence of atomic transformations that can be composed into a *compound transformation*. A compound transformation path, highlighted in red in Figure 4.1, represents a transformation between arbitrary datatypes in a transformation graph. As shown in Chapter 3, a compound transformation maintains any equivalence that is maintained by each of its constituent atomic transformations.

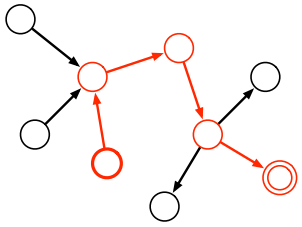


Figure 4.1: A transformation path between two types in a graph

Since arbitrary transformations are represented by directed paths, any standard pathfinding algorithm, such as Dijkstra's [34] or Bellman-Ford [9, 42], can be used as a discovery algorithm. These pathfinding algorithms allow the edges in the graph to be weighted, which causes the algorithms to find the paths with the smallest sum of weights, rather than with the fewest number of edges. We want transformation graphs to be reusable across multiple use cases; therefore, instead of defining weights directly for each atomic transformation, we allow the structures of the graph to have an arbitrary set of *properties*. The client can then use a custom *weight provider* to calculate an appropriate numeric weight for each edge given the properties defined on its atomic transformation. Our model also supports *declaration patterns*, which allow repetitive features of a transformation graph to be defined more concisely. Finally, though we introduce this graph-based approach as a solution to the data mismatch problem, it is in fact more general, and can be used to solve any problem whose solution can be specified by an inductive property. In this section, we describe all of these elements of the transformation framework in more detail.

4.1.1 Datatypes and transformations

The core elements of a transformation graph are *datatypes* and *atomic transformations*. The datatypes are represented by the nodes of the graph. These are datatypes as defined in Chapter 3 — they include any syntactic, structural, or semantic details necessary to fully understand the layout, encoding, and interpretation of data of this type. The atomic transformations represent the ability to directly translate an instance of one datatype into another, while maintaining some equivalence. In this section, we use a running example to show how datatypes and transformations are used to construct transformation graphs.

Images

As an example, Figure 4.2 shows two datatypes representing images: generic TIFF, representing an image in the TIFF format [1], and generic JPEG, representing an image in the JPEG format

[54]. Also shown in the figure are two atomic transformations for translating between these two image types. This could be custom code written by the image processing application that will use this transformation graph; alternatively, the transformations could be implemented using an existing library such as the Java Advanced Imaging API[2] or the ImageMagick C library[3].

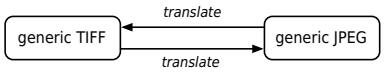


Figure 4.2: A transformation graph with two image formats

Because of the nature of the two image formats, the translate transformations are not always reversible. The TIFF format usually encodes the image pixels using a *lossless* encoding: it is possible to decode the binary representation of the pixels into the exact pixel matrix that was used to create the encoding. The JPEG format, on the other hand, uses a *lossy* compression format, where the decoding process produces a pixel matrix that is very similar, but not identical, to the original. Lossy algorithms like this are able to achieve higher compression ratios, and therefore more efficient storage of an image, at the cost of precision. The algorithms are designed such that the human eye — the usual intended client of an image — cannot readily tell the difference between the original and decoded pixels.

As mentioned, this can have consequences on the translate transformations. The translation between a JPEG image and a TIFF image is not affected, since the TIFF format is able to faithfully reproduce the compressed pixel matrix that is the result of the lossy JPEG algorithm. The same is not true, however, when translating a TIFF image into a JPEG, since the result of the transformation is not a faithful reproduction of the original TIFF pixel matrix. Depending on what our application needs to do with the images, this may or may not be a problem. Therefore, we cannot determine in advance whether the TIFF-to-JPEG translation will be a useful transformation to include in our application's transformation graph.

Metadata extraction

To continue the example, our application might want to extract certain metadata elements from multiple image formats — for instance, the image's dimensions, original format, creator, and creation date — and encode this information in a specialized XML format. As shown in Figure 4.3, we can add a new datatype to the graph to represent this metadata.

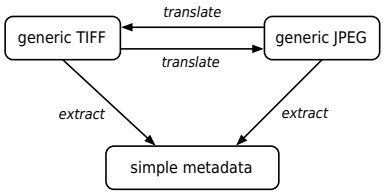


Figure 4.3: Adding a simple metadata type to the graph

[2] http://java.sun.com/products/java-media/jai/
[3] http://www.imagemagick.org/

The simple metadata type, being an XML datatype, will likely have associated with it an XML schema document (XSD) that describes its structure. This XSD could be used for validation purposes, or simply as documentation for application developers.

The graph also contains transformations for extracting the metadata from each of the image datatypes. Both transformations will hard-code the metadata's format field to "TIFF" or "JPEG", respectively. They will also extract whatever metadata they can from the images themselves. The transformations might not always be able to extract the full set of metadata from the images, both because the formats differ in their capabilities for storing metadata, and because individual images might not include all of the possible metadata tags allowed by the format.

An interesting feature of the extraction transformations is that, unlike the translation transformations, it is not possible to write their inverse. That is, given the metadata information about an image, it is not possible to reconstruct some representation of the image itself, since we have thrown away the pixel data in the extraction process. It might seem that the image and its metadata are distinct entities, and therefore that the extraction process is not something that we can represent as a transformation in our graph. However, the two datatypes do refer to the same logical entity — the image — even if they provide markedly different representations of that entity. This is similar to the notion of a *resource* as defined by the REST architectural style [40], where a resource is any abstract "thing that can be named", and which can have many concrete representations. Since the resource itself is abstract, a REST application is only able to directly handle a resource's representations.

This leads to the notion of *resource equivalence*. The image and metadata datatypes are resource-equivalent since they are different concrete representations of the same resource. The extraction process is a valid transformation since it maintains this equivalence. On the other hand, if we were to consider a metadata type that stores information about a *collection* of datatypes, then we would be working with a separate resource. The two datatypes would not be resource-equivalent, so the extraction process could not maintain resource equivalence. Assuming that this was one of the equivalences that our application needed maintained, the extraction would no longer be a valid transformation.

Standardized metadata format

The metadata format described previously was one created specifically for our hypothetical image processing application. We might decide to use a standardized metadata format, such as Dublin Core [33], instead. Figure 4.4 shows the transformation graph with a Dublin Core datatype added. The graph also contains two transformations for generating the Dublin Core metadata. The first simply converts the metadata that was already extracted into the simple metadata format into the Dublin Core encoding. Since both encodings are in XML, this might, for instance, be a simple XSLT transformation. Since the Dublin Core format contains many more metadata fields than our simple format does, this transformation will result in many of the Dublin Core fields being missing. Having noticed that the TIFF format includes support for many more metadata tags than we extracted for the simple metadata datatype, we have also included an extraction transformation that extracts any possible extra metadata from the TIFF, and outputs it directly into the Dublin Core format.

Both of these new transformations are useful for different reasons. The conversion transformation is useful because it will automatically support any image formats that we had written simple metadata extraction transformations for. The new extraction transformation is useful because we were able to

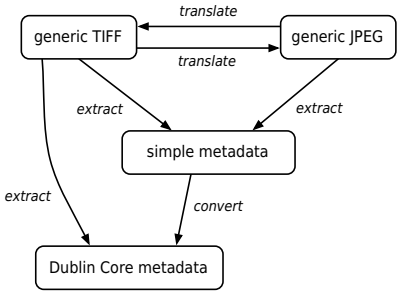


Figure 4.4: Adding a more complex, standardized metadata type to the graph

exploit knowledge specific to the TIFF format to get a more accurate transformation into the Dublin Core format. Our application will most likely want to use the new extraction transformation when extracting Dublin Core metadata from a TIFF image, overriding the other possibility of using our simple metadata format as an intermediary. The next section will introduce *properties*, which provide a means for stating these kinds of preferences.

Specialized image formats

As a final example, our image processing application might want to interface with an existing application, such as the Open Microscopy Environment[4] (OME) [104]. OME defines a robust metadata model for a particular domain of interest — in this case, high-resolution microscope images of biological samples. OME defines two specialized file formats for storing the images and associated metadata: OME-XML [46] and OME-TIFF [62]. Figure 4.5 shows the transformation graph with OME-XML and OME-TIFF datatypes added.

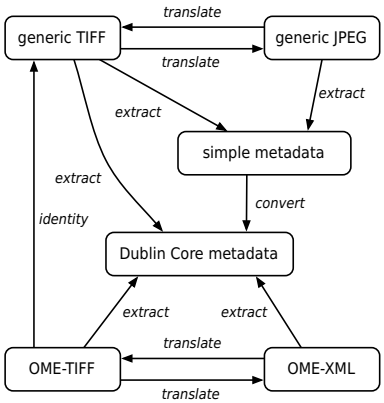


Figure 4.5: Adding a third-party image format to the graph

The graph also contains several transformations of interest. We include two translate transformations, just as with the generic TIFF and generic JPEG datatypes. These new translation transformations are implemented by code included in OME. We also provide transformations for extracting the Dublin Core metadata from the OME image formats. Alternatively, we could have extracted the OME metadata into the simple metadata format, and then relied on the existing conversion transformation

[4] http://www.openmicroscopy.org/

to get to Dublin Core, assuming that we did not care about the Dublin Core fields that are not defined in the simple metadata type.

Finally, we can also include an identity transformation that "converts" an OME-TIFF image into a generic TIFF. However, this transformation does not have to actually do anything, since every OME-TIFF is already a valid generic TIFF. This is another transformation that we cannot write an inverse for, since an arbitrary generic TIFF will not have the appropriate extra metadata to be an OME-TIFF, and we have no way to generate this extra metadata as part of a transformation.

4.1.2 Compound transformations

As mentioned earlier, by representing datatypes and atomic transformations as the nodes and edges of a directed graph, we can use directed paths in this graph to represent compound transformations between arbitrary types. With an appropriate selection of atomic transformations, the transformation graph will be fully connected, giving us the ability to translate between any pair of datatypes in the graph. Moreover, there are many efficient algorithms for finding directed paths in a graph, giving us a transformation discovery algorithm that is much more efficient than those used with other solutions to the data mismatch problem. This efficiency stems from the fact that we have abstracted away a large amount of unnecessary detail: instead of presenting detailed knowledge about the datatypes to the transformation discovery algorithm, we have encapsulated this information into the atomic transformations. A particular atomic transformation will incorporate knowledge of its input and output datatypes into its implementation; however, the high-level discovery algorithm does not need to know the details of how this translation is performed, and therefore does not need to know any details about the datatypes themselves. Instead, it only needs to know that some translation exists between those two types.

This pathfinding-based approach works because of how we have defined transformations in general, and how we have specified compound transformations in particular. As mentioned in Chapter 3, a "transformation" is a computation that maintains some data equivalence between its inputs and outputs. Further, this equivalence maintenance is exactly the property that defines a solution to the data mismatch problem. By implementing some atomic transformation and adding it to a transformation graph, the user has asserted that the atomic transformation maintains some equivalence of interest. Moreover, we have shown in Section 3.2.4 that equivalence maintenance is a property that can be easily extended to compound transformations: a compound transformation will maintain any equivalence that is maintained by all of its constituent atomic transformations. It is this inductive definition which allows us to use a pathfinding algorithm to search for compound transformations. As the algorithm executes, it extends the current directed path by adding another graph edge to it; an inductive proof step easily shows that the new path also maintains the data equivalence, assuming that the edge and the previous path also maintain it. As we will show in the next section, we can use this same mechanism to distinguish between different compound transformations that translate identical datatypes, as long as our optimality criteria can also be specified with inductive definitions; in graph algorithm terms, our pathfinding algorithm becomes a *shortest* pathfinding algorithm.

Even though we have described this technique in terms of datatypes and transformations, it is in fact more general. Our technique works because we can specify the problem that we are trying to solve using a property — in our case, the maintenance of data equivalences — that can be shown

inductively for the possible solutions. This induction-based search is implemented by a pathfinding algorithm. The technique is not limited to the data mismatch problem, though: it can be used to solve any problem that is defined by a property that can be shown inductively for the possible solutions.

4.1.3 Properties

There were several cases in the previous examples where there were multiple ways to transform data from one type to another. For instance, we could generate the Dublin Core metadata from a generic TIFF using the extraction transformation that we wrote directly for that purpose, or we could use the simple metadata datatype as an intermediary. In this case, it is obvious that the direct extraction into Dublin Core will almost always be the ideal solution. Another example involves the lossy transformation between a generic TIFF and a generic JPEG. In this case, it is not always clear what the correct solution is; if our application does not need full precision in the transformed pixel matrix, the lossy transformation is perfectly acceptable. If we need to ensure that the pixels are identical in all formats, we cannot allow the lossy transformation to be executed.

We support these different use cases using *properties*. Each datatype and transformation can include an arbitrary set of key-value pairs. The transformation discovery algorithms will then use a *weight provider* to calculate a numeric weight for each atomic transformation based on its properties. Different use cases are supported by using a different weight provider for each; the properties in the graph remain the same. The weight providers are able to use any information about the transformation to calculate its numeric weight, as long as this information can be expressed as a graph property.

We present several examples to show how this system of properties and weight providers allows us to support multiple transformation use cases using a single transformation graph. First, we examine the hypothetical transformation graph shown in Figure 4.6. This graph consists of datatypes and transformations from three separate applications that deal with postal addresses. The particulars of the postal address types differ between applications.

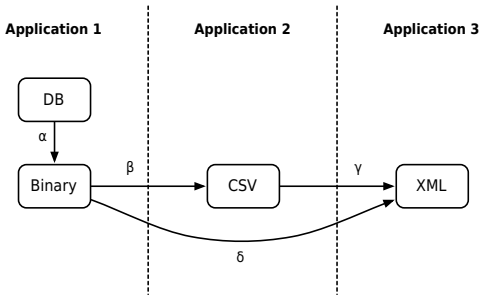


Figure 4.6: Example postal address transformation graph

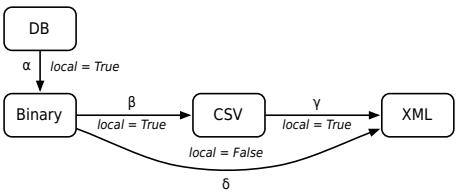
Application 1 stores addresses in a relational database, and has its own custom binary format for encoding addresses. Both of these datatypes contain six fields: Line1, Line2, City, Region, PostCode, and Country. Application 2 stores the addresses in Comma-Separated Value (CSV) text files. This application was written to work explicitly with American postal addresses, and therefore contains a different set of fields: Line1, Line2, City, State, and ZIP. Application 3 stores its addresses in XML; however, the designers of this application made the data model a bit too general, so it does

not contain specialized fields for city and region. Instead, the XML type contains the fields Line1, Line2, Line3, Line4, PostCode, and Country. In addition to these datatypes, the graph contains the transformations labeled α, β, γ, and δ.

Local vs. remote execution

In our first use case, we assume that some client (not necessarily any of the three original applications) wants to transform data from Application 1's database type to Application 3's XML type. There are two paths between these types: $\langle \alpha, \beta, \gamma \rangle$ and $\langle \alpha, \delta \rangle$. Different use cases might imply that either of these two paths is the optimal transformation. At first glance, it would seem that the shorter path would be best; most of the time, this will be the case. It could also be possible, however, that the δ transformation is very slow compared to β and γ. For instance, it might use some networked service to perform the translation. If β and γ both run locally, it could be faster to use the CSV type as an intermediary to prevent the overhead of network traffic and remote computation.

We can model this with an appropriate use of properties and weight providers. For instance, we could use a property named local as a Boolean flag to indicate whether a transformation executes locally. We could then declare that δ is a remote transformation, and that α, β, and γ are local, as shown in Figure 4.7. If the transformation client wants to ensure that only local transformations are used, they would use a weight provider that assigns a numeric weight of infinity to any transformation whose local property was *False*, effectively preventing the corresponding edge from being used by the pathfinding algorithm.



$$\chi(xform) = \textbf{if } xform.local \textbf{ then } 1 \textbf{ else } \infty$$

Figure 4.7: Using properties to prohibit remote transformations

In this situation, any compound transformation that contains any non-local atomic transformation will have an overall weight of infinity. If the pathfinding algorithm returns a shortest path of infinite weight, then we know that there cannot be a finite, and therefore strictly local, compound transformation; if there were, then it would have to have a smaller weight than the supposedly shortest path discovered by the pathfinding algorithm. If we want to truly prevent non-local transformations, we must verify that the discovered path has a finite weight before accepting it as a solution.

This weight provider also does not allow us to distinguish between multiple compound transformations that have a mixture of local and non-local transformations — the compound transformations will all have infinite, and therefore "equal", weights. Luckily, the pathfinding algorithm does not restrict us to scalar transformation weights; we can use any weight that forms a semiring, such as those for which we can define an addition operator and a total ordering.

This allows us to use a more sophisticated solution for this situation: we use two-element vectors instead of scalars as the transformation weights. We add weights together using piecewise vector

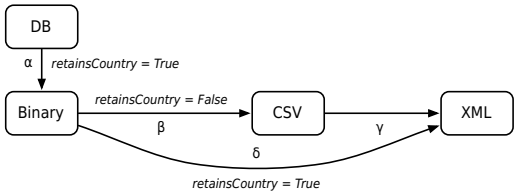
addition. We compare weights by examining the first elements first; we only compare the second elements if the first elements are equal. We can then use the following weight provider:

$$\chi(xform) = \textbf{if } xform.local \textbf{ then } [0, 1] \textbf{ else } [1, 0]$$

With these vector-based weights, the shortest path algorithm will correctly favor strictly local compound transformations, as before; now, though, it will also find the "most" local compound transformation when the possible solutions have a mixture of local and non-local atomic transformations. The first element of a compound transformation's weight vector will be the number of non-local atomic transformations; the second element will be the number of local ones. By comparing the first elements of the vector first, we ensure that any strictly local compound transformation, regardless of length, will be considered more optimal than any compound transformation containing a non-local element. Mixed compound transformations will be considered more optimal if they contain fewer non-local atomic transformations.

Partial transformations

Another scenario that we might want to model concerns the partiality of certain transformations. The CSV type does not contain a Country field, so if we use it as an intermediary type, as in the previous example, we will lose information in the transition from the database type to the XML type. Again, according to the needs of the client, this may or may not be important. If it is, we can model it using properties and weight providers. For instance, we can declare another Boolean property named retainsCountry and use it as shown in Figure 4.8. Then, as before, we could use a weight provider to give transformations whose retainsCountry property is *False* an edge weight of infinity, effectively ruling out the $\langle \alpha, \beta, \gamma \rangle$ path.



$$\chi(xform) = \textbf{if } xform.retainsCountry \textbf{ then } 1 \textbf{ else } \infty$$

Figure 4.8: Using properties to require the preservation of the Country field

As one last example, we can consider the inverse transformation: from the XML type to the database type. This will require corresponding inverse atomic transformations, which we will name α^{-1}, β^{-1}, γ^{-1} and δ^{-1}. As we mentioned above, the XML datatype was made overly general; it does not distinguish the city and region of an address, which are just included somewhere in the Line fields. In order to transform into the CSV or binary types, the γ^{-1} and δ^{-1} transformations must perform some analysis, or make a guess, as to which part of the address corresponds to the city and region name. Our transformation framework cannot make this analysis any easier, since the implementation of the transformations is opaque to the graph structure and the pathfinding algorithm used for discovery. However, since atomic transformations can be written in any existing transformation language, the transformation writer can use whatever tools are available to make this job easier.

4.1.4 Declaration patterns

Often, portions of a transformation graph are repetitive, with the same pattern of datatypes and atomic transformations appearing multiple times, with few differences between them. For instance, as shown in Figure 4.9, the transformation graph for an XML schema will often need (at least) the following:

- a datatype for the "raw" binary XML data,
- a datatype for the XML data stored in a DOM tree [63],
- a transformation for parsing the binary XML into a DOM tree, and
- a transformation for encoding a DOM tree into binary XML.

Ideally, we want to prevent the graph designer from having to repeat these definitions for each XML datatype.

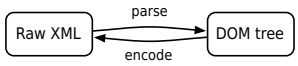


Figure 4.9: An oft-repeated pattern of datatypes and transformations

The solution is to allow *declaration patterns*. A declaration pattern is a set of datatypes and transformations that are parameterized. That is, the name of the datatype or transformation, or some of its properties, can be a parameter rather than a proper value. The pattern can then be *applied* later, with actual values specified for each parameter. This would allow us to define the XML pattern above, with parameters for the datatype and transformation names, and for the property specifying the datatype's XML schema definition. The graph designer would then apply this pattern for each XML datatype.

4.2 Graph definition language

In this section we define a language for constructing transformation graphs, and a file format that encodes this language. It will include support for all of the transformation graph elements defined in the previous section. Along with the denotational semantics presented later in the chapter, this will form the basis for implementing this transformation framework in a software library. We define the language using an EBNF grammar, and illustrate its use by providing several examples.

A graph declaration consists of a list of statements. The different kinds of statement can appear in any order. Some combinations of statements are inconsistent, and will therefore lead to an invalid transformation graph; in Section 4.3 we formally show when and how this can happen, and also show which transformation graph is produced from a sequence of statements when there are no inconsistencies.

$$\begin{aligned}\langle \textit{graph-decl} \rangle &::= \langle \textit{statement} \rangle^* \\ \langle \textit{statement} \rangle &::= \langle \textit{abbreviation} \rangle \mid \langle \textit{provides} \rangle \mid \langle \textit{requires} \rangle \mid \langle \textit{declaration} \rangle \mid \\ &\qquad \langle \textit{pattern-definition} \rangle \mid \langle \textit{pattern-application} \rangle\end{aligned}$$

The different kinds of allowed statements are described in more detail in the following sections.

4.2.1 Identifiers

⟨*id*⟩ ::= ⟨*namespaced-id*⟩ | ⟨*abbrev-id*⟩

Since the datatypes, transformations, and properties in a graph are all named, we need an identifier type. We are dealing with wide-ranging datatypes involving many applications written by different groups; therefore, the identifier scheme should make it easy to generate globally unique identifiers. We provide this feature by adapting the idea of a two-part, namespaced identifier from the XML standard [18]. However, whereas in XML the namespaces are URIs [14], we adopt a simpler notation more similar to Java or Python package names.

⟨*digit*⟩ ::= "0" .. "9"
⟨*alpha*⟩ ::= "a" .. "z" | "A" .. "Z"
⟨*alphanum*⟩ ::= ⟨*alpha*⟩ | ⟨*digit*⟩ | "_"

⟨*id-fragment*⟩ ::= ⟨*alpha*⟩ ⟨*alphanum*⟩*
⟨*local-id*⟩ ::= ⟨*id-fragment*⟩ ("." ⟨*id-fragment*⟩)*
⟨*namespaced-id*⟩ ::= ⟨*local-id*⟩ "::" ⟨*local-id*⟩

Every identifier in a transformation graph consists of two *identifier parts*: a *namespace* and a *local identifier*. These two parts are separated by a double-colon. Local identifiers are unique within the context of a single namespace. Each identifier part consists of one or more *identifier fragments*, separated by periods. An identifier fragment is a string of alphanumeric characters (including underscore), beginning with a letter.

As mentioned, the fragments that make up a namespace have semantic meaning similar to that of a Java package name. They are intended to start with the reverse of the Internet DNS name of the person or organization that "controls" the namespace. (Extra fragments after the reversed DNS name can be used to create multiple namespaces controlled by the same organization.) Namespaces beginning with `core` are reserved for core structures defined by the transformation language and libraries. Example namespaces include:

```
core
core.java
core.xml
com.example.addresses
com.example.employees
```

The fragments that make up a local identifier have no pre-determined semantics — they can be defined in whatever way is most appropriate to the data model or application domain. Example local identifiers include:

```
adapter
class
schema
generic.jpeg
extract.simple.jpeg.metadata
```

Taking together, example full identifiers include:

```
core::adapter
core.java::class
```

```
core.xml::schema
com.example.images::generic.jpeg
```

Namespace abbreviations

⟨*abbreviation*⟩ ::= "`namespace`" ⟨*local-id*⟩ "`:`" ⟨*local-id*⟩ "`;`"
⟨*abbrev-id*⟩ ::= ⟨*local-id*⟩ "`:`" ⟨*local-id*⟩

Within the scope of a single graph declaration file, an abbreviation can be given for a namespace for brevity. Namespace abbreviations have the same structure as any other identifier part, though they will usually consist of only a single identifier fragment. A namespace abbreviation is defined using the **`namespace`** statement:

```
namespace cx:  core.xml;
```

Once a **`namespace`** statement has been encountered, identifiers can be constructed using the new abbreviation. The identifier's namespace is replaced with the abbreviation, and the double colon is replaced with a single colon. For instance, with the above **`namespace`** statement in effect, the following two identifiers are equivalent:

```
core.xml::schema
cx:schema
```

It is not possible for namespaces and namespace abbreviations to clash, since the delimiter used signals whether the namespace is specified in full or via an abbreviation.

Reserved words

The following are reserved words; they cannot be used as namespaces, namespace abbreviations, or local identifiers.

`apply`, **`augment`**, **`datatype`**, **`define`**, **`from`**, **`namespace`**, **`pattern`**, **`provides`**, **`requires`**, **`to`**, **`transformation`**

4.2.2 Declarations

⟨*declaration*⟩ ::= ⟨*datatype-definition*⟩ | ⟨*datatype-augmentation*⟩ |
⟨*xform-definition*⟩ | ⟨*xform-augmentation*⟩

The bulk of a graph definition will consist of *declarations*. As their name implies, these statements declare the structures that make up the graph, and the properties of those structures.

Definitions and augmentations

All declarations fall into two categories: *definitions* and *augmentations*. Definitions are used to bring a structure into existence. It is an error for any structure to be defined more than once. Augmentations, on the other hand, provide new property values for an existing structure. It is an error to have an augmentation that refers to a non-existent structure.

Properties

Datatypes and atomic transformations can include *properties* that affect the pathfinding algorithms and the transformations themselves. They are modeled as a map or dictionary: every structure has a set of *property keys*, each of which map to a single *property value*. A property key is an identifier; different contexts will provide identifiers with specific meanings. (For instance, a transformation written in Java could have a core.java::class property that specifies which Java class implements the transformation.)

⟨*property-value*⟩ ::= ⟨*integer*⟩ | ⟨*string*⟩ | ⟨*id*⟩

Property values can be one of three things:

- An *integer literal*. This can be expressed in decimal or hexadecimal, following the usual C-like syntax:

  ⟨*hex-digit*⟩ ::= ⟨*digit*⟩ | “a” . . “f” | “A” . . “F”
  ⟨*integer*⟩ ::= “-”? ⟨*digit*⟩+ |
  “-”? (“0x” | “0X”) ⟨*hex-digit*⟩+

```
4       // decimal
0xFF03  // leading "0x" means hexidecimal
```

- A *string literal*. A string literal must be enclosed in double-quotes, and the standard C-like escape sequences are valid.

  ⟨*str-char*⟩ ::= “\n” | “\r” | “\f” | “\t” | “\b” |
  “\"” | “\’” | “\\” | (¬ {“\”, “"”})
  ⟨*string*⟩ ::= “"” ⟨*str-char*⟩* “"”

```
"string"
"string 'with single-quotes'"
"string \"with double-quotes\""
```

- An identifier.

⟨*property*⟩ ::= ⟨*id*⟩ ⟨*property-value*⟩ “;”
⟨*properties-clause*⟩ ::= “{” ⟨*property*⟩* “}”

Properties are defined in a *properties clause*, which is a list of property definitions, each ending in a semicolon, enclosed within curly braces. Each property in the clause is specified with its key first, its value second. For instance:

```
{
    // The value of the core::adapter property is another
    // identifier.
    core::adapter   core.java::adapter;

    // The value of the core.java::class property is a string.
    core.java::class   "com.example.TransformationImpl";
}
```

Datatypes

⟨*datatype-definition*⟩ ::= "define" "datatype" ⟨*id*⟩
(⟨*properties-clause*⟩ | ";")
⟨*datatype-augmentation*⟩ ::= "augment" "datatype" ⟨*id*⟩
(⟨*properties-clause*⟩ | ";")

Datatypes represent the nodes in a transformation graph. Datatype declarations specify the name of the datatype and an optional properties clause. If the properties clause is absent, then the declaration statement should be terminated with a semicolon.

```
define datatype demo::datatype1;

define datatype demo::datatype2
{
    demo::property1  "property value";
}

augment datatype demo::datatype1
{
    demo::property2  86;
}
```

Atomic transformations

⟨*xform-definition*⟩ ::= "define" "transformation" ⟨*id*⟩
"from" ⟨*id*⟩ "to" ⟨*id*⟩
(⟨*properties-clause*⟩ | ";")
⟨*xform-augmentation*⟩ ::= "augment" "transformation" ⟨*id*⟩
(⟨*properties-clause*⟩ | ";")

Atomic transformations represent the edges in a transformation graph. Atomic transformation declarations are very similar to datatype declarations. The only difference is that a transformation *definition* must specify a source and destination datatype. (A transformation augmentation should *not* mention the datatypes, since they will have already been specified by the corresponding definition.)

```
define transformation demo::xform1
    from demo::datatype1
    to   demo::datatype2;

define transformation demo::xform2
    from demo::datatype2
    to   demo::datatype1
{
    demo::xform.weight  86;
}

augment transformation demo::xform1
{
    demo::xform.description  "This is a nice transformation.";
}
```

4.2.3 File dependencies

Graph definitions are intended to be modular — different portions of a large data transformation graph can be specified in separate files. These files must then be merged together into a single graph before any transformation discovery can take place. One way to do this would be to require the client to provide an ordered list of files to include in the graph. This is not an ideal solution, however, especially as the number of files increases.

Instead, one can explicitly specify dependencies between files in the files themselves. This is done by naming the files with an identifier. The identifier of a file is specified by including a `provides` statement. The file should also contain a series of `requires` statements, each one specifying a file that must be loaded before this one. These required files should include, for instance, the definitions of any datatypes or transformations that are augmented in the file.

⟨*provides*⟩ ::= "`provides`" ⟨*id*⟩ "`;`"
⟨*requires*⟩ ::= "`requires`" ⟨*id*⟩ "`;`"

As an example, consider the following two files:

```
provides tests::types;

// Binary file format for a JPEG image.
define datatype tests::jpeg;
```

and

```
provides tests::java;

requires tests::types;

namespace java:  core.java;

// Specify the Java class for the binary JPEG image.
augment datatype tests::jpeg
{
    java:class  "java.nio.ByteBuffer";
}

// And create a new datatype to hold a Java-specific internal
// representation of an image.
define datatype tests::java.image
{
    java:class  "java.awt.Image";
}
```

This shows an example of using two files to separate a graph into its language-independent and language-dependent parts. The Java-specific part depends on the language-independent parts — for instance, in how it augments the existing tests::jpeg datatype. The file's "`requires tests::types`" statement expresses this dependency. This scheme for identifying files requires some way to locate a file given its name; however, this does not need to be a complex lookup service. Instead, we can specify a simple set of rules for deriving a file location from its name, such as using the namespace as a directory name, and the local identifier as a filename within that directory.

4.2.4 Declaration patterns

As mentioned in Section 4.1.4, *declaration patterns* provide a way to parameterize common patterns of datatypes and transformations. This can help avoid repetition in the graph's declaration.

To illustrate this, we can examine how the previously mentioned XML pattern could be declared, without patterns, for a hypothetical postal address schema:

```
namespace java:  core.java;
namespace xml:   core.xml;

define datatype demo::raw.address
{
    java:class    "java.nio.ByteBuffer";
    core::syntax  xml:raw;
    xml:schema    "http://example.com/schemas/address.xsd";
}

define datatype demo::dom.address
{
    java:class    "org.w3c.dom.Document";
    core::syntax  xml:dom;
    xml:schema    "http://example.com/schemas/address.xsd";
}

define transformation demo::parse.address
  from demo::raw.address
  to   demo::dom.address
{
    core::adapter  "com.example.xml.DocumentBuilderAdapter";
    xml:schema     "http://example.com/schemas/address.xsd";
}

define transformation demo::encode.address
  from demo::dom.address
  to   demo::raw.address
{
    core::adapter  "com.example.xml.TransformerAdapter";
    xml:schema     "http://example.com/schemas/address.xsd";
}
```

Certain properties in these declarations will have the same value every time. For instance, the raw datatype will always have a core.java::class property of *java.nio.ByteBuffer*. We can recast this example using a declaration pattern as follows:

```
namespace java:  core.java;
namespace xml:   core.xml;

define pattern xml:raw.and.dom
  $raw.datatype,
  $dom.datatype,
  $parse.xform,
  $encode.xform,
  $xml.schema
```

```
{
    define datatype $raw.datatype
    {
        java:class    "java.nio.ByteBuffer";
        core::syntax  xml:raw;
        xml:schema    $xml.schema;
    }

    define datatype $dom.datatype
    {
        java:class    "org.w3c.dom.Document";
        core::syntax  xml:dom;
        xml:schema    $xml.schema;
    }

    define transformation $parse.xform
       from $raw.datatype
       to   $dom.datatype
    {
        core::adapter  "com.example.xml.DocumentBuilderAdapter";
        xml:schema     $xml.schema;
    }

    define transformation $encode.xform
      from $dom.datatype
      to   $raw.datatype
    {
        core::adapter  "com.example.xml.TransformerAdapter";
        xml:schema     $xml.schema;
    }
}

apply pattern xml:raw.and.dom
{
    $raw.datatype:  demo::raw.address;
    $dom.datatype:  demo::dom.address;
    $parse.xform:   demo::parse.address;
    $encode.xform:  demo::encode.address;
    $xml.schema:    "http://example.com/schemas/address.xsd";
}
```

The key feature of a declaration pattern is that its declarations can be *parameterized*. A parameter is represented by a local identifier (i.e., one without a namespace), preceded by a "$".

⟨*parameter*⟩ ::= "$" ⟨*local-id*⟩

The declarations inside a pattern body have the same form as those outside, except a parameter can be used in place of any datatype or transformation name, or property value. (Property names cannot be parameterized.)

⟨*id-or-param*⟩ ::= ⟨*id*⟩ | ⟨*parameter*⟩

⟨*prop-value-or-param*⟩ ::= ⟨*prop-value*⟩ | ⟨*parameter*⟩
⟨*pattern-property*⟩ ::= ⟨*id*⟩ ⟨*prop-value-or-param*⟩
⟨*pattern-properties-clause*⟩ ::= "{" ⟨*pattern-property*⟩* "}"

⟨*pattern-dt-def*⟩ ::= "define" "datatype" ⟨*id-or-param*⟩
(⟨*pattern-properties-clause*⟩ | ";")
⟨*pattern-dt-aug*⟩ ::= "augment" "datatype" ⟨*id-or-param*⟩
(⟨*pattern-properties-clause*⟩ | ";")
⟨*pattern-xform-def*⟩ ::= "define" "transformation" ⟨*id-or-param*⟩
"from" ⟨*id-or-param*⟩ "to" ⟨*id-or-param*⟩
(⟨*pattern-properties-clause*⟩ | ";")
⟨*pattern-xform-aug*⟩ ::= "augment" "transformation" ⟨*id-or-param*⟩
(⟨*pattern-properties-clause*⟩ | ";")

A pattern is defined by providing a list of parameters and a pattern body. As with any declaration, an empty pattern body can be replaced with a semicolon, though empty patterns will not be useful in practice.

⟨*pattern-declaration*⟩ ::= ⟨*pattern-dt-def*⟩ | ⟨*pattern-dt-aug*⟩ |
⟨*pattern-xform-def*⟩ | ⟨*pattern-xform-aug*⟩
⟨*pattern-body*⟩ ::= "{" ⟨*pattern-declaration*⟩* "}"
⟨*param-list*⟩ ::= ϵ | ⟨*parameter*⟩ ("," ⟨*parameter*⟩)*
⟨*pattern-definition*⟩ ::= "define" "pattern" ⟨*id*⟩ ⟨*param-list*⟩
(⟨*pattern-body*⟩ | ";")

Patterns are applied by supplying a *binding*, which gives values for each of the parameters in the pattern definition. The parameters mentioned in the binding must exactly match those mentioned in the pattern definition. If a parameter is mentioned in the pattern's parameter list, but never used in the pattern body, it must still be provided a value when the pattern is applied.

⟨*param-assignment*⟩ ::= ⟨*parameter*⟩ ":" ⟨*property-value*⟩ ";"
⟨*binding*⟩ ::= "{" ⟨*param-assignment*⟩* "}"
⟨*pattern-application*⟩ ::= "apply" "pattern" ⟨*id*⟩
(⟨*binding*⟩ | ";")

Care must be taken with the declarations that appear in a pattern. For instance, the following is a perfectly valid pattern definition:

```
define pattern test::pattern
{
    define datatype test::datatype;
}
```

The datatype definition inside of the pattern is not parameterized, so applying this parameter will always try to define a new datatype named test::datatype. Since it is illegal to define a datatype more than once, it will also be illegal to apply this pattern more than once.

4.3 Formalization and semantics

In this section we present a formalization of our graph-based transformation framework. We use the formalization to show that the graph definition language is well-formed and consistent. Moreover,

it can be used with the graph definition file format as a basis for implementing the framework. This formalization is presented in three sections. First, we provide a definition for transformation graphs. Second, we define a simplified version of the graph definition language, and provide a denotational semantics for the language in terms of the declarations of the graph's datatypes and atomic transformations. Third, we add declaration patterns to the transformation language, and modify the denotational semantics accordingly.

4.3.1 Graph structure

First, we define the identifiers that are used to refer to the different elements of a graph. For the purposes of this formalization, the actual structure of an identifier is unimportant, and we use an opaque type.

Definition 4.1 *(Identifiers).*

$$[Identifier]$$

A *property collection* is a mapping between identifiers and *values*. A property value can be one of three things: an integer, a string, or an identifier. We do not consider how to represent integers or strings; like identifiers, they are treated as opaque objects.

Definition 4.2 *(Property collections).*

$$[Integer, String]$$
$$Value ::= Integer \mid String \mid Identifier$$
$$Properties == Identifier \pfun Value$$

The *Value* type is the first example of implicit free type constructors, one of our extensions to the Z notation described in Section 3.2. In the standard Z notation, as defined in [103], we would have to provide explicit constructors for the three kinds of *Value*; however, since it will always be clear from context (and formally unambiguous) which case is used in any situation, we elide these explicit constructors for brevity.

With these definitions in place, we can define a schema to represent a *datatype*. A datatype has a name, which is an identifier, and a property collection.

Definition 4.3 *(Datatypes).*

$$\begin{array}{|l} Datatype \\ \hline name : Identifier \\ properties : Properties \\ \hline \end{array}$$

To make it easier to add properties to a datatype's property collection, we define an augment function. Given a datatype and a property collection, this function returns an updated datatype schema with the same name and the combination of the two property collections.

Definition 4.4 *(Augmenting a datatype).*

$$\begin{array}{|l} augment : (Datatype \times Properties) \fun Datatype \\ \hline augment\,(dt, \theta) = \\ \quad \langle\!| name \mapsto dt.name, \\ \quad\ properties \mapsto dt.properties \oplus \theta |\!\rangle \end{array}$$

We provide a similar definition for *atomic transformations*. They are modeled much like the *Datatype* schema, except we must also record the transformation's source and destination datatypes.

Definition 4.5 *(Atomic transformations).*

$$
\begin{array}{|l}
\textit{AtomicTransformation} \\
\hline
\textit{name} : \textit{Identifier} \\
\textit{from} : \textit{Identifier} \\
\textit{to} : \textit{Identifier} \\
\textit{properties} : \textit{Properties} \\
\hline
\end{array}
$$

We also provide an augment function for atomic transformations, defined similarly to the augment function for datatypes.

Definition 4.6 *(Augmenting a transformation).*

$$
\begin{array}{|l}
\mathsf{augment} : (\textit{AtomicTransformation} \times \textit{Properties}) \to \textit{AtomicTransformation} \\
\hline
\mathsf{augment}\,(\textit{xf}, \theta) = \\
\quad \langle\!| \textit{name} \rightsquigarrow \textit{xf.name}, \\
\quad\ \textit{from} \rightsquigarrow \textit{xf.from}, \\
\quad\ \textit{to} \rightsquigarrow \textit{xf.to}, \\
\quad\ \textit{properties} \rightsquigarrow \textit{xf.properties} \oplus \theta \,|\!\rangle
\end{array}
$$

Lastly, we can define a *data transformation graph*, which is simply a set of datatypes and atomic transformations, each keyed by identifier. We include constraints to ensure that the name of each datatype (or transformation) matches its key.

Definition 4.7 *(Graphs).*

$$
\begin{array}{|l}
\textit{Graph} \\
\hline
\textit{datatypes} : \textit{Identifier} \pfun \textit{Datatype} \\
\textit{transformations} : \textit{Identifier} \pfun \textit{AtomicTransformation} \\
\hline
\forall\, \textit{id} : \mathrm{dom}\ \textit{datatypes} \bullet \textit{datatypes}\,(\textit{id}).\textit{name} = \textit{id} \\
\forall\, \textit{id} : \mathrm{dom}\ \textit{transformations} \bullet \textit{transformations}\,(\textit{id}).\textit{name} = \textit{id} \\
\hline
\end{array}
$$

For convenience, we provide a name for the empty graph (i.e., the one with no datatypes or transformations):

Definition 4.8 *(The empty graph).*

$$
\begin{array}{|l}
\varnothing_{\textit{Graph}} : \textit{Graph} \\
\hline
\varnothing_{\textit{Graph}} = \langle\!| \textit{datatypes} \rightsquigarrow \varnothing, \textit{transformations} \rightsquigarrow \varnothing \,|\!\rangle
\end{array}
$$

We will call a graph *well-formed* iff its transformations only refer to datatypes that exist in the graph.

Definition 4.9 *(Well-formedness of graphs).*

$$
\begin{array}{|l}
\mathsf{well\text{-}formed}\ \_ : \mathbb{P}\ \textit{Graph} \\
\hline
\mathsf{well\text{-}formed}\ G \Leftrightarrow \\
\quad \forall\, t : \mathrm{ran}\ \textit{G.transformations} \bullet \\
\qquad t.\textit{from} \in \mathrm{dom}\ \textit{G.datatypes} \land t.\textit{to} \in \mathrm{dom}\ \textit{G.datatypes}
\end{array}
$$

4.3.2 Graph declaration language

We can now formalize the language for constructing transformation graphs from Section 4.2. A graph is constructed via a *graph declaration*, which consists of a number of *declarations*. For simplicity, in this section we only include declaration statements in the language. We add pattern statements to the language in the next section. After defining the language, we provide a denotational semantics that specifies which particular transformation graph is defined by a sequence of statements in the graph definition language.

Language definition

There are two categories of declarations: *definitions* and *augmentations*. A definition creates a new object, whereas an augmentation adds new properties to an existing object. Each kind can refer to a datatype or an atomic transformation, yielding four possible declaration statements.

Definition 4.10 *(Declarations).*

$$
\begin{aligned}
\mathit{Declaration} ::= \;& \text{def type } \mathit{Identifier} \text{ as } \mathit{Properties} \\
\mid\; & \text{aug type } \mathit{Identifier} \text{ with } \mathit{Properties} \\
\mid\; & \text{def xform } \mathit{Identifier} \\
& \quad \text{from } \mathit{Identifier} \text{ to } \mathit{Identifier} \text{ as } \mathit{Properties} \\
\mid\; & \text{aug xform } \mathit{Identifier} \text{ with } \mathit{Properties}
\end{aligned}
$$

A *graph declaration* is then a sequence of declarations.

Definition 4.11 *(Graph declarations).*

$$\mathit{GraphDecl} == \text{seq } \mathit{Declaration}$$

It is possible for a declaration to be used erroneously; we will refer to the error condition as Ω. The result of a graph declaration will then be either a *Graph* or Ω, the combination of which we refer to as Graph_Ω.

Definition 4.12 *(Error conditions).*

$$\mathit{Graph}_\Omega ::= \mathit{Graph} \mid \Omega$$

Language semantics

With the language defined, we can now construct a denotational semantics for graph declarations. We do this by defining *interpretation functions* for each element of the language; taken together, these will uniquely define the graph produced by any graph declaration. We will use the $\mathcal{G}[\![\ ]\!]$ notation to refer to all of the interpretation functions defined in this section, since it should be obvious from context when we are interpreting a single declaration or an entire graph definition.

We start by defining how each kind of declaration modifies an existing graph. We will define this interpretation function with several cases.

Definition 4.13 *(Interpreting a declaration in terms of a graph).*

$$\mid\; \mathcal{G}[\![\_]\!]\_ : (\mathit{Declaration} \times \mathit{Graph}_\Omega) \to \mathit{Graph}_\Omega$$

The simplest case is when the graph is already in the error condition; in that case, the error propagates regardless of the declaration.

Case 4.13a *(Propagation of errors).*

$$\mathscr{G}[\![d : \mathit{Declaration}]\!]_{\Omega} = \Omega$$

A datatype definition adds a new *Datatype* schema instance to the graph's datatype set if there is not already a datatype with the same name. If there is an existing datatype, the result is the error condition.

Case 4.13b *(Datatype definitions).*

$$\begin{array}{l}
\mathscr{G}[\![\text{def type } \alpha \text{ as } \theta]\!]_G = \\
\quad \textbf{if } \alpha \notin \operatorname{dom} G.\mathit{datatypes} \textbf{ then} \\
\quad\quad \langle\!| \mathit{datatypes} \rightsquigarrow \\
\quad\quad\quad G.\mathit{datatypes} \cup \{\alpha \mapsto \langle\!| \mathit{name} \rightsquigarrow \alpha, \mathit{properties} \rightsquigarrow \theta |\!\rangle\}, \\
\quad\quad\; \mathit{transformations} \rightsquigarrow G.\mathit{transformations} |\!\rangle \\
\quad \textbf{else} \\
\quad\quad \Omega
\end{array}$$

A datatype augmentation adds new properties to the property collection of an existing datatype. We use the previously defined augment function to simplify this definition. If the specified datatype does not already exist, the result is the error condition.

Case 4.13c *(Datatype augmentations).*

$$\begin{array}{l}
\mathscr{G}[\![\text{aug type } \alpha \text{ with } \theta]\!]_G = \\
\quad \textbf{if } \alpha \in \operatorname{dom} G.\mathit{datatypes} \textbf{ then} \\
\quad\quad \langle\!| \mathit{datatypes} \rightsquigarrow \\
\quad\quad\quad G.\mathit{datatypes} \oplus \{\alpha \mapsto \text{augment}\,(G.\mathit{datatypes}\,(\alpha), \theta)\}, \\
\quad\quad\; \mathit{transformations} \rightsquigarrow G.\mathit{transformations} |\!\rangle \\
\quad \textbf{else} \\
\quad\quad \Omega
\end{array}$$

A transformation definition adds a new *AtomicTransformation* schema instance to the graph's transformation set if there is not already a transformation with the same name. Further, the graph must already contain the transformation's source and destination datatypes. If either of these constraints is not met, the result is the error condition.

Case 4.13d *(Transformation definitions).*

$$\begin{array}{l}
\mathscr{G}[\![\text{def xform } \alpha \text{ from } \beta \text{ to } \gamma \text{ as } \theta]\!]_G = \\
\quad \textbf{if } (\alpha \notin \operatorname{dom} G.\mathit{transformations}) \wedge \\
\quad\quad (\{\beta, \gamma\} \subseteq \operatorname{dom} G.\mathit{datatypes}) \textbf{ then} \\
\quad\quad \langle\!| \mathit{datatypes} \rightsquigarrow G.\mathit{datatypes}, \\
\quad\quad\; \mathit{transformations} \rightsquigarrow \\
\quad\quad\quad G.\mathit{transformations} \cup \{\alpha \mapsto \langle\!| \mathit{name} \rightsquigarrow \alpha, \\
\quad\quad\quad\quad \mathit{from} \rightsquigarrow \beta, \\
\quad\quad\quad\quad \mathit{to} \rightsquigarrow \gamma, \\
\quad\quad\quad\quad \mathit{properties} \rightsquigarrow \theta |\!\rangle\} |\!\rangle \\
\quad \textbf{else} \\
\quad\quad \Omega
\end{array}$$

A transformation augmentation adds new properties to the property collection of an existing transformation. We use the previously defined augment function to simplify this definition. If the specified transformation does not already exist, the result is the error condition.

Case 4.13e *(Transformation augmentations).*

$$\begin{array}{l}\mathcal{G}[\![\text{aug xform } \alpha \text{ with } \theta]\!]_G = \\ \quad \textbf{if } \alpha \in \text{dom } G.\mathit{transformations} \textbf{ then} \\ \quad\quad (\!| \mathit{datatypes} \rightsquigarrow G.\mathit{datatypes}, \\ \quad\quad\ \mathit{transformations} \rightsquigarrow \\ \quad\quad\quad G.\mathit{transformations} \oplus \{\alpha \mapsto \text{augment } (G.\mathit{transformations}\,(\alpha), \theta)\} |\!) \\ \quad \textbf{else} \\ \quad\quad \Omega\end{array}$$

This completes the interpretation function for a single declaration. We can provide a similar interpretation function for graph declarations, showing how a sequence of declarations modifies an existing graph.

The interpretation function for graph declarations is defined in two parts. An empty sequence obviously leaves the graph unchanged, regardless of whether the graph was in an error condition. For a non-empty sequence, we interpret the sequence's first declaration in terms of the original graph. We then interpret the remainder of the sequence in terms of the new graph. This allows us to inductively step through the sequence of declarations, interpreting each one in turn.

Definition 4.14 *(Interpreting a graph declaration in terms of a graph).*

$$\begin{array}{|l} \mathcal{G}[\![\_]\!]\_ : (\mathit{GraphDecl} \times \mathit{Graph}_\Omega) \to \mathit{Graph}_\Omega \\ \hline \mathcal{G}[\![\langle\rangle]\!]_G = G \\ \mathcal{G}[\![\langle d \rangle \frown \mathit{rest}]\!]_G = \mathcal{G}[\![\mathit{rest}]\!]_{\mathcal{G}[\![d]\!]_G} \end{array}$$

We can show that the interpretation of graph declarations propagates errors, just like the interpretation of declarations does.

Theorem 4.15 (Propagation of errors in graph declarations). *For any graph declaration* gd, $\mathcal{G}[\![gd]\!]_\Omega = \Omega$.

Proof. We prove this inductively on the structure of the declaration sequence.

Base case. Let $gd = \langle\rangle$. Definition 4.14 trivially shows that $\mathcal{G}[\![gd]\!]_\Omega = \Omega$.

Inductive case. Let $gd = \langle d \rangle \frown \mathit{rest}$ and assume that $\mathcal{G}[\![\mathit{rest}]\!]_\Omega = \Omega$. By Definition 4.14 we know that

$$\mathcal{G}[\![gd]\!]_\Omega = \mathcal{G}[\![\mathit{rest}]\!]_{\mathcal{G}[\![d]\!]_\Omega}$$

Case 4.13a tells us that errors propagate through an individual declaration, so that $\mathcal{G}[\![d]\!]_\Omega = \Omega$. By substitution,

$$\mathcal{G}[\![gd]\!]_\Omega = \mathcal{G}[\![\mathit{rest}]\!]_\Omega$$

Finally, by the inductive assumption, we know that $\mathcal{G}[\![\mathit{rest}]\!]_\Omega = \Omega$. By substitution,

$$\mathcal{G}[\![gd]\!]_\Omega = \Omega$$

□

Definition 4.14 provides a semantics for graph declarations in terms of an existing graph. Finally, we can provide a new interpretation function that provides a semantics for graph declarations on their own — we simply start with an empty graph.

Definition 4.16 *(Interpreting a graph declaration).*

$$\frac{\mathcal{G}[\![\_]\!] : \mathit{GraphDecl} \to \mathit{Graph}_\Omega}{\mathcal{G}[\![gd]\!] = \mathcal{G}[\![gd]\!]_{\varnothing_{\mathit{Graph}}}}$$

We can say that this interpretation function is "useful" by showing that any graph declaration evaluates to either a well-formed graph or the error condition.

Theorem 4.17 (No errors implies well-formedness). *For every graph declaration gd, if $\mathcal{G}[\![gd]\!] \neq \Omega$, then $\mathcal{G}[\![gd]\!]$ is a well-formed graph.*

Proof. We prove this by induction on the sequence of declarations in gd. Any errors that arise are propagated through to the end by Case 4.13a. By Definition 4.16, we start with the empty graph, which is trivially well-formed. We must therefore show that each declaration, if given a well-formed graph, must produce either a well-formed graph or Ω.

First, we must show that the constructed graph satisfies all of the constraints defined in the *Graph* schema. Specifically, since we have modeled the *datatypes* and *transformations* elements as partial functions, we must ensure that any declaration that adds to these mappings maintains functionality. In both definitions (Cases 4.13b and 4.13d), we first check that there is not an existing object with the same name, returning Ω otherwise. The augmentations (Cases 4.13c and 4.13e) do not add to the mappings, so functionality cannot be violated. We therefore correctly ensure functionality in all cases.

Finally, we must also show that the graph is well-formed, by showing that the source and destination datatype of each transformation exists. Case 4.13d ensures that this is true when a transformation is defined, returning Ω otherwise. Since no other declarations create transformations, and no declarations remove datatypes, we can be sure that this well-formedness propagates through any remaining declarations. □

Another way to show that the language is useful is to show that it is *complete* — that the language provides a way to construct every well-formed graph.

Theorem 4.18 (Constructibility of well-formed graphs). *Every well-formed graph G has at least one graph declaration gd that constructs it:*

$$\forall\, G : \mathit{Graph} \mid \mathit{well\text{-}formed}\ G \bullet \exists\, gd : \mathit{GraphDecl} \bullet \mathcal{G}[\![gd]\!] = G$$

Proof. We prove this by construction. We create a graph declaration that starts with a sequence of def type declarations, one for each of the datatypes in the graph. Each def type declaration contains the full property collection of the corresponding datatype. The def type declarations can appear in any order.

Following the def type declarations, the graph declaration contains a sequence of def xform declarations, one for each of the transformations in the graph. Each def xform declaration contains the

full property collection of the corresponding transformation. The def xform declarations can appear in any order.

Assuming that the graph declaration does not inadvertently interpret to Ω, it obviously interprets to the graph G. There are two possible ways that the graph declaration could interpret to Ω. First, it might declare a particular datatype or transformation twice. This cannot happen, however, since we have added exactly one def type declaration for each distinct datatype in the graph G, and exactly one def xform declaration for each distinct transformation. None of these definitions repeat. Second, we might try to create a transformation with a nonexistent source or destination datatype. However, we know that since the graph G is well-formed, the transformations must have existing source and destination datatypes. Further, since we are careful to create *all* of the datatypes before creating *any* of the transformations, we are sure that the def xform declarations cannot cause false errors. Since $\mathscr{G}[\![gd]\!]$ cannot be Ω, it must equal G. □

4.3.3 Declaration patterns

In this section we augment the transformation language to include the declaration patterns described in Section 4.1.4. This requires a modification to the denotational semantics of the language, as well. As before, we use a modified form of the Z notation, allowing implicit free type constructors. These are used, for instance, in the definitions of $Value_P$ (Definition 4.20) and $Identifier_P$ (Definition 4.21).

Language definition

Declaration patterns are useful because they are *parameterized*. Like identifiers, we do not concern ourselves with the low-level encoding of a parameter, and treat it as an abstract type. A *binding* is a mapping of parameters to concrete values.

Definition 4.19 *(Parameters and bindings).*

$$[Parameter]$$
$$Binding == Parameter \pfun Value$$

Parts of the new transformation language will accept a parameter name when a value is expected; we call this combination $Value_P$ to distinguish it from when only a value is allowed.

Definition 4.20 *(Parameterized values).*

$$Value_P ::= Value \mid Parameter$$

Similarly, parts of the new transformation language will accept a parameter name when an identifier is expected; we call this combination $Identifier_P$ to distinguish it from when only an identifier is allowed.

Definition 4.21 *(Parameterized identifiers).*

$$Identifier_P ::= Identifier \mid Parameter$$

Finally, parts of the new transformation language will accept a property collection whose values can be parameters; we call this combination $Properties_P$ to distinguish it from a property collection

that can only contain concrete values. Note that the *names* of the properties cannot be parameterized; they must be explicit identifiers.

Definition 4.22 *(Parameterized properties).*

$$Properties_P == Identifier \pfun Value_P$$

A *parameterized declaration* can accept a parameter for the name of the object to define or augment, and can accept parameters for the values in its property collection.

Definition 4.23 *(Parameterized declarations).*

$$\begin{aligned} Declaration_P ::= & \ \mathsf{def_P\ type}\ Identifier_P\ \mathsf{as}\ Properties_P \\ | & \ \mathsf{aug_P\ type}\ Identifier_P\ \mathsf{with}\ Properties_P \\ | & \ \mathsf{def_P\ xform}\ Identifier_P \\ & \quad \mathsf{from}\ Identifier_P\ \mathsf{to}\ Identifier_P\ \mathsf{as}\ Properties_P \\ | & \ \mathsf{aug_P\ xform}\ Identifier_P\ \mathsf{with}\ Properties_P \end{aligned}$$

A *parameterized graph declaration* is then simply a sequence of parameterized declarations.

Definition 4.24 *(Parameterized graph declarations).*

$$GraphDecl_P == \mathsf{seq}\ Declaration_P$$

A *declaration pattern* has a name, which is an identifier, and a body, which is a parameterized graph declaration.

Definition 4.25 *(Declaration patterns).*

$$\begin{array}{|l} Pattern \\ \hline name : Identifier \\ body : GraphDecl_P \\ \hline \end{array}$$

With the previous definitions in place, we can add two new statements to the transformation language. The first defines a new pattern; the second applies an existing pattern based on a binding of parameters to values. All of the previous (unparameterized) declarations are still allowed. (To distinguish them, we use the term *statement* to refer to elements of the pattern-aware language and the term *declaration* for elements of the pattern-free language.)

Definition 4.26 *(Statements).*

$$\begin{aligned} Statement ::= & \ Declaration \\ | & \ \mathsf{def\ pattern}\ Identifier\ \mathsf{as}\ GraphDecl_P \\ | & \ \mathsf{apply\ pattern}\ Identifier\ \mathsf{with}\ Binding \end{aligned}$$

A *patterned graph declaration* is then a sequence of statements.

Definition 4.27 *(Patterned graph declaration).*

$$PatternedGraphDecl == \mathsf{seq}\ Statement$$

Language semantics

In the previous section, the semantics of the transformation language defined how a transformation graph was constructed from a graph declaration. We must now augment the semantics to show how

a graph can be constructed from a patterned graph declaration. We used the $\mathscr{G}[\![\ ]\!]$ notation to refer to the interpretation functions for the pattern-free language. We will define a new set of interpretation functions, called $\mathscr{P}[\![\ ]\!]$, for the pattern-aware language.

We start by defining *evaluation functions* for each of the parameterized clauses in the language. We will use the $[\ ]$ notation to refer to all of the evaluation functions defined in this section, since it should be obvious from context which is referred to.

Given a binding, we can *evaluate* a parameterized value. The evaluation of a concrete value is the concrete value. A parameter is evaluated by looking for it in the binding: if it is bound to some value, the evaluation is that value; if not, the evaluation is the error condition Ω.

Definition 4.28 *(Evaluating a parameterized value).*

$$\mathit{Value}_\Omega ::= \mathit{Value} \mid \Omega$$

$$\begin{array}{|l} [\_]\_ : (\mathit{Value}_P \times \mathit{Binding}) \to \mathit{Value}_\Omega \\ \hline [\alpha : \mathit{Identifier}]_\phi = \alpha \\ [i : \mathit{Integer}]_\phi = i \\ [s : \mathit{String}]_\phi = s \\ [p : \mathit{Parameter}]_\phi = \\ \quad \textbf{if } p \in \mathrm{dom}\ \phi \textbf{ then } \phi(p) \textbf{ else } \Omega \end{array}$$

Given a binding, we can *evaluate* a parameterized identifier. The evaluation of an identifier is the identifier. A parameter is evaluated by looking for it in the binding: if it is bound to some identifier (but not to any other kind of value), the evaluation is that identifier; if not, the evaluation is the error condition Ω.

Definition 4.29 *(Evaluating a parameterized identifier).*

$$\mathit{Identifier}_\Omega ::= \mathit{Identifier} \mid \Omega$$

$$\begin{array}{|l} [\_]\_ : (\mathit{Identifier}_P \times \mathit{Binding}) \to \mathit{Identifier}_\Omega \\ \hline [\alpha : \mathit{Identifier}]_\phi = \alpha \\ [p : \mathit{Parameter}]_\phi = \\ \quad \textbf{if } (p \in \mathrm{dom}\ \phi) \wedge (\phi(p) \in \mathit{Identifier}) \textbf{ then } \phi(p) \textbf{ else } \Omega \end{array}$$

Given a binding, we can *evaluate* a parameterized property collection. We evaluate each parameterized value in the property collection in turn; if any evaluate to Ω, the property collection as a whole evaluates to Ω. Otherwise, each value is replaced by its evaluation.

Definition 4.30 *(Evaluating a parameterized property collection).*

$$\mathit{Properties}_\Omega ::= \mathit{Properties} \mid \Omega$$

$$\begin{array}{|l} [\_]\_ : (\mathit{Properties}_P \times \mathit{Binding}) \to \mathit{Properties}_\Omega \\ \hline [\theta_P]_\phi = \\ \quad \textbf{if } \exists\, v_P : \mathrm{ran}\ \theta_P \bullet [v_P]_\phi = \Omega \textbf{ then } \Omega \\ \quad \textbf{else } \{\, (\alpha \mapsto v_P) : \theta_P \bullet \alpha \mapsto [v_P]_\phi \,\} \end{array}$$

Next, given a binding, we can *evaluate* a parameterized declaration. We provide separate definitions for each kind of declaration, but the rationale for each is identical: if any part of the declaration evaluates to Ω, then the declaration does as well; otherwise, each part is replaced by its evaluation.

Definition 4.31 *(Evaluating a parameterized declaration).*

$$Declaration_\Omega ::= Declaration \mid \Omega$$

$$\mid \ [\_]\_ : (Declaration_P \times Binding) \to Declaration_\Omega$$

Case 4.31a *(Datatype definitions).*

$$\begin{array}{l}[\mathsf{def_P\ type}\ \alpha_P\ \mathsf{as}\ \theta_P]_\phi = \\ \quad \textbf{if}\ ([\alpha_P]_\phi = \Omega) \lor ([\theta_P]_\phi = \Omega)\ \textbf{then}\ \Omega \\ \quad \textbf{else}\ \mathsf{def\ type}\ [\alpha_P]_\phi\ \mathsf{as}\ [\theta_P]_\phi\end{array}$$

Case 4.31b *(Datatype augmentations).*

$$\begin{array}{l}[\mathsf{aug_P\ type}\ \alpha_P\ \mathsf{with}\ \theta_P]_\phi = \\ \quad \textbf{if}\ ([\alpha_P]_\phi = \Omega) \lor ([\theta_P]_\phi = \Omega)\ \textbf{then}\ \Omega \\ \quad \textbf{else}\ \mathsf{aug\ type}\ [\alpha_P]_\phi\ \mathsf{with}\ [\theta_P]_\phi\end{array}$$

Case 4.31c *(Transformation definitions).*

$$\begin{array}{l}[\mathsf{def_P\ xform}\ \alpha_P\ \mathsf{from}\ \beta_P\ \mathsf{to}\ \gamma_P\ \mathsf{as}\ \theta_P]_\phi = \\ \quad \textbf{if}\ ([\alpha_P]_\phi = \Omega) \lor ([\beta_P]_\phi = \Omega) \lor \\ \quad\quad ([\gamma_P]_\phi = \Omega) \lor ([\theta_P]_\phi = \Omega)\ \textbf{then}\ \Omega \\ \quad \textbf{else}\ \mathsf{def\ xform}\ [\alpha_P]_\phi\ \mathsf{from}\ [\beta_P]_\phi\ \mathsf{to}\ [\gamma_P]_\phi\ \mathsf{as}\ [\theta_P]_\phi\end{array}$$

Case 4.31d *(Transformation augmentations).*

$$\begin{array}{l}[\mathsf{aug_P\ xform}\ \alpha_P\ \mathsf{with}\ \theta_P]_\phi = \\ \quad \textbf{if}\ ([\alpha_P]_\phi = \Omega) \lor ([\theta_P]_\phi = \Omega)\ \textbf{then}\ \Omega \\ \quad \textbf{else}\ \mathsf{aug\ xform}\ [\alpha_P]_\phi\ \mathsf{with}\ [\theta_P]_\phi\end{array}$$

Finally, given a binding, we can *evaluate* a parameterized graph declaration. We evaluate each of the sequence's declarations in turn; if any evaluate to Ω, then the parameterized graph declaration does as well; otherwise, each declaration is replaced by its evaluation.

Definition 4.32 *(Evaluating a parameterized graph declaration).*

$$GraphDecl_\Omega ::= GraphDecl \mid \Omega$$

$$\mid \ [\_]\_ : (GraphDecl_P \times Binding) \to GraphDecl_\Omega$$

$$\begin{array}{l}[gd]_\phi = \\ \quad \textbf{if}\ (\exists\ d : \mathrm{ran}\ gd \bullet [d]_\phi = \Omega)\ \textbf{then}\ \Omega \\ \quad \textbf{else}\ [\_]_\phi \circ gd\end{array}$$

With these evaluation functions in place, we can proceed with defining the $\mathscr{P}[\![\ ]\!]$ interpretation functions. In the previous section, the interpretation functions directly yielded graphs. With the new language features, however, we must keep track of extra state, and therefore require a separate state schema.

The *state* of an interpretation is defined by the graph constructed so far, and the set of patterns that have been defined. An interpretation can also be in an erroneous state, which we signify by Ω. We will use subscripts when it is important to distinguish the erroneous state (Ω_S) from the erroneous graph (Ω_G).

Definition 4.33 *(States).*

$$
\begin{array}{|l}
\textit{State} \\
\hline
\textit{graph} : \textit{Graph} \\
\textit{patterns} : \textit{Identifier} \pfun \textit{Pattern} \\
\hline
\forall\, \textit{id} : \operatorname{dom} \textit{patterns} \bullet \textit{patterns}\,(\textit{id}).\textit{name} = \textit{id} \\
\hline
\end{array}
$$

$$\textit{State}_\Omega ::= \textit{State} \mid \Omega$$

We say that a state and a graph are *consistent* iff they are either both erroneous, or the graphs are identical.

Definition 4.34 *(Consistency of states and graphs).*

$$S \text{ and } G \text{ are consistent} \Leftrightarrow (S = \Omega_S \land G = \Omega_G) \lor (S.\textit{graph} = G)$$

For convenience, we provide a name for the empty state (i.e., the one with an empty graph and no patterns):

Definition 4.35 *(The empty state).*

$$
\begin{array}{|l}
\varnothing_{\textit{State}} : \textit{State} \\
\hline
\varnothing_{\textit{State}} = \langle\!| \textit{graph} \rightsquigarrow \varnothing_{\textit{Graph}}, \textit{patterns} \rightsquigarrow \varnothing |\!\rangle \\
\end{array}
$$

We can now proceed with defining the interpretation functions. We start by defining how each kind of statement modifies an existing state. As before, we will define this function with several cases.

Definition 4.36 *(Interpreting a statement in terms of a state).*

$$\mathscr{P}[\![\_]\!]\_ : (\textit{Statement} \times \textit{State}_\Omega) \to \textit{State}_\Omega$$

The simplest case is when we are already in the error condition; in that case, the error propagates regardless of the statement.

Case 4.36a *(Propagation of errors).*

$$\mathscr{P}[\![\textit{stmt} : \textit{Statement}]\!]_\Omega = \Omega$$

An unparameterized declaration is interpreted in terms of the current graph using $\mathscr{G}[\![\ ]\!]$. If the declaration interprets to the erroneous graph, then the corresponding statement interprets to the erroneous state.

Case 4.36b *(Unparameterized declarations).*

$$
\begin{array}{l}
\mathscr{P}[\![d : \textit{Declaration}]\!]_S = \\
\quad \textbf{if } \mathscr{G}[\![d]\!]_{S.\textit{graph}} \neq \Omega_G \textbf{ then} \\
\quad\quad \langle\!| \textit{graph} \rightsquigarrow \mathscr{G}[\![d]\!]_{S.\textit{graph}}, \\
\quad\quad\quad \textit{patterns} \rightsquigarrow S.\textit{patterns} |\!\rangle \\
\quad \textbf{else} \\
\quad\quad \Omega_S
\end{array}
$$

A pattern definition adds a new *Pattern* schema instance to the state if there is not already a pattern with the same name. If there is an existing pattern, the result is the error condition.

Case 4.36c *(Pattern definitions).*

$$\begin{array}{l}\mathscr{P}[\![\text{def pattern } \alpha \text{ as } gd_P]\!]_S = \\ \quad \textbf{if } (\alpha \notin \text{dom } S.patterns) \textbf{ then} \\ \quad\quad \langle\!| graph \rightsquigarrow S.graph, \\ \quad\quad\quad patterns \rightsquigarrow S.patterns \cup \{\alpha \mapsto \langle\!| name \rightsquigarrow \alpha, body \rightsquigarrow gd_P |\!\rangle\} |\!\rangle \\ \quad \textbf{else} \\ \quad\quad \Omega\end{array}$$

A pattern can be applied with a binding of parameters to values. First, we evaluate the pattern's body in terms of the given binding; this substitutes each parameter's actual value into the pattern's declarations. We then interpret this substituted pattern body in terms of the current graph. Since there are no longer any parameters in the pattern body, this final step is performed using the existing $\mathscr{G}[\![\ ]\!]$ function.

There are three ways that a pattern application can interpret to Ω. First, the pattern might not have been defined. Second, the pattern's body might evaluate to Ω, signifying that some of the parameters mentioned in the pattern body were not assigned values by the binding. Third, the evaluated body might interpret to Ω; this might occur, for instance, if, after successfully substituting values for all of the parameters, the pattern body tries to define a datatype or transformation that already exists.

Case 4.36d *(Pattern applications).*

$$\begin{array}{l}\mathscr{P}[\![\text{apply pattern } \alpha \text{ with } \phi]\!]_S = \\ \quad \textbf{if } (\alpha \in \text{dom } S.patterns) \wedge ([\![S.patterns(\alpha).body]\!]_\phi \neq \Omega) \wedge \\ \quad\quad (\mathscr{G}[\![[\![S.patterns(\alpha).body]\!]_\phi]\!]_{S.graph}) \neq \Omega \textbf{ then} \\ \quad\quad \langle\!| graph \rightsquigarrow \mathscr{G}[\![[\![S.patterns(\alpha).body]\!]_\phi]\!]_{S.graph}, \\ \quad\quad\quad patterns \rightsquigarrow S.patterns |\!\rangle \\ \quad \textbf{else} \\ \quad\quad \Omega\end{array}$$

This completes the interpretation function for a single statement. We can provide a similar interpretation function for entire patterned graph declarations, showing how the sequence of statements modifies an existing state.

The interpretation function for patterned graph declarations is defined in two parts. An empty sequence obviously leaves the state unchanged, regardless of whether the state was in an error condition. For a non-empty sequence, we interpret the sequence's first statement in terms of the original state. We then interpret the remainder of the sequence in terms of the new state. This allows us to inductively step through the sequence of statements, interpreting each one in turn.

Definition 4.37 *(Interpreting a patterned graph declaration in terms of a state).*

$$\begin{array}{|l}\mathscr{P}[\![\_]\!]\_ : (PatternedGraphDecl \times State_\Omega) \to State_\Omega \\ \hline \mathscr{P}[\![\langle\rangle]\!]_S = S \\ \mathscr{P}[\![\langle stmt\rangle \frown rest]\!]_S = \mathscr{P}[\![rest]\!]_{\mathscr{P}[\![stmt]\!]_S}\end{array}$$

We can show that the interpretation of patterned graph declarations propagates errors, just like the interpretation of statements does.

Theorem 4.38 (Propagation of errors in patterned graph declarations). *For any patterned graph declaration* gd_P, $\mathscr{P}[\![gd_P]\!]_\Omega = \Omega$.

Proof. We prove this inductively on the structure of the statement sequence.

Base case. Let $gd_P = \langle\rangle$. Definition 4.37 trivially shows that $\mathscr{P}[\![gd_P]\!]_\Omega = \Omega$.

Inductive case. Let $gd_P = \langle stmt \rangle \frown rest$ and assume that $\mathscr{P}[\![rest]\!]_\Omega = \Omega$. By Definition 4.37 we know that

$$\mathscr{P}[\![gd_P]\!]_\Omega = \mathscr{P}[\![rest]\!]_{\mathscr{P}[\![stmt]\!]_\Omega}$$

Case 4.36a tells us that errors propagate through individual statements, so that $\mathscr{P}[\![stmt]\!]_\Omega = \Omega$. By substitution,

$$\mathscr{P}[\![gd_P]\!]_\Omega = \mathscr{P}[\![rest]\!]_\Omega$$

Finally, by the inductive assumption, we know that $\mathscr{P}[\![rest]\!]_\Omega = \Omega$. By substitution,

$$\mathscr{P}[\![gd_P]\!]_\Omega = \Omega$$

□

Definition 4.37 provides a semantics for patterned graph declarations in terms of an existing state. Finally, we can provide a new interpretation function that provides a semantics for patterned graph declarations on their own — we simply start with an empty state.

Definition 4.39 *(Interpreting a patterned graph declaration).*

$$\frac{\mathscr{P}[\![\_]\!] : PatternedGraphDecl \to State_\Omega}{\mathscr{P}[\![gd_P]\!] = \mathscr{P}[\![gd_P]\!]_{\varnothing_{State}}}$$

This concludes the denotational semantics for the extended transformation language. To prove its correctness, we must show two things. First, we must show that the semantics of the new language is consistent with the semantics of the old. Second, we must show that the new semantics is still useful, by proving that any patterned graph declaration that interprets without errors yields a well-formed graph.

We prove consistency by showing that a sequence of unparameterized declarations, which is a valid graph declaration in both languages, yields an equivalent interpretation in each. We first prove this in terms of an already consistent state and graph, and then prove it in the more general case.

Lemma 4.40 (Consistency is maintained by interpretation). *Given an unpatterned graph declaration* gd, *if* S_0 *and* G_0 *are consistent according to Definition 4.34, then* $\mathscr{P}[\![gd]\!]_{S_0}$ *and* $\mathscr{G}[\![gd]\!]_{G_0}$ *are also consistent.*

Proof. There are two ways that S_0 and G_0 can be consistent. If S_0 and G_0 are Ω, then $\mathscr{P}[\![gd]\!]_{S_0}$ and $\mathscr{G}[\![gd]\!]_{G_0}$ are also Ω by Theorems 4.38 and 4.15, respectively. They are therefore trivially consistent.

If S_0 and G_0 are not Ω, we must prove this by induction on the structure of the sequence of declarations.

Base case. Let $gd = \langle\rangle$. Definition 4.14 tells us that the empty sequence does not affect the interpreted graph:

$$\mathscr{G}[\![gd]\!]_{G_0} = \mathscr{G}[\![\langle\rangle]\!]_{G_0} = G_0$$

Similarly, Definition 4.37 tells us that the empty sequence does not affect the interpreted state:

$$\mathscr{P}[\![gd]\!]_{S_0} = \mathscr{P}[\![\langle\rangle]\!]_{S_0} = S_0$$

Since we already know that S_0 and G_0 are consistent, we can see by substitution that $\mathscr{P}[\![gd]\!]_{S_0}$ and $\mathscr{G}[\![gd]\!]_{G_0}$ are consistent as well.

We must consider the inductive case in two parts, depending on whether the first declaration in the sequence interprets to Ω.

Inductive case 1. Let $gd = \langle d \rangle \frown rest$. In this case, we assume that the first declaration in the sequence interprets to an error, so we also let $\mathscr{G}[\![d]\!]_{G_0} = \Omega_G$. We would like to show that both interpretations of gd are Ω, and therefore trivially consistent.

Definition 4.14 tells us that

$$\mathscr{G}[\![gd]\!]_{G_0} = \mathscr{G}[\![rest]\!]_{\mathscr{G}[\![d]\!]_{G_0}}$$

One of our assumptions for this case is that $\mathscr{G}[\![d]\!]_{G_0} = \Omega_G$, so by substitution,

$$\mathscr{G}[\![gd]\!]_{G_0} = \mathscr{G}[\![rest]\!]_{\Omega_G}$$

Theorem 4.15 tells us that Ω_G propagates through the $\mathscr{G}$ interpretation of any sequence of declarations, so $\mathscr{G}[\![rest]\!]_{\Omega_G} = \Omega_G$. By substitution,

$$\mathscr{G}[\![gd]\!]_{G_0} = \Omega_G$$

Showing that the state-based interpretation is Ω_S takes slightly more work. To start with, we are given that S_0 and G_0 are consistent, and that they are not Ω. Therefore,

$$S_0.graph = G_0$$

We have assumed for this case that $\mathscr{G}[\![d]\!]_{G_0} = \Omega_G$, and so by substitution:

$$\mathscr{G}[\![d]\!]_{S_0.graph} = \Omega_G$$

Case 4.36b tells us that since the $\mathscr{G}$ interpretation of d in terms of $S_0.graph$ is Ω_G, then the $\mathscr{P}$ interpretation in terms of S_0 is Ω_S:

$$\mathscr{P}[\![d]\!]_{S_0} = \Omega_S$$

We can now proceed with a similar argument as for the graph-based interpretation. Definition 4.37 tells us that

$$\mathscr{P}[\![gd]\!]_{S_0} = \mathscr{P}[\![rest]\!]_{\mathscr{P}[\![d]\!]_{S_0}}$$

We have just proved that $\mathscr{P}[\![d]\!]_{S_0} = \Omega_S$, so by substitution,

$$\mathscr{P}[\![gd]\!]_{S_0} = \mathscr{P}[\![rest]\!]_{\Omega_S}$$

Theorem 4.38 tells us that Ω_S propagates through the interpretation of any sequence of statements, so $\mathscr{P}[\![rest]\!]_{\Omega_S} = \Omega_S$. By substitution,

$$\mathscr{P}[\![gd]\!]_{S_0} = \Omega_S$$

We have showed that $\mathscr{P}[\![gd]\!]_{S_0} = \Omega_S$ and that $\mathscr{G}[\![gd]\!]_{G_0} = \Omega_G$. We know that Ω_S and Ω_G are trivially consistent; therefore, by substitution, we know that $\mathscr{P}[\![gd]\!]_{S_0}$ and $\mathscr{G}[\![gd]\!]_{G_0}$ are consistent as well.

Inductive case 2. Let $gd = \langle d \rangle \frown rest$. In this case, we assume that the first declaration in the sequence does not interpret to an error, so we also let $\mathscr{P}[\![d]\!]_{S_0} = S_1$ and $\mathscr{G}[\![d]\!]_{G_0} = G_1$. Assume that, by induction, if any S' and G' are consistent, then $\mathscr{P}[\![rest]\!]_{S'}$ and $\mathscr{G}[\![rest]\!]_{G'}$ are also consistent. We would like to show that $\mathscr{P}[\![gd]\!]_{S_0}$ and $\mathscr{G}[\![gd]\!]_{G_0}$ are consistent.

We are given that $\mathscr{G}[\![d]\!]_{G_0}$, and therefore $\mathscr{G}[\![d]\!]_{S_0.graph}$, do not evaluate to Ω. Therefore, Case 4.36b tells us that interpreting d in terms of S_0 gives us a new state whose graph is $\mathscr{G}[\![d]\!]_{S_0.graph}$:

$$(\mathscr{P}[\![d]\!]_{S_0}).graph = \mathscr{G}[\![d]\!]_{(S_0.graph)}$$

We are also given that S_0 and G_0 are consistent. Since they are not Ω, we know that $S_0.graph = G_0$. Therefore, by substitution,

$$(\mathscr{P}[\![d]\!]_{S_0}).graph = \mathscr{G}[\![d]\!]_{G_0}$$

We are also given that $\mathscr{P}[\![d]\!]_{S_0} = S_1$ and that $\mathscr{G}[\![d]\!]_{G_0} = G_1$. Again by substitution,

$$S_1.graph = G_1$$

This shows that S_1 and G_1 are consistent. The inductive hypothesis tells us that interpreting *rest* in terms of S_1 and G_1 maintains this consistency:

$$\mathscr{P}[\![rest]\!]_{S_1} \text{ and } \mathscr{G}[\![rest]\!]_{G_1} \text{ are consistent}$$

Next, Definition 4.14 tells us that

$$\mathscr{G}[\![gd]\!]_{G_0} = \mathscr{G}[\![rest]\!]_{\mathscr{G}[\![d]\!]_{G_0}}$$

We are given that $\mathscr{G}[\![d]\!]_{G_0} = G_1$, so by substitution,

$$\mathscr{G}[\![gd]\!]_{G_0} = \mathscr{G}[\![rest]\!]_{G_1}$$

Similarly, Definition 4.37 tells us that

$$\mathscr{P}[\![gd]\!]_{S_0} = \mathscr{P}[\![rest]\!]_{\mathscr{P}[\![d]\!]_{S_0}}$$

We are given that $\mathscr{P}[\![d]\!]_{S_0} = S_1$, so by substitution,

$$\mathscr{P}[\![gd]\!]_{S_0} = \mathscr{P}[\![rest]\!]_{S_1}$$

Since we have shown that $\mathscr{P}[\![rest]\!]_{S_1}$ and $\mathscr{G}[\![rest]\!]_{G_1}$ are consistent, by substitution,

$\mathscr{P}[\![gd]\!]_{S_0}$ and $\mathscr{G}[\![gd]\!]_{G_0}$ are consistent

□

We have just proved that if we have a consistent state and graph, then every declaration in the language maintains this consistency. We can now show that entire graph declarations yield consistent results regardless of which interpretation function we use.

Theorem 4.41 (Consistency of the two languages). *Given an unpatterned graph declaration gd, $\mathscr{P}[\![gd]\!]$ and $\mathscr{G}[\![gd]\!]$ are consistent.*

Proof. Definition 4.39 tells us that the $\mathscr{P}$ interpretation of gd starts with an empty state:

$$\mathscr{P}[\![gd]\!] = \mathscr{P}[\![gd]\!]_{\varnothing_{State}}$$

Similarly, Definition 4.16 tells us that the $\mathscr{G}$ interpretation starts with an empty graph:

$$\mathscr{G}[\![gd]\!] = \mathscr{G}[\![gd]\!]_{\varnothing_{Graph}}$$

Since the empty state's graph is, by Definition 4.35, the empty graph, it is obvious that

$\varnothing_{State}$ and $\varnothing_{Graph}$ are consistent

Lemma 4.40 tells us that interpreting any sequence in terms of $\varnothing_{State}$ and $\varnothing_{Graph}$ maintains this consistency:

$\mathscr{P}[\![gd]\!]_{\varnothing_{State}}$ and $\mathscr{G}[\![gd]\!]_{\varnothing_{Graph}}$ are consistent

Therefore, by substitution,

$\mathscr{P}[\![gd]\!]$ and $\mathscr{G}[\![gd]\!]$ are consistent

□

For the second part of our correctness proof, we must show that the new transformation language is still "useful" — that any patterned graph declaration produces either an error or a well-formed graph.

Theorem 4.42 (No errors implies well-formedness). *For any patterned graph declaration gd_P, if $\mathscr{P}[\![gd_P]\!] \neq \Omega$, then $\mathscr{P}[\![gd_P]\!].graph$ is well-formed.*

Proof. We prove this by induction on the sequence of statements in gd_P. Any errors that arise are propagated through to the end by Case 4.36a. By Definition 4.39, we start with the empty state, whose graph is trivially well-formed. We must therefore show that each declaration, if given a state with a well-formed graph, must produce either Ω or another state with a well-formed graph.

For unpatterned declarations, we use the graph-based interpretation function from the previous section directly on the state's graph. By Theorem 4.17, we know that this function maintains well-formedness.

Pattern definitions obviously maintain well-formedness, since they do not modify the state's graph.

Pattern applications can generate Ω if the pattern does not exist, or if the binding does not supply values for each of the parameters used in the pattern's body. Otherwise, according to Definition 4.36d, the pattern body is evaluated in terms of the binding, yielding a sequence of unpatterned declarations. This sequence is then interpreted in terms of the state's graph. As before, Theorem 4.17 shows that a sequence of unpatterned declarations maintains well-formedness or generates Ω. Therefore, pattern applications must, as well. □

Another interesting property of the new language is that it is *reducible*. That is, it is possible to replace all of the pattern statements in a patterned graph declaration with unpatterned declarations, without changing the resulting graph.

Theorem 4.43 (Reducibility of patterns). *Every patterned graph declaration gd_P that does not interpret to Ω can be reduced to an equivalent unpatterned graph declaration gd, where $\mathscr{P}[\![gd_P]\!]$ is consistent with $\mathscr{G}[\![gd]\!]$.*

Proof. Since $\mathscr{P}[\![gd_P]\!] \neq \Omega$, we know by Theorem 4.42 that $\mathscr{P}[\![gd_P]\!].graph$ is well-formed. Further, we know by Theorem 4.18 that this well-formed graph must have some unpatterned graph declaration gd that constructs it. □

Summary

In this chapter we presented a transformation discovery framework based on *transformation graphs*. This framework exploits the fact that datatypes can be opaque: since they are opaque, the datatypes can be used as the nodes of a transformation graph, without having to provide any details of their internal structure. This knowledge is encapsulated into the atomic transformations that operate on a datatype; these atomic transformations then form the edges of the graph. We can use declaration patterns to simplify graph definitions when there are common patterns of datatypes and transformations that occur frequently.

With this graph structure in place, a path represents a compound transformation between arbitrary datatypes; this works because the atomic transformations that comprise the compound transformation are composable. We have defined a transformation to be any translation between two datatypes that maintains some equivalence; a compound transformation, then, maintains any equivalence that is maintained by each of its constituent atomic transformations.

Because compound transformations are represented by paths, we can use any efficient pathfinding algorithm to discover them. Again, this works because the datatypes are opaque: we do not (and cannot) concern ourselves with the detailed internal structure of the datataypes while searching for a solution. Instead, all of the possible interactions with a datatype are fully specified by which atomic transformations are available that can operate on it, greatly reducing the state space we must search through when looking for a transformation solution. Moreover, our use of properties and weight providers allows different clients to use the same transformation graph for different use cases, which have different criteria for determining which compound transformation is optimal.

Finally, we have presented a formalization and denotational semantics for our graph-based framework, which served two purposes. First, it showed that the more complex features, like declaration patterns, can be represented in terms of a simple "datatypes and atomic transformations" model

without losing correctness. Second, the denotational semantics can be used, along with the graph definition file format, as a description of a language-independent implementation of the transformation framework.

Having presented a conceptual overview and formalization of the graph-based model, we will next consider two case studies. These case studies, while hypothetical, will be of sufficient complexity to show that our transformation discovery approach can be useful in practice. This will also highlight the limitations of the model, specifically with how it handles datatypes whose internal structure changes rapidly, either over time or by use case. In later chapters, we will consider extensions to the graph model that let us overcome these weaknesses.

5

CASE STUDY
Zucchini Corporation

In this chapter we present a case study based on a hypothetical manufacturing company named Zucchini Corporation. Zucchini has an existing software system that it uses to keep track of its product line, warehouse inventory, and manufacturing schedule. This kind of *manufacturing resource planning* (MRP) application is very common in industrial contexts. While this particular example is very simple when compared to an MRP application that would be used by a real-world organization, it contains enough detail to highlight various data transformation issues that arise during such a system's development and use, and to show how the graph-based transformation framework from the previous chapter can be a useful solution.

This case study examines one particular use case in Zucchini's day-to-day interactions with its MRP software — that of generating purchase orders. The products that Zucchini manufactures each require certain subcomponent parts before being assembled. Some of these subcomponents are also produced in-house, meaning that they too have constituent subparts. This breakdown repeats, yielding a *bill of materials* tree for each product manufactured and sold by Zucchini. The internal nodes of the tree are subcomponents that are produced in-house; the leaf nodes are externally supplied subcomponents that are bought from suppliers.

We focus on these leaf nodes. In response to a set of orders from Zucchini's customers, the MRP application will schedule the manufacturing facilities to produce the appropriate products, including all of the necessary in-house subcomponents defined in the bill of materials. Each of these scheduled productions requires its constituent parts to be on-hand, either already in Zucchini's standing inventory, or bought and received from a supplier in time for manufacturing to begin. The details of how the MRP application schedules productions and maintains a standing inventory are well outside the scope of this case study. For our purposes, we treat the MRP application as an opaque black box that generates *purchase orders* — such as the one shown in Figure 5.1 — which request the purchase of subcomponents from their suppliers, and the delivery of those items by a certain date.

In this case study, we assume that Zucchini's suppliers have differing capabilities for receiving purchase orders electronically, and differing formats for encoding the purchase orders. We examine how data transformation graphs, as defined in Chapter 4, can help mitigate these differences.

19 October 2006

Purchase order: 200610-0102
Customer number: 20693-482

Bill to: Zucchini Corp.
Corp. HQ
New York, NY

Ship to: Zucchini Corp.
NY-Penn Warehouse
Scranton, PA

Qty	Product		Unit price	Total price
25	WD-21-893	Whizza-widget	4.25	106.25
14 yd^2	CL-INV-289	Invisibility fabric	19.50	273.00
	Subtotal			379.25
	5% sales tax (New York state)			18.96
	TOTAL			**$398.21**

Figure 5.1: An example purchase order

5.1 Initial capabilities

Zucchini Corp. runs a slightly customized version of a common MRP software suite written by Turnipsoft Solutions, a large enterprise software developer. Purchase orders generated by this software can be output in one of two formats. The first is a simple printout in the Adobe Portable Document Format (PDF). The second, the Turnipsoft Binary Data Interchange (TBDI) format, is suitable for electronic interchange. TBDI is a binary format for business documents developed specifically for the Turnipsoft suite. The details of the TBDI purchase order format are shown in Tables 5.1 and 5.2. A purchase order consists of a *header section*, containing information about the purchase order as a whole, followed by a sequence of *line item sections*, one for each line item in the order.

The header section, shown in Table 5.1, starts with a *magic number* that identifies this as a TBDI document. This magic number is always the hexadecimal quantity ⟪54 42 44 49⟫, which is equivalent to the ASCII character string "TBDI". It is followed by a *document type field* that identifies which particular TBDI business document is represented — in our case, a purchase order. Next comes a *length field* that states the overall length in bytes of the purchase order document, including all of its line items. These three fields are all encoded as unsigned 32-bit integers. Next comes the *purchase order date*, which is the date that the purchasing company generated this order. The date is expressed as eight numeric ASCII bytes (⟪30⟫ through ⟪39⟫, inclusive), in *YYYYMMDD* format. Next is the *purchase order number*. This is encoded as an *ASCIIZ string*, which is a sequence of 8-bit ASCII characters terminated by a null byte (⟪00⟫). Next are the *customer ID*, *billing address*, and *shipping address* fields, also encoded as ASCIIZ strings. Next comes the *currency field*, an ASCIIZ string specifying the ISO 4217 [55] code of the currency used throughout this purchase order. Amounts in differing currencies are not supported within a single order. Next comes the *decimal modifier*, a signed 32-bit integer that is used to encode non-integral currency amounts in the purchase order using only integral types. Given a decimal modifier d, a currency value z actually corresponds to the amount $z \times 10^{-d}$. For instance, to encode the amount US$53.42, one could use a currency field of "USD", a decimal modifier of 2, and an amount value of 5,342. Like the currency field, a single decimal

Table 5.1: TBDI purchase order header format

Field name	Length	Type	Description
`Magic`	4	`uint32`	Magic number (`0x54424449`)
`DocType`	4	`uint32`	Document type (1 = purchase order)
`Length`	4	`uint32`	Length (in bytes) of the entire purchase order
`Date`	8	`ASCII`	Date the purchase order was placed, in `YYYYMMDD` format
`PONumber`	$1 + n$	`ASCIIZ`	Purchase order number
`CustID`	$1 + n$	`ASCIIZ`	Customer ID
`BillTo`	$1 + n$	`ASCIIZ`	Billing address
`ShipTo`	$1 + n$	`ASCIIZ`	Shipping address
`Currency`	$1 + n$	`ASCIIZ`	The currency used for each amount in this purchase order
`Decimals`	4	`int32`	The decimal modifier for this purchase order
`Total`	4	`int32`	The total amount of the purchase order
`NumLines`	4	`uint32`	The number of line items on this purchase order

modifier is used throughout the purchase order. Following the currency information is *total purchase order price*, a signed 32-bit currency amount. The last field in the header specifies the number of line item sections that follow.

Table 5.2: TBDI line item format

Field name	Length	Type	Description
`Length`	4	`uint32`	Length (in bytes) of this line item
`ProductID`	$1 + n$	`ASCIIZ`	The product ID for this line item
`Quantity`	4	`int32`	The quantity for this line item
`Unit`	$1 + n$	`ASCIIZ`	The unit of the quantity field
`UnitPrice`	4	`int32`	Price per unit for this line item
`LineTotal`	4	`int32`	Total price for this line item

Each line item section, shown in Table 5.2, starts with a *length field*, specifying the total length of the line item section in bytes. Next is the *product ID* for this line, expressed as an ASCIIZ string. Next is the *quantity*, expressed as a signed 32-bit integer. Next is the *unit field*, an ASCIIZ string that specifies the units of the quantity value. Often this will be "`ea`" (for *each*), signifying a product sold as discrete entities. Next is the *unit price*, a signed 32-bit currency amount, which expresses the cost for each unit of product. Last is the *line total*, which will usually equal the unit price multiplied by the quantity. This will not be the case for a line item that does not represent an actual product, such as a subtotal.

Any element which goes into the total price of the purchase order, including taxes, service fees, and shipping charges, should be represented by a line item section. Extra line item sections can be used to represent informational line items such as subtotals, so the total cost of the purchase order

might not exactly equal the sum of every line item. (It must, however, not be greater than this sum.) All of the identifiers in a purchase order are expected to contain only alphanumeric characters, spaces, and hyphens, a restriction carried over from the Turnipsoft MRP software suite. All multi-byte values are big-endian.

As an example, we can look at the purchase order shown in Figure 5.1. We can encode this purchase order as a TBDI document as shown in Table 5.3.

Table 5.3: Purchase order encoded as a TBDI document

Data	Field	Length
	HEADER	
"TBDI"	Magic	4
⟪00 00 00 01⟫	DocType	4
⟪00 00 01 00⟫	Length (256)	4
"20061019"	Date	8
"200610-0102" ⟪00⟫	PONumber	12
"20693-482" ⟪00⟫	CustID	10
"Zucchini Corp." ⟪0A⟫	BillTo	15
"Corp. HQ" ⟪0A⟫	BillTo	9
"New York, NY" ⟪00⟫	BillTo	13
"Zucchini Corp." ⟪0A⟫	ShipTo	15
"NY-Penn Warehouse" ⟪0A⟫	ShipTo	18
"Scranton, PA" ⟪00⟫	ShipTo	13
"USD" ⟪00⟫	Currency	4
⟪00 00 00 02⟫	Decimals (2)	4
⟪00 00 9B 8D⟫	Total (39,821 ⇒ $398.21)	4
⟪00 00 00 04⟫	NumLines (4)	4
	Section length	*141*
	LINE ITEM 1	
⟪00 00 00 1D⟫	Length (29)	4
"WD-21-893" ⟪00⟫	ProductID	10
⟪00 00 00 19⟫	Quantity (25)	4
"ea" ⟪00⟫	Unit	3
⟪00 00 01 A9⟫	UnitPrice (425 ⇒ $4.25)	4
⟪00 00 29 81⟫	LineTotal (10,625 ⇒ $106.25)	4
	Section length	*29*
	LINE ITEM 2	
⟪00 00 00 23⟫	Length (35)	4
"CL-INV-289" ⟪00⟫	ProductID	11
⟪00 00 00 0E⟫	Quantity (14)	4
"sq. yd." ⟪00⟫	Unit	8
⟪00 00 07 9E⟫	UnitPrice (1950 ⇒ $19.50)	4
⟪00 00 6A A4⟫	LineTotal (27,300 ⇒ $273.00)	4
	Section length	*35*
	LINE ITEM 3	
⟪00 00 00 18⟫	Length (24)	4
"SUB" ⟪00⟫	ProductID	4
⟪00 00 00 00⟫	Quantity (0)	4
"N/A" ⟪00⟫	Unit	4
⟪00 00 00 00⟫	UnitPrice (0)	4
⟪00 00 94 25⟫	LineTotal (37,925 ⇒ $379.25)	4
	Section length	*24*

Table 5.3: *(continued)*

Data	Field	Length
	LINE ITEM 4	
⟪00 00 00 1B⟫	Length (27)	4
"TAX-NY" ⟪00⟫	ProductID	7
⟪00 00 00 00⟫	Quantity (0)	4
"N/A" ⟪00⟫	Unit	4
⟪00 00 00 00⟫	UnitPrice (0)	4
⟪00 00 07 68⟫	LineTotal (1,896 ⇒ $18.96)	4
	Section length	*27*

5.2 Supplier with the same software

The first of Zucchini's suppliers that we examine is Eggplant Widgets, Ltd. Like Zucchini, Eggplant also uses the Turnipsoft application suite for its MRP needs. Since both companies' applications support the TBDI format natively, no actual transformation is needed before sending the data to the Eggplant system. We are left with the simple transformation graph shown in Figure 5.2.

Figure 5.2: Initial purchase order transformation graph

5.3 Supplier with non-electronic purchasing

The next supplier we examine is Amalgamated Okra (AO). AO is an old, well-established company, and has a legacy purchasing process in place that is non-electronic. Purchase orders must be sent to AO's office via the postal service or by fax, after which orders are confirmed by telephone.

Since Zucchini's software can output purchase orders in the PDF format for printing, we once again do not need any transformations. We can add the Turnipsoft PDF format to the transformation graph, as shown in Figure 5.3.

Figure 5.3: Adding PDF to the purchase order graph

5.4 The big consortium

Our next supplier is Global Cucumber, Inc., one of the founding members of the VeggieBiz Electronic Commerce Consortium. The Consortium was formed several years ago to define an XML- and Web Service-based communication standard for companies in this business area.

Under these auspices, the Consortium has developed the VeggieBiz XML Business Document (VXBD) format. Like TBDI, the Consortium standard defines many different business document for-

mats under the name VXBD. We look only at the purchase order format. This format is described by the following RelaxNG [26, 25] schema:

```
start = purchaseOrder

customerID = element customerID { xsd:anyURI }
poNumber   = element purchaseOrderNumber { xsd:anyURI }
productID  = element productID { xsd:anyURI }

addressLine = element line { text }

amount = {
    attribute currency { text },
    xsd:decimal
}

lineItem = element lineItem {
    productID,
    (
        element quantity {
            attribute unit { text }?,
            xsd:integer
        },
        element pricePer { amount }
    )?,
    element totalLinePrice { amount }
}

purchaseOrder = element purchaseOrder {
    element date { xsd:date },
    poNumber,
    customerID,
    element billingAddress { addressLine+ },
    element shippingAddress { addressLine+ }?,
    element lineItems { lineItem+ },
    (
        element subtotal { amount },
        element adjustments { lineItem+ }
    )?,
    element total { amount }
}
```

Apart from the obvious difference in syntaxes, there are some subtle differences between the TBDI and VXBD formats. First, VXBD assumes that all of the identifiers (purchase order numbers, customer IDs, product IDs) are expressed as URIs. TBDI, on the other hand, inherits its assumptions about identifiers from the Turnipsoft application that it was written for, and assumes that identifiers are short and contain only alphanumeric characters, spaces, and hyphens.

Another difference regards the line items in the purchase order. In TBDI, there is a single set of line items, which contains not only the main line items of the purchase order, but also informational items such as subtotals, and extra charges such as taxes and shipping. In VXBD, the `lineItems` tag only contains the purchase order's main line items. Extra charges like taxes and shipping are

considered *adjustments*, and appear in a separate section. The subtotal (which must be the sum of the costs in the `lineItems` tag) only appears if there are adjustments in the purchase order.

A final difference involves the currency amounts in the purchase order. In TBDI, the header section contains a currency and decimal modifier field that is in force for the entire purchase order document. In the VXBD format, currency amounts are encoded as strings using XML Schema's *`xsd:decimal`* data type, so decimal modifiers are not needed. Further, each amount has its own `currency` XML attribute, allowing for different currencies to appear in a single purchase order.

The purchase order from Figure 5.1 can be expressed in this VXBD format as follows:

```
<purchaseOrder>
  <date>2006-10-19</date>
  <purchaseOrderNumber>
    http://turnipmrp.zucchini.com/purchaseOrders/200610-0102
  </purchaseOrderNumber>

  <customerID>
    http://vxbd.globalcucumber.co.uk/customers/20693-482
  </customerID>
  <billingAddress>
    <line>Zucchini Corp.</line>
    <line>Corp. HQ</line>
    <line>New York, NY</line>
  </billingAddress>
  <shippingAddress>
    <line>Zucchini Corp.</line>
    <line>NY-Penn Warehouse</line>
    <line>Scranton, PA</line>
  </shippingAddress>

  <lineItems>
    <lineItem>
      <productID>
        http://vxbd.globalcucumber.co.uk/products/WD-21-893
      </productID>
      <quantity>25</quantity>
      <pricePer currency="USD">4.25</pricePer>
      <totalLinePrice currency="USD">106.25</totalLinePrice>
    </lineItem>
    <lineItem>
      <productID>
        http://vxbd.globalcucumber.co.uk/products/CL-INV-289
      </productID>
      <quantity unit="sq. yd.">14</quantity>
      <pricePer currency="USD">19.50</pricePer>
      <totalLinePrice currency="USD">273.00</totalLinePrice>
    </lineItem>
  </lineItems>

  <subtotal currency="USD">379.25</subtotal>
  <adjustments>
    <lineItem>
      <productID>
```

```
        http://www.state.ny.us/sales-tax
      </productID>
      <totalLinePrice currency="USD">18.96</totalLinePrice>
    </lineItem>
  </adjustments>

  <total currency="USD">398.21</total>
</purchaseOrder>
```

Zucchini's software does not directly support the VXBD format, and for various political reasons, Turnipsoft does not plan on adding this support in the near future. Zucchini would still like to automate purchasing with Global Cucumber, however, so some transformation into VXBD is needed. Luckily, the transformation from TBDI into VXBD is not very complex, and Zucchini's in-house software team is able to develop and test an implementation in a couple of months. While they are at it, they implement an inverse transformation, as well. Since VXBD supports multiple currencies within a single purchase order, whereas TBDI does not, the inverse transformation must be partial — there is not always an equivalent TBDI document for a VXBD purchase order.

Another interesting feature of the transformations is that they must be implemented in a declaration pattern, rather than as two individual transformations. This is due to the different nature of the identifiers in the two formats. This is a similar problem to the fact that different companies will have different identifiers for the same product; when Zucchini's MRP software decides that it needs to order a certain quantity of product *A*, it has to remember that the supplier calls this product *B* instead. When outputting a purchase order, it uses the supplier's identifier for the product, not its own.

It would seem that this mechanism could be used to handle the different identifiers in TBDI and VXBD. However, this is not the case — the Turnipsoft developers made an unfortunate decision to place too many restrictions on what is considered a valid identifier. Since URIs contain characters that are not valid in a Turnipsoft identifier, it is therefore not possible to use the same identifier mapping mechanism to translate Zucchini's alphanumeric product identifiers into the URIs required by VXBD. Instead, a separate mapping must be maintained. Rather than incorporating this mapping into the transformations directly, the transformations can be encapsulated into a declaration pattern that takes this identifier mapping as a parameter. This allows the transformations to be reused by other Turnipsoft users; all that is needed is to provide a different identifier mapping when applying the declaration pattern.

We can add the VXBD format, and the transformation pattern developed by Zucchini, to the graph as shown in Figure 5.4. The red dashed box around the new transformations indicates that they are defined in a parameterized declaration pattern; the box's label defines the pattern's formal parameters.

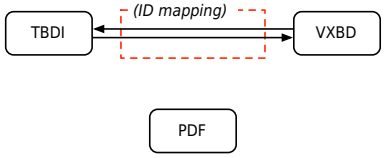


Figure 5.4: Adding VXBD to the purchase order graph

5.5 A competing consortium

While the goals of the VeggieBiz consortium are laudable, there was a technical and political disagreement that developed early in the standardization process. As often happens, this led some of the founding member companies to split away and form a competing consortium. Thinking that the VeggieBiz communication protocols were needlessly complex, the new consortium called its the competing standard the Simplified Business Exchange Platform (SBEP). The SBEP Group had the same overall goals as the VeggieBiz consortium: to develop a standard suite of data formats and protocols for electronic commerce. The main difference of opinion was over the choice of implementation; whereas VeggieBiz developed a solution based on Web Services [2], SBEP opted for a more lightweight solution based on a REST architecture [40].

Both consortia decided to use XML for their data interchange formats; unfortunately, the schism occurred before the design of the VXBD format was finalized. As a result, the SBEP Group proceeded to develop a similar, but incompatible, XML purchase order format, conforming to the following schema:

```
start = purchaseOrder

customerID = element customerID { xsd:anyURI }
poNumber   = element purchaseOrderNumber { xsd:anyURI }
productID  = element productID { xsd:anyURI }

addressLine = element line { text }

lineItem = element lineItem {
    productID,
    (
        element quantity {
            attribute unit { text }?,
            xsd:integer
        },
        element pricePer { xsd:decimal }
    )?,
    element totalLinePrice { xsd:decimal }
}

purchaseOrder = element purchaseOrder {
    element date { xsd:date },
    poNumber,
    customerID,
    element currency { text },
    element billingAddress { addressLine+ },
    element section {
        element shippingAddress { addressLine+ }?,
        element lineItems { lineItem+ },
        (
            element subtotal { xsd:decimal },
            element adjustments { lineItem+ }
        )?
    }+,
    (
```

```
        element subtotal { xsd:decimal },
        element adjustments { lineItem+ }
    )?,
    element total { xsd:decimal }
}
```

As the two XML formats have a common ancestry in the pre-schism consortium, the two datatypes are obviously quite similar. The differences stem from design decisions that were made after the split. SBEP decided not to support multiple currencies within a single purchase order. They did decide, however, to support multiple shipping addresses; a single SBEP purchase order can contain separate line item sections, each of which can be delivered to different locations. Our example purchase order, though, does not require this extra feature, so its SBEP rendering is very similar to the VXBD version:

```
<purchaseOrder>
  <date>2006-10-19</date>
  <purchaseOrderNumber>
    http://turnipmrp.zucchini.com/purchaseOrders/200610-0102
  </purchaseOrderNumber>

  <customerID>
    http://sbep.broccoli.com/customers/20693-482
  </customerID>
  <currency>USD</currency>
  <billingAddress>
    <line>Zucchini Corp.</line>
    <line>Corp. HQ</line>
    <line>New York, NY</line>
  </billingAddress>

  <section>
    <shippingAddress>
      <line>Zucchini Corp.</line>
      <line>NY-Penn Warehouse</line>
      <line>Scranton, PA</line>
    </shippingAddress>
    <lineItems>
      <lineItem>
        <productID>
          http://sbep.broccoli.com/products/WD-21-893
        </productID>
        <quantity>25</quantity>
        <pricePer>4.25</pricePer>
        <totalLinePrice>106.25</totalLinePrice>
      </lineItem>
      <lineItem>
        <productID>
          http://sbep.broccoli.com/products/CL-INV-289
        </productID>
        <quantity unit="sq. yd.">14</quantity>
        <pricePer>19.50</pricePer>
        <totalLinePrice>273.00</totalLinePrice>
      </lineItem>
    </lineItems>
```

```
  </section>

  <subtotal>379.25</subtotal>
  <adjustments>
    <lineItem>
      <productID>
        http://www.state.ny.us/sales-tax
      </productID>
      <totalLinePrice>18.96</totalLinePrice>
    </lineItem>
  </adjustments>

  <total>398.21</total>
</purchaseOrder>
```

While the two consortia are unlikely to recombine in the near future, many companies, including Zucchini Corp., are in the unenviable position of having to do business with companies that use both architectures. As a result, an interoperability group was formed to try to mitigate the differences between the two. Their attempts to reconcile the Web Service implementation and the REST implementation are outside the scope of this case study (and many members of the interoperability group secretly believe that these attempts will never be successful). However, their work on the data formats has been fruitful; the similarity of the two XML schemas made it straightforward to define two XSLT [24] transformations for converting between them. As before, these transformations will necessarily be partial: a SBEP purchase order with multiple shipping addresses cannot be transformed into VXBD, while a VXBD purchase order with multiple currencies cannot be transformed into the SBEP format. Since both formats use URIs for the various identifiers in the purchase order, no identifier mapping is needed; therefore, unlike in the previous example, we do not need to define these transformations in a declaration pattern. We can add the SBEP datatype, and the XSLT transformations, to our transformation graph as shown in Figure 5.5.

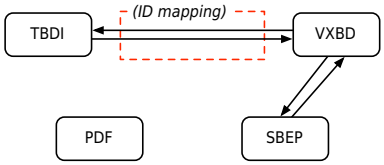


Figure 5.5: Adding SBEP to the purchase order graph

5.6 Putting it together

With the transformation graph that we are slowly piecing together, we can put together a generic strategy for handling purchase orders regardless of the supplier they are intended for. The Turnipsoft software is able to create the purchase orders in two formats directly. As shown in Figure 5.6, we highlight this in the transformation graph with a thicker border for these two datatypes. This graph also shows the VXBD declaration pattern instantiated (and therefore shown with a blue dashed border) with the Zucchini-Global Cucumber identifier mapping.

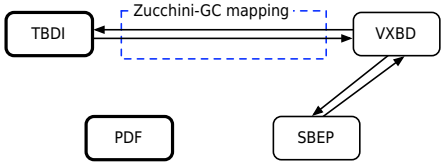


Figure 5.6: Identifying the source nodes in the transformation graph

We then maintain a mapping between suppliers and datatypes. The mapping that we have developed to this point is shown in Table 5.4. For each supplier, we run the pathfinding algorithm twice — once from each source datatype provided by the MRP application — searching for a shortest path to the supplier's destination type. This gives us the compound transformation necessary to translate the purchase order output of Zucchini's MRP application into a format readable by the corresponding supplier.

Table 5.4: Initial supplier mapping

Supplier	Datatype
Eggplant Widgets, Ltd.	TBDI
Amalgamated Okra	PDF
Global Cucumber, Inc.	VXBD
Broccoli, Inc.	SBEP

Note that we are not considering what to do with the data once it has been transformed into the appropriate type. The functional semantics of invoking a VeggieBiz Web Service to submit the purchase order, for instance, might be quite different from the semantics of the Turnipsoft binary exchange, or the Amalgamated Okra post/telephone process. This might necessitate a fundamentally different application logic on the part of Zucchini's software.

At this point, we have provided enough detail that we can use the Python prototype mentioned in the previous chapter to implement this transformation graph. As described in earlier sections, the transformations between the TBDI and VXBD datatypes are implemented as custom code, containing the specific logic needed to parse these two formats. In the graph, the transformations are declared in a declaration pattern, with a parameter that specifies the necessary identifier mapping; this means that the concrete transformation code must be parameterized similarly. Our pathfinding and execution engine is then responsible for providing the right value for this parameter, based on the value provided in the graph file's `apply pattern` statement.

The transformations between the VXBD and SBEP datatypes, on the other hand, are written as XSLT transformations. Much of the logic for executing an XSLT transformation — for loading the XSLT library and linking the input and output channels to it, for instance — is boilerplate. Our prototype supports the use of *transformation adaptors* to support this kind of boilerplate code. The adaptors are conceptually similar to, and are used in the same way as, the parameterized transformations described previously. In this case, for instance, the XSLT transformation adaptor takes as a parameter the specific XSLT file to use. Transformation adaptors are different, however, in that the same code is used for an entire class of related transformations, rather than for several instantiations of the same transformation. As such, transformation adaptors are largely an implementation detail, and do not

appear in the underlying abstract formalism.

5.7 Supplier that can receive multiple datatypes

One final example to consider is a supplier that can accept multiple purchase order formats. Avocado LLC, for instance, is hedging their bets by supporting both the VXBD and SBEP purchase order formats. This does not require any changes to the transformation graph, but the mapping between suppliers and datatypes can now have multiple datatypes for a single supplier, as shown in Table 5.5.

Table 5.5: Supplier mapping with multiple output formats

Supplier	Datatypes
Eggplant Widgets, Ltd.	TBDI
Amalgamated Okra	PDF
Global Cucumber, Inc.	VXBD
Broccoli, Inc.	SBEP
Avocado LLC	SBEP, VXBD

We can still follow the same basic strategy for finding a purchase order transformation, but we must now execute the discovery algorithm many times. Before, the algorithm was executed D times for each supplier, where D is the number of datatypes that can be output directly by Zucchini's software. Now, we must execute the algorithm $D \cdot S$ times for each supplier, where S is the number of datatypes that the supplier can accept.

Luckily, we can use a clever trick to eliminate many of these new executions. We add a dummy source node and sink node to the graph, which do not correspond to any real datatypes. We then connect the source node to each datatype generated by Zucchini's software, and connect each of the supplier's accepted datatypes to the sink node. The modified graph for Avocado LLC, for example, is shown in Figure 5.7.

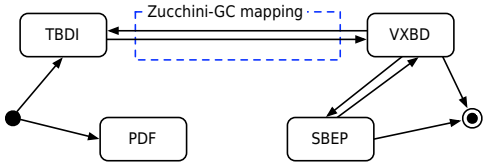


Figure 5.7: Using dummy nodes to reduce the number of discovery executions

A shortest path in this modified graph will provide us with a shortest path (and therefore a compound transformation) from one of Zucchini's output datatypes to one of the supplier's accepted datatypes. Further, there cannot be a shorter path involving one of the other generated or accepted datatypes; if there were, this would yield a shorter path from the source node to the sink node, which is not possible, as we already have the shortest such path. With this modification in place, we now only require a single execution of the discovery algorithm for each of Zucchini's suppliers.

Summary

In this chapter, we have presented a case study based on the hypothetical Zucchini Corporation, and used a prototype implementation of the framework from Chapter 4 to show how transformation graphs could be used as a solution to one particular aspect of their manufacturing process. We were able to support Zucchini suppliers that have differing requirements for accepting electronic purchase orders, taking into account standardization efforts by multiple competing industry consortia. A naïve use of this graph-based solution requires several executions of the discovery algorithm when there are many purchase order formats that can be generated or accepted; however, we can use a dummy source and sink node to reduce this to a single discovery execution, exploiting the fact that the shortest path algorithm can find the optimal pairing.

While our transformation graph framework, as currently presented, is sufficient to tackle the issues raised in this case study, it is not sophisticated enough to deal with all of the use cases that might arise when dealing with data transformations. In our next case study, we present some of these troublesome use cases, giving us insight into possible extensions to the model that could support them.

CASE STUDY

6 Generic data server

In this chapter we present another case study involving the transformation of data, and show how our current transformation framework cannot provide a full solution. We examine the design, implementation, and use of a *generic data server* — an application responsible for the persistent storage of data whose structure, though well-defined, is not known in advance. At first glance, this might seem to fall perfectly under the purview of a standard relational database management system (RDBMS). However, in our running example, not all of the data formats that we need to support will be relational structures; even if we decide to use an RDBMS for the actual storage, we will need to somehow translate each data format into the appropriate relational tables.

We start by presenting a detailed overview of the problem that we wish to solve. We then show how transformation graphs can be used to solve part of this problem: the need to support a wide variety of proprietary image and metadata formats. Initially, we only examine one aspect of the solution, and concentrate on transforming the different representations of the images' pixel arrays. Finally, we highlight some issues that arise when we try to incorporate the more complex metadata formats into the design, giving insight into how we can extend our model to better support them.

6.1 Problem description

The example that we present is based on the Open Microscopy Environment[1] (OME), an overview of which can be found in [104]. The primary goal of the OME project is to provide a centralized mechanism for storing biological microscope images and their associated metadata, including any derived computational results. Two issues complicate this goal. First, the different microscope manufacturers have traditionally stored the images collected by their microscope software in opaque, proprietary image formats. Worse, at acquisition time, they collect differing sets of metadata describing the images. Any software that aims to provide a generic imaging infrastructure must be able to tolerate and reconcile these differences. The OME project has helped alleviate this problem by producing a standard set of the most commonly useful metadata elements for biological imaging, along with an extensible XML-based file format for storing this metadata [46]. As we have pointed out several times throughout this thesis, standardized formats are only useful when there is at least a tacit agree-

[1] http://www.openmicroscopy.org/

ment, by all parties involved, to use them in the spirit that they are intended. Encouragingly, this data model and file format are being accepted and adopted by the microscope manufacturers and scientific researchers, suggesting that it can be a key part of a viable, long-term solution. However, the profusion of proprietary formats remains a problem, since existing tools based on them are still quite common.

The second issue is that the purpose of these images is for researchers to apply sophisticated and complex computational and analytic routines to them. The very nature of scientific research means that we cannot know which particular analyses will be run; what acquired or previously computed metadata will be needed by those analyses; or what kinds of results those analyses will generate. This means that, in addition to the wide variety of acquisition formats that must be supported, we must also support an unknown number of highly specific data formats for storing computational results. The former can be alleviated to a certain extent by developing a standard acquisition format, as OME has done. The latter, however, is much more open-ended, and does not have as simple a solution.

Complicating matters further, it is not just the high-level structure and semantics of the different data formats that varies; it is also the modeling formalisms that are used. As we mentioned previously, it would be tempting to assume that an RDBMS adequately solves our problem: the acquisition metadata could be stored in relational tables; the image pixels themselves could be stored in binary large object (BLOB) columns or as pointers into a separate mass storage; and the relational schema could be extended at runtime to support new, unanticipated data formats needed by the analytic routines.

However, many of the data formats that we need to support do not translate easily into relational tables, and the way in which this data is accessed does not always fall easily into the declarative query pattern provided by SQL. Most obvious is the low-level pixel data. It is theoretically possible to store this information in a relational database — again, either as a BLOB or as a pointer into other storage. The usage of the pixel data will vary by application, though: an analysis routine will likely need to examine the precise numeric value of each and every pixel, while visualization software will often be able to handle compressed representations of the pixels, even if information is lost as a result. Further, the difference in scale between the size of the pixel data and the associated metadata can be several orders of magnitude: a typical high-throughput screen, for instance, can easily have several terabytes of raw image data, as compared to a few kilobytes or megabytes of computational results and acquisition metadata. Assuming that our RDBMS can handle this disparity of scale, tuning it to do so efficiently can be an administrative nightmare. As we will see, the OME design considers this to be such an important consideration that there is a separate component specifically designed to support the fast storage and retrieval of the image pixels.

The metadata associated with the images will also not always fit perfectly into a relational model. Dense numeric matrices, for instance, seem to be a perfect fit for a relational table. However, since they are almost always accessed atomically, the querying capabilities provided by a relational representation are usually wasted; it is more efficient to treat a dense matrix as a special kind of pixel array, and store it in a packed binary form. (Actually, it is more accurate to say the reverse — that a pixel array is a special kind of dense matrix — but this distinction is not important.) On the other hand, most feature abstraction [77] and motion tracking [96] analyses generate data structures that are very hierarchical in nature: images contain features, which contain subfeatures, and so on; in addition to these containment relations, the features are grouped across time and space into linked

lists and other similar structures. While it is certainly possible to design relational tables to store these data structures, they lend themselves much more to the hierarchical model provided by XML.

As we can see, we not only need to support a wide, extensible variety of data formats; the differences appear at all levels of the S classification, from syntax to semantics. A system like an RDBMS, which focuses on a single modeling paradigm, cannot be a solution on its own.

6.2 Pixel transformations

Having given an overview of the problem we are trying to solve, we can now examine one part of the solution: efficiently providing access to the pixels stored in the different proprietary formats. (For now, we only consider accessing the pixels; we will add support for the acquisition metadata in the next section.) If OME were designed to be a single, standalone application (or a single library that many applications link to), we could use an object-oriented design like the one described by the UML diagram in Figure 6.1. This would define an *Image* interface that defines methods for reading the pixels from a particular image. There would then be a class for each of the specific image formats supported, each of which would be responsible for implementing the logic for reading pixels from that particular format. Client code would then be written strictly in terms of the *Image* interface; the underlying server code would instantiate the implementation class appropriate to each image as it is encountered. Further, this design is extensible: a new image format could be supported simply by writing a new implementation class and ensuring that the server's instantiation code is aware of it. The client code would require no modification.

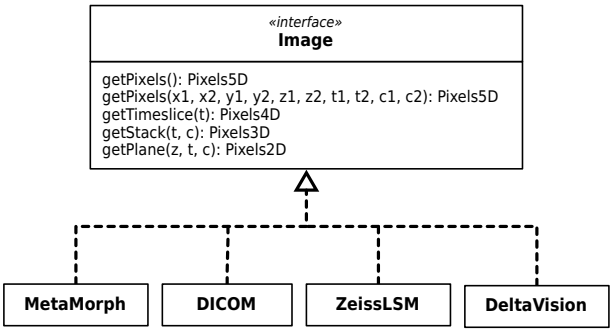


Figure 6.1: Possible object-oriented design for the image server

However, OME is not intended to be a single application or library — rather, it is a component in a larger, decoupled, heterogeneous system, as shown in Figure 6.2. Client components can be third-party software, over which the OME developers have no design control. Further, they will be running on different physical machines, and therefore cannot always use OME-provided code as a linked library.

To fit into this design, OME includes a separate *image server* component (OMEIS) that is responsible for providing fast, network-visible, format-neutral access to the images' pixel data. This component effectively serves the same purpose as the *Image* interface, but over an HTTP-based [41] network protocol instead of as a locally linked object-oriented library. As shown in Figure 6.3, its interface also includes a new operation for storing the pixels, instead of just providing read access.

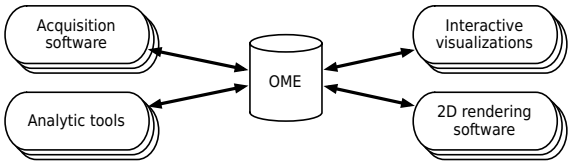


Figure 6.2: Decoupled design of the OME system

Since the implementation of the image server is hidden behind this HTTP interface, it can be implemented using whichever design (and programming language) is most appropriate, without having to consider the design and language used to implement its clients. Specifically, we could choose to write the image server in an object-oriented language like C++, Java, or C#, and use the same basic design from Figure 6.1.

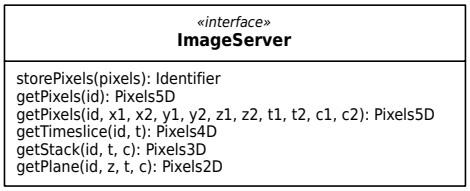


Figure 6.3: Interface provided by the image server

However, to meet its speed requirements, the image server was designed differently. Instead of a complex, extensible program that itself knows about many image formats, the image server, written in C, only handles a single internal image format, which is highly optimized to the particular operations provided by the server's interface. This has the benefit that the image server, even when accessed via the network, is quite fast — often limited only by the speed of the hard disks used to store the images. It has the drawback that instead of storing the images on the server directly in their original format, they must first be converted to the image server's internal format, with this conversion logic being the responsibility of another component.

This conversion is an obvious application of a transformation graph, such as the one shown in Figure 6.4. In this particular transformation graph, there is an atomic transformation directly connecting each external image format to the OMEIS internal format, though there is nothing that requires this. Any transformation graph can be used; a particular graph would allow an external format to be stored in the image server as long as there were some compound transformation (represented as a path in the graph) connecting the external format to the OMEIS internal format.

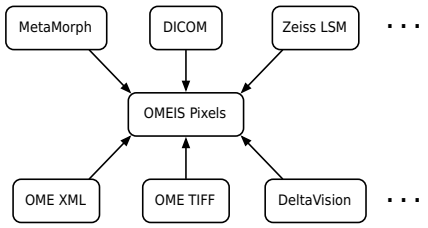


Figure 6.4: An image server transformation graph

With a transformation graph in place, we can add an *importer* component to the design as shown

in Figure 6.5. The importer, which contains the transformation graph, is able to read files from whichever external formats the graph supports. Using its transformation graph, these proprietary formats are translated into the OMEIS internal format, after which they are stored in the image server. Client code can then use the image server's format-neutral API to read the pixel data, regardless of which proprietary format the image was originally stored in.

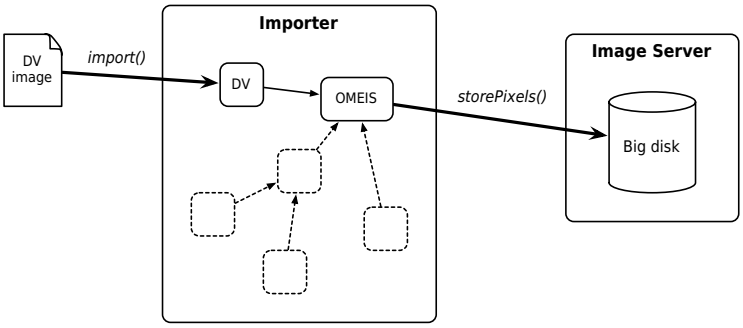


Figure 6.5: Using a transformation graph to import images

Apart from two minor differences, this is the design that OME implements. First, as currently implemented, the import component does not include an actual transformation graph. Rather, it uses an object-oriented design similar to Figure 6.1, but where the implementation classes are responsible for translating an external format into the OMEIS format, rather than for providing direct read access to the pixels. However, this can be viewed as a restricted form of transformation graph, which requires atomic transformations to directly connect the external formats to the OMEIS format, as is the case in the graph in Figure 6.4.

The second, more interesting, difference is that the image server can store the original files in addition to the translated internal pixels. Instead of translating the image into the internal format locally and sending it to the image server in bulk, the important component sends the original file along with *instructions* for how to read each part of the raw pixel array from the original file. This has two benefits. First, even though the original file is opaque to the image server itself (since it has no knowledge of its particular format), there are many client applications that can make use of the original file. It can also be useful to store for archival purposes. Either way, storing the original file and the converted pixels in one place is helpful in that it simplifies the design.

The second benefit is that it allows us to reduce the storage requirements for the server, which can be useful given the amount of image data that a typical laboratory can produce. Microscope images tend to go through a life cycle: initially, they are part of an active experiment, and will be accessed many times by analysis and visualization software. Once the experiment has been completed, though, the image data will be accessed much less frequently. The instructions for extracting the image from the original file are much smaller than the converted pixels themselves; we therefore only need to archive the original file and the translation instructions, knowing that this is sufficient to reconstruct the internal pixel representation if it happens to be needed again. This is only possible because the translation between binary pixel representations is a highly regular, structured transformation — every step that is performed comes from a small set of translation primitives. In the case of arbitrary datatypes, though, the transformations can come in a wide variety of forms; therefore, we would have to archive both the input and output of each transformation execution.

6.3 Metadata transformations

In the previous section we showed how to use a transformation graph to store images in an arbitrary format on a highly specialized image server. However, the external image formats also contain a large amount of metadata describing the images, which we have ignored to this point. In this section, we show what changes must be made to the design to incorporate the associated metadata.

The easiest way to proceed would be to reuse and modify the design of the OMEIS importer from Figure 6.5. The modified design, shown in Figure 6.6, looks remarkably similar. In fact, there are only two changes. First, we are no longer storing the translated results in an image server; this makes sense, since we are no longer dealing with pixel arrays. Instead, the metadata is stored in a new component called the *data server* (OMEDS). As hinted at earlier, we do not want to reinvent any persistence and querying capabilities when perfectly good solutions exist, so the data server is basically just a thin wrapper around an RDBMS.

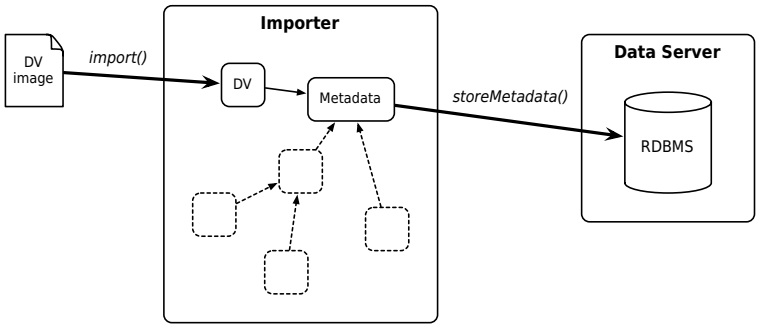


Figure 6.6: Using a transformation graph to import image metadata

This leads nicely to the second difference from the image server importer: the transformation graph in the importer component no longer translates the external formats into an internal pixel representation. Rather, the acquisition metadata is extracted from the external image files and converted to the format expected by the new data server. Since the data server is implemented using a relational database, this internal metadata format will be based on a standardized relational schema designed to store the OME-defined set of common acquisition metadata. This set of metadata is relatively static, even if few existing external formats support it in its entirety. This makes this modified design perfectly acceptable for storing the acquisition metadata — the static nature of the data means that we only need a single internal metadata datatype. Supporting a new external format works as before, by simply adding a new atomic transformation to the importer's graph; this would require no changes to the rest of the importer component or to the data server itself.

This solution works for the static datatypes used for the acquisition metadata. The computational results, on the other hand, are very dynamic — we do not know in advance what datatypes will be needed, and therefore cannot have a single, predefined internal format for storing the data. OME solves this by introducing *semantic types*. Each input and output of an OME analysis module is an instance of some semantic type, which is a compound data structure similar to a C `struct` or Pascal `record`. Each element of a semantic type can be an atomic data value (string, integer, Boolean, etc.) or a simple typed reference to another semantic type instance. This admittedly simple design does not provide the full generality described in previous chapters, but it has the benefit that it translates

readily into both relational and XML schemas.

These extensible semantic types require many corresponding datatypes in a transformation graph. First, we have the description of a semantic type: the name of the type, and the names and types of each element. This gives us an XML schema that defines how semantic type descriptions appear in an OME-XML file, and an equivalent relational schema, which defines the internal database table used by the data server to store descriptions of semantic types. Second, in addition to the datatypes used to *describe* the semantic types, we have datatypes for the semantic type instances themselves. There are at least two datatypes for each semantic type: one for the XML transmission representation, and one for the data server's internal relational table representation.

We can also model the original acquisition metadata format as a collection of semantic types, allowing us to reuse some of this transformation logic. These acquisition semantic types will have even more corresponding datatypes in the transformation graph — one for each external format that contains that particular kind of metadata. In fact, these will be the *same* datatypes that represented the external formats in the image server's transformation graph; after all, it is the same external format, from which we can independently extract the pixel data or acquisition metadata.

For each semantic type, there will be a transformation between the XML representation and the internal relational representation, and vice versa. These transformations correspond, respectively, to importing and exporting arbitrary metadata from the data server using the OME-XML transmission format. The logic of this transformation will be very similar across semantic types, which means that we can use a declaration pattern to greatly simplify the description of the transformation graph. The XML and relational datatypes, along with the transformations between them, are all defined in a declaration pattern, which can be instantiated with a semantic type description.

Combining all of this together, we have the single, large transformation graph shown in Figure 6.7. This has the original transformations between the external image formats and the OMEIS internal pixel format, and can therefore be used as the image server importer's transformation graph. It also contains the three datatypes for describing a semantic type, and the declaration pattern (shown with a red dashed box) used to define the datatypes for each semantic type's instances. This pattern can then be instantiated (shown with a blue dashed box) for each semantic type that is needed. For those semantic types that are known at acquisition time, we can add extra transformations that can extract that metadata from the external formats that contain it. This lets us use the same transformation graph for importing the acquisition metadata, whose types are known in advance, and for storing the computational results, whose are not.

The only major problem with this solution is that the import process now requires multiple steps. There are many different datatypes that must be extracted from a particular external format: one for the internal OMEIS pixel representation, and one for each metadata semantic type that the external format supports. In the previous chapter's case study, we had a similar situation, where a supplier could accept multiple purchase order datatypes. This meant that there were multiple possible destination nodes in the transformation graph. By connecting each of these possible destinations to a dummy sink node, we were able to find the optimal compound transformation to any one of these destinations.

Unfortunately, the same trick does not work with the image and metadata transformation graph. With the purchase order example, we only needed a single compound transformation to *any* of the destinations; in this example, we need to generate values for *all* of the destination datatypes. This

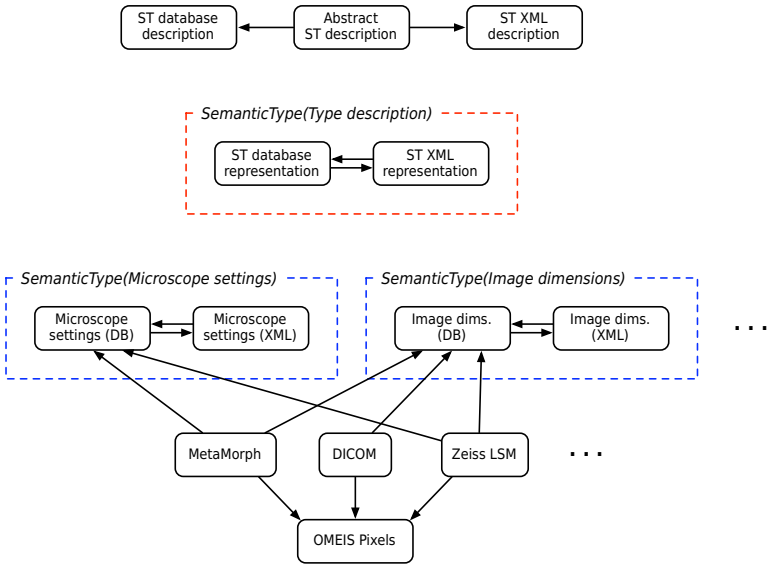


Figure 6.7: A combined transformation graph for importing pixels and metadata

means that we must still execute the transformation discovery algorithm once for every datatype that we want to extract from the external format. Not only does this waste effort in finding the compound transformations, but the compound transformations that we find might also repeat a lot of work in extracting each semantic type independently. In the next chapter, we will show how to expose part of the structure of these datatypes to the transformation layer, which provides a more elegant solution to this problem.

Summary

In this chapter, we presented a second case study, based on a *generic data server*. This server, which must store data from a wide range of datatypes, differs from a standard relational database management system in that the supported datatypes can differ across the full range of the S classification. In an RDBMS, on the other hand, the data can differ structurally and semantically, but must all be based on the relational model.

For highly structured data, such as the pixel data from a microscope image, we can use a specialized internal format to provide highly optimized access to the data. We can then use a transformation graph to translate the pixels into this internal format from the original proprietary image formats. The transformation graph also works in a straightforward way for the images' acquisition metadata, since these datatypes will tend not to change much over time. For more dynamic types, though, such as the computational results from highly experiment-specific analysis routines, more sophisticated techniques are needed. Part of a solution is provided by declaration patterns, which can be parameterized by the user-defined semantic types. However, it would also be helpful to expose part of this dynamic internal structure to the transformation discovery layer; we must be careful, though, to do this in a controlled manner, without losing the efficiency benefits that were gained because of fully opaque datatypes. In the next chapter, we extend our graph model to allow this.

7 Polyadic graphs

As shown in the previous case studies, the graph structure described in Chapter 4 can be used to model some fairly complex, real-world data transformation use cases. However, it includes a major limitation that we would like to lift if possible. The transformation graph model discussed so far has required all of the atomic transformations to be *unary*: they must have exactly one input and one output. In this chapter, we describe *polyadic transformations* — those that can have multiple inputs and outputs — and examine how transformation discovery is affected by the presence of multiple inputs and outputs. First we present several examples of transformation graphs that are more difficult, or impossible, to express without polyadic transformations, including the highly dynamic datatypes found in the case study from Chapter 6. We present these examples using an intuitive *workflow notation* that focuses on atomic transformations as opaque units of computation. This notation, though intuitive, does not lend itself well to an obvious discovery algorithm, so we next introduce a notation based on *sets* of datatypes. In this notation, polyadic discovery can be performed using a simple pathfinding algorithm, as with unary graphs. We then show that the workflow and set notations are *equivalent* — that any set-based solution can be correctly translated into an analogous workflow solution, and vice versa. This allows us to use the same set-based discovery algorithm to find workflow-based compound transformations. Unfortunately, due to the size of the set graphs, this polyadic discovery algorithm is very inefficient; by introducing a third notation, based on *hypergraphs*, we can further show that polyadic discovery is *NP*-hard. This implies that it is not just our algorithm that is inefficient, but rather that the underlying problem is inherently difficult, and that an efficient discovery algorithm is unlikely to exist.

7.1 Overview and examples

To this point we have only considered atomic transformations in terms of how they are pieced together into transformation graphs, such as the simple graph shown in Figure 7.1. Since this view strongly equates transformations to the edges in a graph, we have implicitly included many of the assumptions about a graph edge into our understanding of a transformation. Specifically, since an edge in a directed graph must have exactly one source node and exactly one destination node, we have only been able to consider transformations with exactly one input and exactly one output.

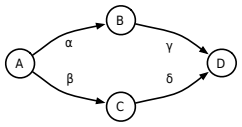


Figure 7.1: A simple transformation graph in the graph notation

A different approach is to focus on transformations as units of computation — since, as we have pointed out previously, we can use this pathfinding technique to find any compound computation that can be defined by an inductive property. This view does not impose the same restriction on the number of inputs and outputs as the graph-based view. Instead, a transformation can have any number of inputs and outputs, and we define a transformation's *arity* to be the maximum of these two values. The special class of transformations with one input and output are called *unary transformations*; those with multiple inputs or outputs are called *polyadic transformations*. We can also consider the arity of an entire graph: a graph that contains only unary transformations is also unary, whereas a graph that contains any polyadic transformations is also polyadic.

We can introduce a new *workflow notation* that highlights this computational view. Instead of merging several transformations together into a graph, each is represented as a "black box" of logic or code. Its input datatypes are listed on the left side of this box, and its output datatypes on the right. Every unary graph can also be represented in this notation: Figure 7.2 shows the same example graph using this workflow notation.

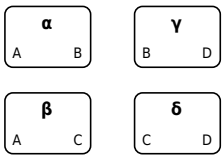


Figure 7.2: The same graph in the workflow notation

In this section we present several examples of transformation graphs that benefit from polyadic transformations, using this workflow notation. We then present a more precise description of what forms a valid workflow-based transformation graph.

7.1.1 Examples

In this section we present several examples of how polyadic transformation graphs can be used to solve problems whose solutions are less efficient or not possible when using unary graphs.

Polyadic compound transformations

One important example is that the compound transformation that we extract from a transformation graph can itself be polyadic. One situation where this is useful is described in Section 5.6 of the Zucchini Corporation case study. Once we had a sufficiently large transformation graph containing all of the various purchase order datatypes, we used a pathfinding algorithm to find optimal transformations for each supplier's datatype. At first, this required executing the pathfinder multiple times for each supplier — once for each possible source type provided directly by the MRP application software

— since we did not know in advance which source datatype would lead to the optimal solution. We were able to eliminate these extra executions by introducing a dummy source node and sink node, which allowed a single discovery execution to find the optimal solution, regardless of which actual source and sink datatype were used.

We could have also solved this using a polyadic graph. To encode the fact that we have multiple source datatypes given to us, we simply add a source node for each to the workflow graph, as shown in Figure 7.3. We also add a sink node for the desired output type, such as VXBD for Global Cucumber, or PDF for Amalgamated Okra.

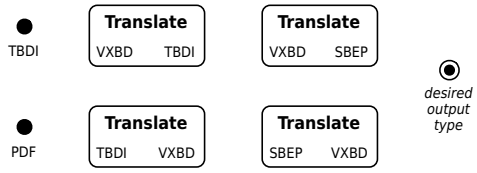


Figure 7.3: The Zucchini transformation graph in the workflow notation

Running the polyadic discovery algorithm on these two cases yields the workflows in Figure 7.4 (shown with the unused transformations removed to reduce clutter). Note that in each case, the compound transformation that results has two input types (the source nodes), and is therefore polyadic. This is true even though, in both cases, only one of the sources is actually used — this does not change the fact that both are available.

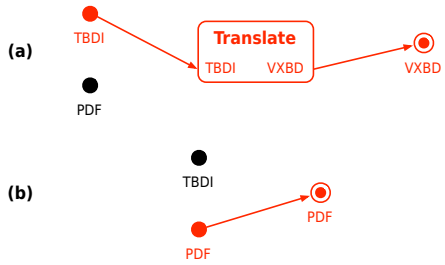


Figure 7.4: Workflow solutions for two Zucchini suppliers

Another important point is that we must still execute the discovery algorithm once for each supplier. It might seem reasonable to follow the same pattern for the destination datatypes as we did for the source datatypes, and to create a single graph with sink nodes for each supplier's purchase order type. However, our definition of a workflow will require that a value must be generated for *every* sink node. If we then execute this resulting workflow for each Zucchini purchase order, then each order will be translated into *every* supplier format. Since the desired behavior is for each purchase order to be translated only into the format for its particular supplier, this is incorrect. Note that we could use multiple sinks if a given purchase order needed to be sent to multiple suppliers; in this case, the purchase order *does* have to be translated into several datatypes. However, we would still need separate sets of sink nodes and separate discovery executions for each distinct combination of suppliers on some purchase order.

Components of a datatype

Polyadic graphs can also be used to include part of the internal structure of a datatype directly into the transformation graph. As we have seen in our case studies, this can be very useful when the structure of a datatype changes often. An example of this involves the image metadata types from Section 4.1.1. Instead of defining a direct conversion transformation between simple metadata and Dublin Core, we could define an *extractor transformation* for simple metadata that has one output for each of the constituent metadata elements, and a *constructor transformation* that can create a Dublin Core record from these elements, as shown in Figure 7.5.

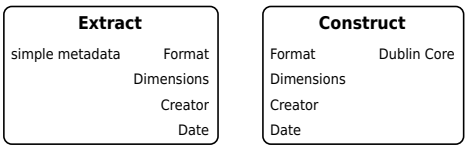


Figure 7.5: Extractor and constructor transformations

At first glance, this might not seem like a more elegant solution, since it requires twice as many transformations to perform the same task. However, the focus of the transformations has shifted slightly: instead of a conversion transformation that must incorporate knowledge of both datatypes, we have two transformations that each only need to know about a single datatype. This can provide more opportunities for transformation reuse; by writing a constructor transformation for a new metadata type, we automatically have the equivalent of a conversion transformation from simple metadata to the new datatype, without having to know or incorporate any knowledge of simple metadata. Further, we get conversion transformations from any other datatype that we have defined an extractor transformation for.

There are two issues with this approach that must be considered. First, the constituent parts must be fully specified datatypes. This means, for instance, that we must decide on some concrete encoding for the metadata elements; since these element datatypes will be shared by all of the extractor and constructor transformations, their encoding should not be tied too closely to any one of the metadata types' encodings.

The second issue concerns the constructor transformations. As mentioned, Dublin Core supports many more metadata elements than the simple metadata type; it seems reasonable to define its constructor transformation with inputs for these extra elements, as shown in Figure 7.6. Unfortunately, doing so would prevent us from transforming from simple metadata to Dublin Core, since polyadic transformations must have *all* of their inputs satisfied before they can be executed.

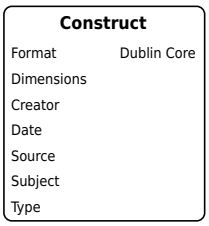


Figure 7.6: A Dublin Core constructor with more metadata inputs

Since we have not yet developed a discovery algorithm for compound transformations in polyadic graphs, it would be tempting to change our requirements and allow polyadic transformations to have optional inputs. Unfortunately, this will make it much more difficult to develop an efficient discovery algorithm. A different solution would be to represent this use case with multiple constructor transformations — one for each possible combination of input elements. Each constructor would still need all of its inputs satisfied before it could be executed; the discovery algorithm would determine the optimal constructor transformation to use, just as it would whenever there were multiple valid transformation paths between two datatypes. This solution would also allow the use of properties on the different constructors to allow for tradeoffs between constructing the data quickly (but possibly with missing elements) or more fully (but possibly requiring more time to obtain the necessary elements). Of course, we would not want to require the user to have to declare each of these constructors individually; this would be cumbersome and repetitive. A language feature like the declaration patterns of Section 4.1.4 could be used to alleviate this.

Merging data from multiple sources

Another situation to consider is when we want to merge data from two or more different sources into a single, more generic, datatype. For example, the OME-XML format from Section 4.1.1 supports a large, extensible, set of metadata elements, different subsets of which overlap with the other metadata types in the transformation graph. For instance, an OME-XML file could store the name of the original image format and the image dimensions from simple metadata, the image source description from Dublin Core, while extracting the color model of the image directly from a TIFF file. OME-XML stores the image metadata in the same file as the image itself, so we could extract the pixel array from the TIFF as well. This yields three sources, all of which are necessary to create a complete OME-XML file. The resulting creation transformation is shown in Figure 7.7.

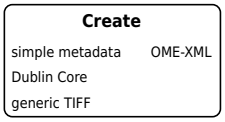


Figure 7.7: A creation transformation for OME-XML with multiple sources

Of course, we can combine this example with the previous one and use extractor and constructor transformations instead. Rather than defining the transformation to specifically use simple metadata, Dublin Core, and generic TIFF as inputs, we would define an OME-XML constructor transformation (shown in Figure 7.8) that lists its component parts as inputs. The discovery algorithm would then be responsible for finding the best way to generate these inputs. It might use simple metadata, Dublin Core, and generic TIFF as we did, yielding the workflow shown in Figure 7.9. If a different solution were more optimal, a different workflow would result, without having to write any new transformations.

Single-use datatypes

The last feature of polyadic graphs that we consider is that of single-use datatypes. Up to this point, we have assumed that the data we are dealing with exists in a form that can be read in any order and

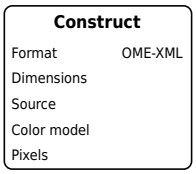


Figure 7.8: An alternative constructor transformation for OME-XML

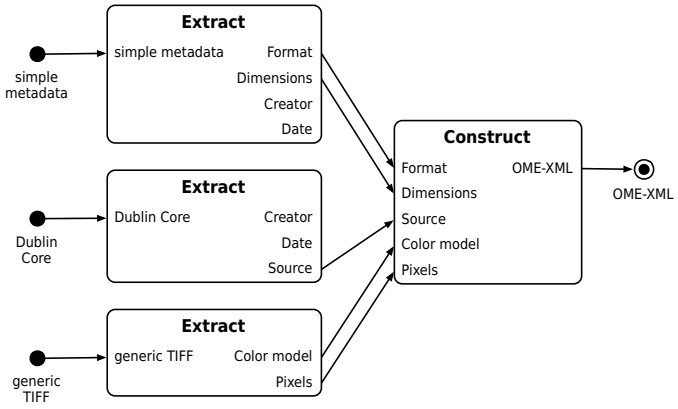


Figure 7.9: The workflow corresponding to the OME-XML creation transformation

multiple times, such as a local file or an internal memory buffer. We might also want to transform data that comes from a streaming source, in which case the data must be processed in order and can only be read once. Properly utilizing a streaming source can lead to large efficiency gains, since the memory overhead needed is constant regardless of the size of the data; this scales extremely well to large datasets.

In a unary graph, this distinction is less important; it cannot change which transformation paths are possible, since the data cannot be used multiple times in a path anyway. The only way that a datatype (and by extension, any instance of that datatype) can appear in a path more than once is due to a cycle, as shown in Figure 7.10. However, a cyclic path such as $\langle \alpha, \beta, \gamma, \delta, \epsilon \rangle$ cannot ever be an optimal transformation, since we can always remove the cycle and instead use $\langle \alpha, \epsilon \rangle$, in which the B datatype does not repeat. Even if we did use the cyclic path as a solution, though, no data would be reused. Rather, there are two instances of datatype B: the first is generated from A by transformation α, the second from D by transformation δ. Each of these instances is used exactly once.

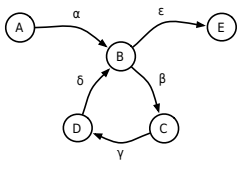


Figure 7.10: A cyclic transformation path

In a workflow, however, it is perfectly acceptable to use a datatype more than once; a simple example is shown in Figure 7.11, where the same instance of datatype A is used as an input to transformations α and β.

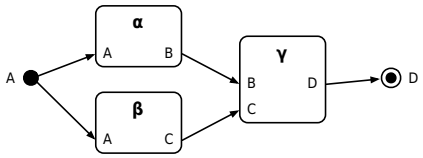


Figure 7.11: A workflow with datatype A used twice

If we want to allow certain datatypes to be limited to a single use, we need to slightly modify the definition of a workflow. For a *single-use datatype*, denoted in a workflow by italicizing it, each source node and transformation output of that type can have at most one outgoing dataflow link. This correctly prevents a datatype from being reused without first being regenerated by some transformation. We can modify the workflow graph to limit datatype A to a single use, as shown in Figure 7.12. In this case, only one of the dashed dataflow links can be used, and by extension, only one of transformations α and β can be executed. Since transformation γ requires the outputs of both α and β, there is no longer a valid workflow solution from $\{A\}$ to $\{D\}$.

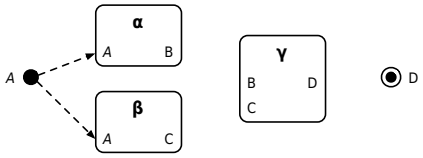


Figure 7.12: No workflows are possible when datatype A is limited to a single use

7.1.2 Workflow notation

Having presented several examples of polyadic transformation graphs, we can now more precisely describe what constitutes a valid transformation workflow.

A workflow represents a compound transformation; the workflow solution only includes the atomic transformations that are actually executed as part of the compound transformation. As mentioned previously, each atomic transformation is represented as a "black box" of logic or code. Unlike the graph notation, the workflow notation does not represent the underlying datatypes directly. This means that we must introduce special *source* and *sink* nodes for the given and desired datatypes. There are then *dataflow links* that show the flow of data through the transformation graph. Each dataflow link can receive data from a source node or from an output of one of the workflow's executed transformations. Each link can provide data to a sink node or to the input of a transformation. Each transformation that is executed as part of a workflow must have all of its inputs satisfied by an incoming dataflow link. No transformation input can have more than one dataflow link providing it data. Each sink node must also be satisfied by (exactly one) incoming link. Finally, each dataflow link must be *well-typed*: both sides must be of the same type.

For example, $\langle \beta, \delta \rangle$ is a valid compound transformation between datatypes A and D in the example graph from Figure 7.1. This solution is shown in Figure 7.13 as a unary path and in Figure 7.14 as a workflow.

We can use the dataflow links in a workflow to infer an order of execution for the graph's transformations. Before any particular transformation can be executed, we must have already generated

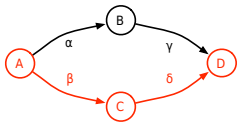


Figure 7.13: A compound transformation represented as a path

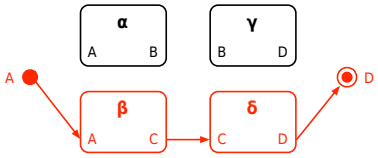


Figure 7.14: The same compound transformation represented as a workflow

values for each of its inputs. If an input receives its value from an output of another transformation, then the transformation that generates the value must obviously be executed before the transformation that uses the value. By looking at all of the links, we can piece together an overall *execution order* for the workflow. Note that this might not fully specify the order of execution; consider, for instance, the workflow shown in Figure 7.15. Transformation α must execute before β since it generates the instance of datatype B that β uses as an input; similarly, α must execute before γ. However, there is nothing that requires β to execute before γ, or vice versa. Therefore, this workflow has two valid execution orders: $\langle \alpha, \beta, \gamma \rangle$ and $\langle \alpha, \gamma, \beta \rangle$. Thus, the execution orders of a workflow, which can be found using a simple topological sort, induce a partial order on the transformations in the workflow.

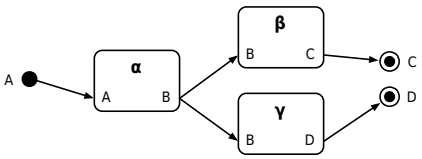


Figure 7.15: A workflow with many possible execution orders

It is also possible for a workflow to have *no* valid executions. For instance, a workflow might contain *cycles*, as shown in Figure 7.16. None of the workflow rules described above are violated, since each of α's inputs has an incoming dataflow link. However, because of the link between α's C output and its C input, any valid execution order would require that α execute before itself, which is obviously not possible. Since they are clearly not very useful, we will not consider workflows that do not have valid execution orders.

Finally, while it is useful to think of transformations as units of computation, and to represent compound transformations as workflows, it is important to realize that not all computations are valid transformations. Even though transformation graphs are generic enough to be used with other kinds of computation (assuming that they can be specified by an inductive property), we must remain aware of the limitations of transformations when we use this technique to solve the data mismatch problem. The most important distinction is that, as mentioned in Chapter 3, transformations are defined in terms of data equivalence. They deal with pre-existing data, and generate new representations or encodings of that data that are equivalent to the originals according to some criteria that are important to the application. Fundamentally new data cannot be created by a pure transfor-

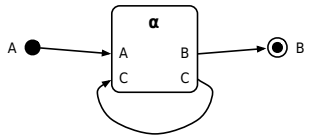


Figure 7.16: A workflow containing a cycle

mation. We assume that this property also holds when the transformation is polyadic — all of the inputs and outputs of a transformation must be representations of equivalent data. Because of this, we further assume that a transformation cannot have two inputs (or outputs) of the same datatype, since this would only provide two indistinguishable copies of the same data representation. As stated in Section 3.1.3, any important, distinguishable difference must be modeled with separate datatypes.

7.2 Polyadic discovery

The workflow notation from the previous section allows us to intuitively describe several example polyadic transformation graphs. However, the notation does not lend itself to an obvious discovery algorithm. In this section, we present a new set-based notation that allows us to again use a simple pathfinding algorithm to discover compound transformations.

7.2.1 Set notation

We begin by introducing a new *set notation* for representing transformation graphs. The idea behind this notation is to keep track of the set of datatypes that are available at each step of a compound transformation. To do this, we create a graph that contains a node for each subset of the graph's datatypes. As with the original unary graph notation, each edge represents a transformation; however, since there might be many situations where a transformation is eligible for execution, each transformation will likely have many edges in a set notation graph.

As an example, we can consider the graph shown in Figure 7.17. Part (a) shows the graph in the workflow notation, while part (b) shows its equivalent representation in the set notation. This graph defines four datatypes — A, B, C, and D — which requires $2^4 = 16$ nodes in the set notation, one for each subset of datatypes. We then add edges for each transformation. A transformation is eligible for execution when we have values for all of its inputs, so we add an edge starting from any node that is a superset of the transformation's inputs. After the transformation has executed, we have available all of its output datatypes, in addition to all of the datatypes that were available previously. (For now, we assume that all datatypes are reusable.) Therefore, each edge is connected to the union of its source and its transformation's outputs. It is possible that no new datatypes will be introduced, as we might already have values for each of the transformation's output datatypes. This will cause a self-cycle: the source and sink of that particular edge will be the same node. Even though such a self-cycle will never be included in a shortest path, we include them in the set notation to simplify our definitions. However, to remove clutter, they will not be displayed visually.

To illustrate these construction rules we can examine transformation β. This transformation has a single input of type B. There are eight nodes in the graph that include B. We add edges from $\{B\}$ to $\{B, C\}$, from $\{A, B\}$ to $\{A, B, C\}$, from $\{B, D\}$ to $\{B, C, D\}$, and from $\{A, B, D\}$ to $\{A, B, C, D\}$. In each

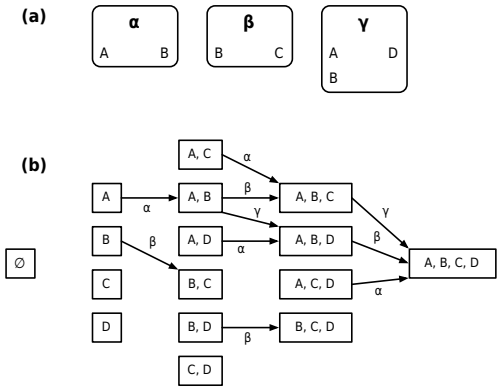


Figure 7.17: A transformation graph in the (a) workflow and (b) set notations

case we add C, the transformation's output, to the set of previously available datatypes. Technically, there will also be self-cycles at $\{B, C\}$, $\{A, B, C\}$, $\{B, C, D\}$, and $\{A, B, C, D\}$; however, since they already contain the only type that would be created by the transformation, they are not shown.

With the structure of a transformation graph in place, we can now turn our attention to compound transformations. Since we are including datatypes in the graph directly, we do not need to introduce special source and sink nodes as in the workflow notation; instead, we will use some of the existing nodes as sources and sinks. We define the *source node*, which we denote with a thick border, to be the node with the specific set of datatypes that we are given to start with. We define several *sink nodes*, which we denote with a double border, each of which is a superset of the datatypes that we desire. There is only one source node since we know exactly which datatypes we are provided with; on the other hand, there are many sink nodes since there are many possible ways to provide the desired output types. With the source and sink nodes defined in this way, a compound transformation is simply a path from the source node to any one of the sink nodes. Figure 7.18 shows a compound transformation between $\{A\}$ and $\{C, D\}$ in both notations. Note that there are two possible paths in the set notation, just like there are two valid execution orders for the workflow: β and γ can be executed in any order.

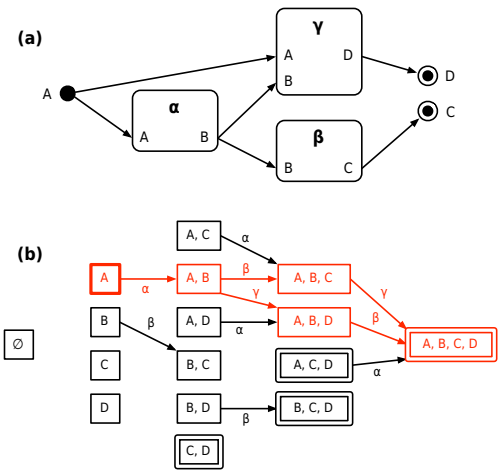


Figure 7.18: Compound transformations in both notations

There is another slight difference between the workflow and set notations: in a workflow, a single datatype instance can sometimes be obtained from several separate possible locations, a distinction that the set notation cannot reproduce. For instance, Figure 7.19 shows two distinct workflows that have the same representation in the set notation. Transformation β requires an input of type A; this input can either be provided by the output of transformation α, or by the instance of A that we are given. Regardless of which A is used, $\langle \alpha, \beta \rangle$ is the only possible solution. However, as mentioned in Section 3.1.3, every important difference between data instances must be modeled using separate datatypes. Thus, it does not matter which A instance we use for β's input; if there was an important difference between the two, they would have to be separate datatypes. Therefore, this does not affect the correctness of our discovery algorithm; if there is more than one possible workflow, we choose one arbitrarily.

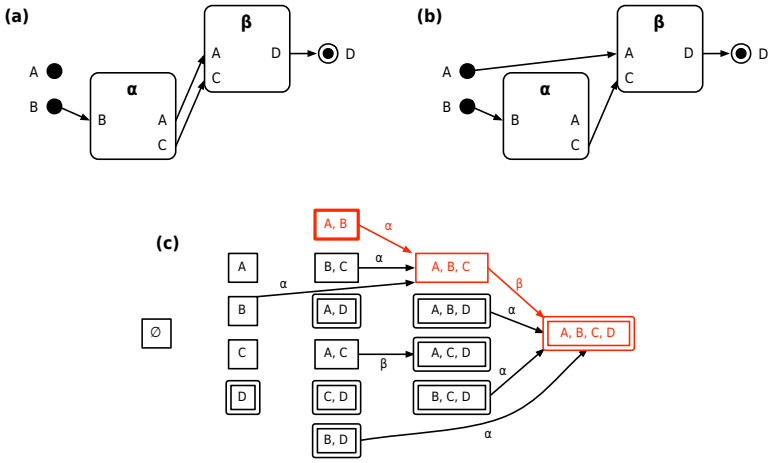


Figure 7.19: Two workflows with the same set notation equivalent

7.2.2 Discovery algorithm

Having defined the workflow and set notations for polyadic graphs, we can now describe an algorithm for discovering polyadic compound transformations. In the next section, we will show that this algorithm is correct, by showing that the two notations are equivalent — specifically, that a solution found in the set notation has an equivalent solution in the workflow notation, and vice versa.

The polyadic discovery algorithm is fairly straightforward:

1. Translate the polyadic transformation graph into the set notation.
2. Find a shortest path from the set graph's source node to *any* of its sink nodes.
3. Use this path to add dataflow links to the transformation graph's workflow.

The first step is simple, given the rules from Section 7.2.1 for creating a valid set graph. First, we create a node for every subset of datatypes in the transformation graph. Then, for each transformation, we identify the nodes that satisfy the transformation's inputs — i.e., that contain a superset of the transformation's input types. For each of these, we add an edge representing the transformation.

The edge starts at the node we have just identified, and ends at the node representing the union of the source node (the datatypes we had before executing the transformation) and the transformation's outputs (the datatypes created by the transformation). After adding the correct edges for each transformation, we mark the source and sink nodes of the set graph. There is one source node, containing the datatypes that we start with. There are many sink nodes, one for each superset of the desired output datatypes.

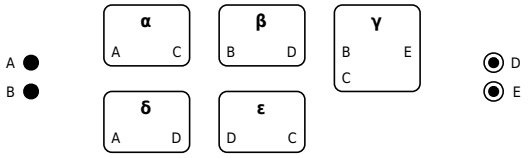


Figure 7.20: An example transformation graph for the discovery algorithm

As an example, we can examine the transformation graph shown in Figure 7.20 using the workflow notation. This graph contains five datatypes and five atomic transformations. The equivalent set notation graph is shown in Figure 7.21. We require thirty-two nodes in the set graph, one for every subset of the five datatypes. We then add edges for each of the five transformations — for example, an edge labeled γ from $\{A, B, C, D\}$ to $\{A, B, C, D, E\}$, since $\{A, B, C, D\}$ satisfies all of γ's inputs, and γ then additionally generates an instance of E. Finally, we mark $\{A, B\}$ as the set graph's source node, and mark every superset of $\{D, E\}$ as a sink node.

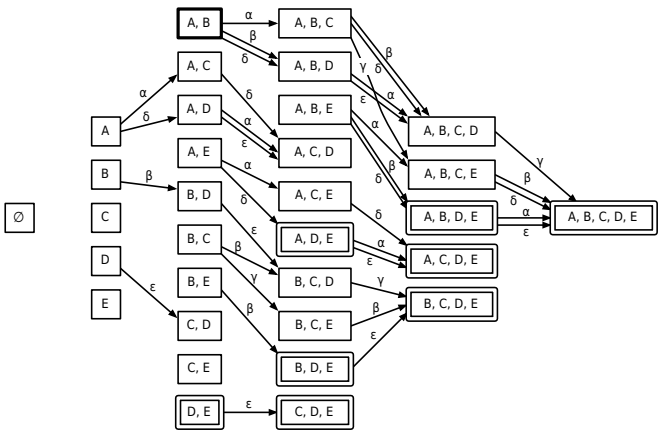


Figure 7.21: The corresponding set graph

As mentioned earlier, we can make several slight optimizations to the set graph. First, we do not have to include a node for the empty set of datatypes: since every transformation must have at least one input, and transformations cannot reduce the number of datatypes we have available, the $\varnothing$ node cannot ever be reachable. Second, we do not have to include any self-loops in the set graph, since they cannot possibly appear in a shortest path. (However, we *will* include these self-loops in the formalism that appears later, to simplify the specification and our proofs.)

With this set graph defined, we can see that $\{A, B, C, D, E\}$ is the only sink node reachable from the source. There are eight possible paths that connect $\{A, B\}$ to $\{A, B, C, D, E\}$, all of length three:

$\langle \alpha, \beta, \gamma \rangle$ $\langle \alpha, \delta, \gamma \rangle$ $\langle \beta, \epsilon, \gamma \rangle$ $\langle \delta, \epsilon, \gamma \rangle$
$\langle \alpha, \gamma, \beta \rangle$ $\langle \alpha, \gamma, \delta \rangle$
$\langle \beta, \alpha, \gamma \rangle$ $\langle \delta, \alpha, \gamma \rangle$

These eight possibilities stem from several decisions that can be made. First, we can use either α or ϵ to generate an instance of C. Similarly, we can use either β or δ to generate an instance of D. Finally, for two of the outcomes, the transformations can be executed in multiple orders. As with unary transformation graphs, we can use properties to assign numeric weights to each atomic transformation, which might cause one of the combinations to be more optimal. However, this will not eliminate any nondeterminism if an optimal combination can be executed in multiple orders.

The final step in the discovery algorithm is to construct a workflow from one of the optimal set paths. (If there are many optimal paths, as is the case in this example, we can choose one arbitrarily.) We start with an empty workflow — one that contains only the source and sink nodes, and no transformations. We then step through the set path; as we encounter each transformation in the path, we add it to the workflow, also adding dataflow links to satisfy all of its inputs. Finally, after adding all of the set path's transformations, we add dataflow links to satisfy each of the sink nodes. As we will show in the next section, we will always be able to satisfy each atomic transformation's inputs when it is added to the workflow, assuming that the set path is a valid compound transformation.

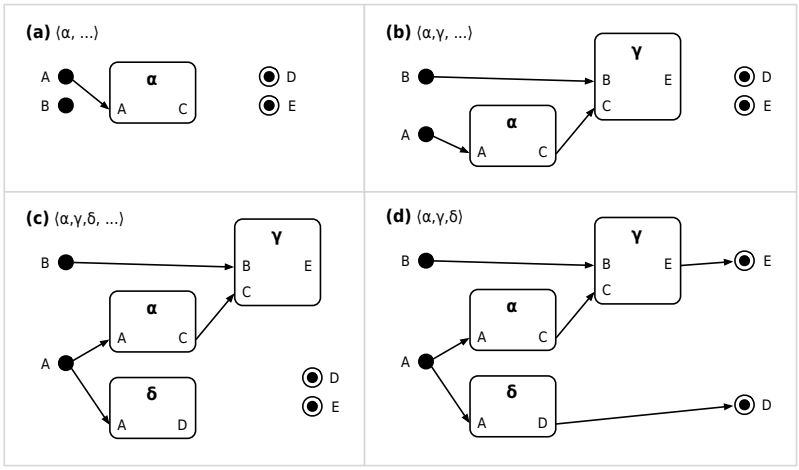


Figure 7.22: Constructing a workflow from a set path

We can illustrate this process on the set path $\langle \alpha, \gamma, \delta \rangle$, as shown in Figure 7.22. We first add α to the workflow, connecting its input to the A source node. Next, we add γ, connecting its inputs to the B source node and α's C output, respectively. Next, we add δ, connecting its input to the A source node. This is the last transformation in the set path; finally, we connect the workflow's sink nodes to δ's D output and γ's E output, respectively.

We can repeat this process for all eight optimal set paths. Combinations of transformations that can be executed in multiple orders only result in a single workflow, since the workflow notation hides the explicit ordering of transformations. This gives us the four workflows shown in Figure 7.23. Any one of these workflows is an optimal compound transformation between $\{A, B\}$ and $\{D, E\}$.

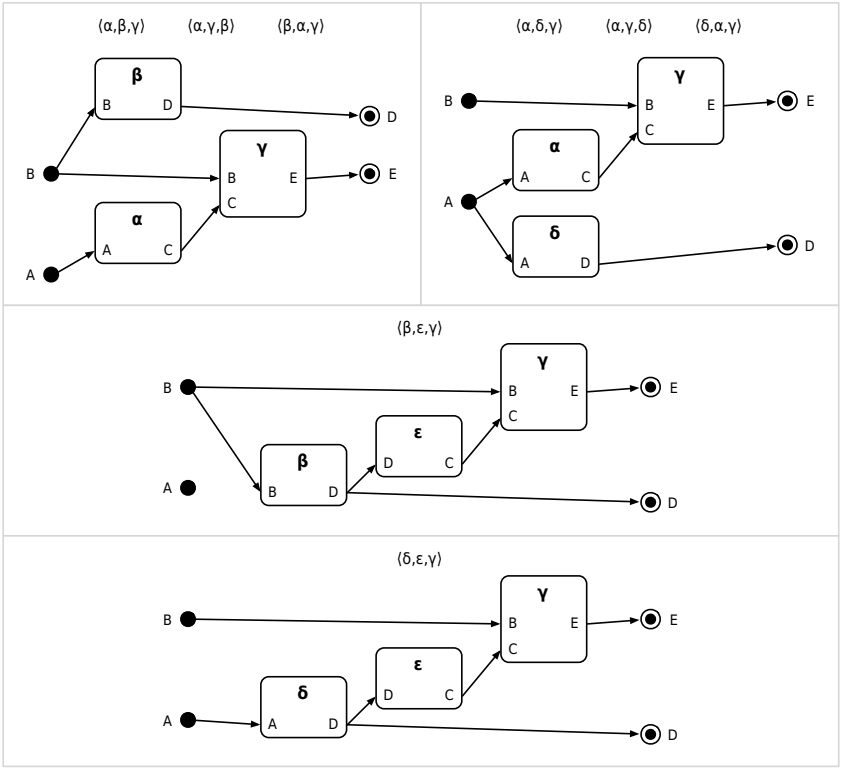


Figure 7.23: The four possible workflow solutions

7.3 Correctness of the algorithm

Having described an algorithm for discovering polyadic compound transformations, we would now like to show that the algorithm is correct. To do this, we will first develop formal descriptions of polyadic transformation graphs, and of the workflow and set notations. We will then show that the notations are *equivalent* — that we can translate between the two notations without losing any important information. This will allow us to show that the optimal set path found by the discovery algorithm can be correctly translated into an analogous optimal workflow.

7.3.1 Polyadic transformation graphs

Our first task is to formalize the notion of a polyadic transformation graph, independent of the notation used to visualize and reason about it. This will be very similar to the formalization for unary graphs defined in Section 4.3; the only real difference is that the atomic transformations in the graph can have multiple inputs and outputs.

A *polyadic transformation* is an atomic transformation that allows for multiple input and output datatypes. It is modeled using a schema similar to the *AtomicTransformation* schema, with the input and output components defined to be *sets* of identifiers rather than single identifiers.

Definition 7.1 *(Polyadic transformations).*

PolyadicTransformation

$$
\begin{array}{l}
name : Identifier \\
inputs : \mathbb{P}\ Identifier \\
outputs : \mathbb{P}\ Identifier \\
properties : Properties
\end{array}
$$

A *polyadic graph* is a transformation graph whose transformations can be polyadic. Its definition is identical to the *Graph* schema (Definition 4.7), except that *PolyadicTransformation* is used instead of *AtomicTransformation*.

Definition 7.2 *(Polyadic graphs).*

PolyadicGraph

$$
\begin{array}{l}
datatypes : Identifier \pfun Datatype \\
transformations : Identifier \pfun PolyadicTransformation \\
\hline
\forall\ id : \mathrm{dom}\ datatypes \bullet datatypes\,(id).name = id \\
\forall\ id : \mathrm{dom}\ transformations \bullet transformations\,(id).name = id
\end{array}
$$

7.3.2 Workflow notation

With polyadic graphs defined, we can now turn our attention to the workflow notation. A *dataflow link* connects elements of a workflow. Each link starts at one of the workflow's source nodes (denoted by source), or at the output of one of the graph's transformations. It ends at one of the workflow's sink nodes (denoted by sink), or at the input of one of the graph's transformations. If it starts or ends at a transformation, we only need to specify the transformation's name, since transformations can only have a single input or output of the appropriate type. Links also have a datatype; later on we will use this to ensure that all of the links in a workflow are well-typed.

Definition 7.3 *(Dataflow links).*

$$LinkSource ::= \mathsf{source} \mid Identifier$$
$$LinkSink ::= \mathsf{sink} \mid Identifier$$

DataflowLink

$$
\begin{array}{l}
from : LinkSource \\
to : LinkSink \\
datatype : Identifier
\end{array}
$$

A *workflow* is a view of a polyadic graph based on units of computation. To simplify things, we will define the *Workflow* schema in parts.

Definition 7.4 *(Workflows).* First, we specify the structure of a workflow, which consists of *executions* and *dataflow links*. An execution is an instance of one of the graph's transformations; each transformation can be executed at most once, so we can model this with a set of transformation names. We also specify the workflow's set of *source* and *sink nodes*, each of which is one of the graph's datatypes.

WorkflowStructure

$$
\begin{array}{l}
sources : \mathbb{P}\ Identifier \\
sinks : \mathbb{P}\ Identifier \\
executions : \mathbb{P}\ Identifier \\
links : \mathbb{P}\ DataflowLink
\end{array}
$$

We must include several structural constraints to ensure that the workflow is valid. First, we ensure that each source and sink node's datatype, each link's datatype, and each execution's transformation, are all actually defined in the graph.

WorkflowDefinedness

$$
\begin{array}{l}
WorkflowStructure \\
PolyadicGraph \\
\hline
sources \subseteq \mathrm{dom}\ datatypes \\
sinks \subseteq \mathrm{dom}\ datatypes \\
executions \subseteq \mathrm{dom}\ transformations \\
\forall\ l : links \bullet l.datatype \in \mathrm{dom}\ datatypes
\end{array}
$$

Further, if a link starts at a source node, then that source node must actually be defined in the workflow. If it starts at a transformation, then the transformation must be one that is executed by the workflow, and it must have an output of the appropriate datatype.

ValidLinkSources

$$
\begin{array}{l}
WorkflowStructure \\
PolyadicGraph \\
\hline
\forall\ l : links \bullet \\
\quad \textbf{if}\ l.from = \mathsf{source}\ \textbf{then} \\
\quad\quad l.datatype \in sources \\
\quad \textbf{else} \\
\quad\quad l.from \in executions \land \\
\quad\quad l.datatype \in transformations\,(l.from).outputs
\end{array}
$$

Similarly, if a link ends at a sink node, then that sink node must actually be defined in the workflow. If it ends at a transformation, then the transformation must be one that is executed by the workflow, and it must have an input of the appropriate datatype.

ValidLinkSinks

$$
\begin{array}{l}
WorkflowStructure \\
PolyadicGraph \\
\hline
\forall\ l : links \bullet \\
\quad \textbf{if}\ l.to = \mathsf{sink}\ \textbf{then} \\
\quad\quad l.datatype \in sinks \\
\quad \textbf{else} \\
\quad\quad l.to \in executions \land \\
\quad\quad l.datatype \in transformations\,(l.to).inputs
\end{array}
$$

We must ensure that each transformation input in the graph has at most one incoming dataflow link.

NoConflictingInputs

WorkflowStructure
PolyadicGraph

$$\forall\, xform : \text{ran}\ transformations \bullet \forall\, input : xform.inputs \bullet \#\{\, l : links \mid l.to = xform.name \land l.datatype = input \,\} \leq 1$$

Similarly, we must ensure that each sink node in the workflow has exactly one incoming dataflow link.

NoConflictingSinks

WorkflowStructure
PolyadicGraph

$$\forall\, s : sinks \bullet \#\{\, l : links \mid l.to = \mathsf{sink} \land l.datatype = s \,\} = 1$$

Finally, we must ensure that each executed transformation has an incoming dataflow link for all of its inputs.

AllInputsSatisfied

WorkflowStructure
PolyadicGraph

$$\forall\, xform : executions \bullet \forall\, input : transformations\,(xform).inputs \bullet \exists\, l_{IN} : links \bullet l_{IN}.to = xform \land l_{IN}.datatype = input$$

We then conjoin all of these different constraints together to define the *Workflow* schema.

Workflow

WorkflowDefinedness
ValidLinkSources
ValidLinkSinks
NoConflictingInputs
NoConflictingSinks
AllInputsSatisfied

7.3.3 Set notation

Next we can provide a similar formalization for the set notation. We will define this specification in two parts: the first to represent the basic structure of a set graph, and the second to encode a compound transformation found by the discovery algorithm.

Graph structure

Our first task is to define how the datatypes and atomic transformations in a polyadic transformation graph are encoded as nodes and edges in a set graph. We start by defining the nodes: each node in a set-based graph represents a set of datatypes.

Definition 7.5 *[Set-based nodes].*

SetNode

$$datatypes : \mathbb{P}\ Identifier$$

Each edge in a set-based graph represents one transformation step: given one set of available datatypes (*from*), we execute the given polyadic transformation (*transformation*) to reach a new set of datatypes (*to*).

Definition 7.6 *(Set-based edges).*

SetEdge

$$from : \mathbb{P}\ Identifier$$
$$to : \mathbb{P}\ Identifier$$
$$transformation : Identifier$$

A *set-based graph* is a view of a polyadic graph based on sets of datatypes. Like the *Workflow* schema, there are many constraint clauses in *SetGraph*, which we will examine in turn.

Definition 7.7 *(Set-based graphs).* First, we define the components of a set-based graph. A set-based graph consists of a set of *nodes* and a set of *edges*, represented by the *SetNode* and *SetEdge* schemas, respectively.

SetGraphStructure

$$nodes : \mathbb{P}\ SetNode$$
$$edges : \mathbb{P}\ SetEdge$$

We also include constraints to ensure that each node's datatype set, each edge's source and destination, and each edge's transformation are all defined in the graph.

SetGraphDefinedness

SetGraphStructure
PolyadicGraph

$$\forall\ n : nodes \bullet n.datatypes \subseteq \mathrm{dom}\ datatypes$$
$$\forall\ e : edges \bullet$$
$$(\exists\ n : nodes \bullet n.datatypes = e.from) \land$$
$$(\exists\ n : nodes \bullet n.datatypes = e.to) \land$$
$$e.xform \in \mathrm{dom}\ transformations$$

Next we define a consistency constraint for edges. Each edge must satisfy two conditions. First, we must be able to execute the edge's transformation; this means that the edge's source node (*e.from*) must contain at least all of the datatypes needed by the edge's transformation. Second, we ensure that all of the datatypes created by the transformation are in the edge's destination node, in addition to all the datatypes that were previously available in the source node.

SetEdgeInputsSatisfied

SetGraphStructure
PolyadicGraph

$$\forall\ e : edges \bullet$$
$$transformations\,(e.transformation).inputs \subseteq e.from$$

SetEdgeOutputsCreated

SetGraphStructure
PolyadicGraph

$$\forall\, e : edges \bullet \\ e.to = e.from \cup transformations\,(e.transformation).outputs$$

Finally, we define two completeness constraints. The first states that there is a node for every possible subset of the datatypes defined by the graph. The second states that whenever a source set contains all of the necessary inputs for a transformation, then there is an edge for that transformation from that source set.

SetNodeCompleteness

SetGraphStructure
PolyadicGraph

$$nodes = \{\, ids : \mathbb{P}\,\mathrm{dom}\, datatypes \bullet (\!|\, datatypes \rightsquigarrow ids \,|\!) \,\}$$

SetEdgeCompleteness

SetGraphStructure
PolyadicGraph

$$\forall\, id : \mathrm{dom}\, transformations;\ source : \mathbb{P}\,\mathrm{dom}\, datatypes \mid \\ transformations\,(id).inputs \subseteq source \bullet \\ \exists\, e : edges \bullet e.from = source \land e.transformation = id$$

With each of these constraint schemas defined, we can construct the overall *SetGraph* schema to be the conjunction of them.

SetGraph

PolyadicGraph
SetGraphStructure
SetGraphDefinedness
SetEdgeInputsSatisfied
SetEdgeOutputsCreated
SetNodeCompleteness
SetEdgeCompleteness

Compound transformations

Next we must describe the compound transformations found by the discovery algorithm. A *set path* is a directed path through a set graph. We will define the *SetPath* schema in parts.

Definition 7.8 *(Set paths).* First, we consider the overall structure of a set path: it consists of a source node and sink node, and the sequence of set edges that comprise the path. We also associate the set path with the set graph that contains it; we must therefore ensure that any edges or nodes mentioned within the path actually exist in the graph.

SetPathStructure

$$
\begin{array}{l}
graph : SetGraph \\
source : SetNode \\
sink : SetNode \\
edges : \mathrm{seq}\ SetEdge \\
\hline
source \in graph.nodes \\
sink \in graph.nodes \\
\mathrm{ran}\ edges \subseteq graph.edges
\end{array}
$$

Next we ensure that the path's source and sink are consistent with its sequence of edges. Specifically, the first edge in the path must start from the path's source, and the last edge in the path must end at the path's sink. If the path is empty (contains no edges), than its source and sink must be identical; this implies that there can be many distinct empty paths within a set graph — one for each node.

SetPathSourceSinkConsistency

$$
\begin{array}{l}
SetPathStructure \\
\hline
\textbf{if}\ \#edges = 0\ \textbf{then} \\
\quad source = sink \\
\textbf{else} \\
\quad (head\ edges).from = source.datatypes \land \\
\quad (last\ edges).to = sink.datatypes
\end{array}
$$

We must also ensure that the edges are consistent with themselves; each edge must end where the following edge begins.

SetPathEdgeConsistency

$$
\begin{array}{l}
SetPathStructure \\
\hline
\forall\, i : 1 \mathinner{..} (\#edges - 1) \bullet edges\,(i).to = edges\,(i+1).from
\end{array}
$$

We can now define the *SetPath* schema as the conjunction of these constraint schemas.

SetPath

$$
\begin{array}{l}
SetPathStructure \\
SetPathSourceSinkConsistency \\
SetPathEdgeConsistency
\end{array}
$$

7.3.4 Equivalence of the two notations

With formal specifications for the workflow and set notations, we can now show that the two notations are equivalent. Our overall strategy for this proof will involve three phases. First, we examine the workflow notation, and provide a precise description of the *execution orders* and *traces* of a workflow. Second, we examine the set notation, and show that set graphs and set paths are *deterministic*. Finally, we will exploit this determinism to show how the relationship between workflow executions and set paths links the two notations.

Execution orders and workflow traces

A workflow defines the set of atomic transformations that need to be executed for a compound transformation, but does not explicitly state the order that they should be executed in. However, the workflow's dataflow links allow us to infer a valid order of execution for the transformations.

A sequence of transformations is an *execution order* of a workflow if two conditions are met. First, the sequence must contain exactly those transformations executed by the workflow. Second, for any dataflow link that connects two transformations, the link's source must precede the link's destination. We can then show that this definition ensures that every transformation input will have a value available when the transformation is executed. Since we have already stated that workflows cannot contain multiple executions of a given transformation, we only consider injective sequences (i.e., those with no duplicates).

Definition 7.9 *(Execution orders).*

$$Execution == \text{iseq } Identifier$$

$$\_ \text{ executes } \_ : Execution \leftrightarrow Workflow$$

$$\forall\, exec : Execution;\ w : Workflow \bullet$$
$$exec \text{ executes } w \Leftrightarrow$$
$$\text{ran } exec = w.executions \land$$
$$\forall\, l : w.links \mid l.from \neq \text{source} \land l.to \neq \text{sink} \bullet$$
$$exec^{\sim}(l.from) < exec^{\sim}(l.to)$$

For convenience, we also provide a function that returns all of the valid execution orders for a workflow.

$$\text{executionsOf } \_ : Workflow \rightarrow \mathbb{P}\, Execution$$

$$\text{executionsOf } w = \{\, exec : Execution \mid exec \text{ executes } w \,\}$$

It is possible for a workflow to have more than one valid execution order. However, as we have mentioned previously, we do not want to consider workflows, like the cyclic one in Figure 7.16, that have no valid execution order.

Definition 7.10 *(Executable workflows).* A workflow is *executable* if there is at least one valid execution order for it.

$$ExecutableWorkflow == \text{ran executes}$$

An execution order tells us one possible sequence of transformations for executing a workflow. It will also be important to know which datatypes are available at each step of the execution's sequence. The *trace* of a workflow execution determines which datatypes are available at each step of the corresponding compound transformation. The trace starts with the workflow's source, since these are the only datatypes that are given. Each successive element of the trace adds the datatypes created by the corresponding transformation. Each step is represented by a bag, since it is possible that a datatype will have been generated more than once.

Definition 7.11 *(Workflow traces).*

$$TraceElement == \text{bag } Identifier$$
$$Trace == \text{seq } TraceElement$$

$$\begin{array}{|l} \text{trace}\_\ [\_] : (Execution \times Workflow) \pfun Trace \\ \hline \text{dom trace} = \text{executes} \\ \forall\ exec : Execution;\ w : Workflow \mid exec \text{ executes } w \bullet \\ \quad \#(\text{trace}_w\ [exec]) = (\#exec) + 1 \land \\ \quad \text{trace}_w\ [exec](1) = \text{bag } w.sources \land \\ \quad \text{trace}_w\ [exec](i+1) = \\ \qquad \text{trace}_w\ [exec](i) \uplus \text{bag } w.transformations\,(exec\,(i)).outputs \end{array}$$

With these definitions in place, we can prove that every execution order is *valid*: that it ensures that each transformation input has a value when the transformation is executed. To prove this validity, we must show two things. First, we must show that each datatype appears in an execution trace immediately after it is generated. This, coupled with the monotonicity of execution traces, allows us to prove that each datatype is also available immediately before it is used.

Theorem 7.12 (Traces consistent with link sources). *For every dataflow link in a workflow, the link's datatype appears in every execution trace immediately after the link's source generates the datatype. For links that start at a source node, the datatype is available immediately, and must appear in the first element of the trace. For links that start at a transformation, the datatype appears in the trace immediately after the source transformation is executed.*

$$\begin{array}{l} \forall\ w : Workflow \bullet \forall\ l : w.links;\ exec : \text{executionsOf } w \bullet \\ \quad \textbf{if } l.from = \text{source } \textbf{then} \\ \qquad l.datatype \in (head\ \text{trace}_w\ [exec]) \\ \quad \textbf{else} \\ \qquad l.datatype \in \text{trace}_w\ [exec](exec^{\sim}(l.from) + 1) \end{array}$$

Proof. We prove this separately for links that start at a source node and links that start at a transformation.

Links from a source node. If the link begins at one of the workflow's source nodes (signified when $l.from = \text{source}$), then the *ValidLinkSources* constraint from Definition 7.4 tells us that the link's datatype must be one of the workflow's source datatypes:

$$l.datatype \in w.sources$$

Definition 7.11 tells us that the first element of an execution trace is the bag containing the workflow's source datatypes:

$$\text{trace}_w\ [exec](1) = \text{bag } w.sources$$

Therefore, the link's datatype is obviously a member of the first trace element:

$$l.datatype \in \text{trace}_w\ [exec](1)$$
$$l.datatype \in (head\ \text{trace}_w\ [exec])$$

Links from a transformation. If the link begins at a transformation (signified when $l.from \neq$

source), then the *ValidLinkSources* constraint from Definition 7.4 tells us that the link's datatype must be an output of the link's source transformation:

$$l.datatype \in w.transformations\,(l.from).outputs$$

Since we know that the datatype is an element of the transformation's output set, we can trivially say that it is a member of the corresponding bag:

$$l.datatype \sqsubseteq \mathsf{bag}\ w.transformations\,(l.from).outputs$$

We know that the link's source transformation was executed in position $exec^{\sim}(l.from)$. Definition 7.11 tells us the contents of the trace immediately after this step.

$$\begin{array}{l}\mathsf{trace}_w\,[exec](exec^{\sim}(l.from) + 1) = \\ \quad \mathsf{trace}_w\,[exec](exec^{\sim}(l.from)) \uplus \\ \quad\quad \mathsf{bag}\ w.transformations\,(exec\,(exec^{\sim}(l.from))).outputs\end{array}$$

This can be simplified to

$$\begin{array}{l}\mathsf{trace}_w\,[exec](exec^{\sim}(l.from) + 1) = \\ \quad \mathsf{trace}_w\,[exec](exec^{\sim}(l.from)) \uplus \\ \quad\quad \mathsf{bag}\ w.transformations\,(l.from).outputs\end{array}$$

Since we have already shown that $l.datatype$ is a member of the bag union's second operand, we know that it must also be a member of the union as a whole:

$$l.datatype \sqsubseteq \mathsf{trace}_w\,[exec](exec^{\sim}(l.from) + 1)$$

This proves both branches of the **if** statement, thus proving the theorem.

□

Theorem 7.13 (Traces consistent with link sinks). *For every dataflow link in a workflow, the link's datatype appears in every execution trace immediately before the link's sink consumes the datatype. For links that end at a sink node, the datatype is not consumed until the end of the execution, and must appear in the last element of the trace. For links that end at a transformation, the datatype must appear in the trace immediately before the sink transformation is executed.*

$$\begin{array}{l}\forall\ w : Workflow \bullet \forall\ l : w.link;\ exec : \mathsf{executionsOf}\ w \bullet \\ \quad \textbf{if}\ l.from = \mathsf{sink}\ \textbf{then} \\ \quad\quad l.datatype \sqsubseteq (last\ \mathsf{trace}_w\,[exec]) \\ \quad \textbf{else} \\ \quad\quad l.datatype \sqsubseteq \mathsf{trace}_w\,[exec](exec^{\sim}(l.to))\end{array}$$

Proof. We prove this separately for links that end at a sink node and links that end at a transformation.

Links to a sink node. Let $l.to = \mathsf{sink}$. According to the *ValidLinkSinks* constraint from Definition 7.4, the link's datatype must be one of the workflow's sink nodes:

$$l.datatype \in w.sink$$

Theorem 7.12 tells us that this datatype also must appear in some element of the execution trace. (If

the link starts at a source node, it will appear in the trace's first element; otherwise, it will appear immediately after the link's source transformation is executed.)

$$\exists\, i : \operatorname{dom} \mathsf{trace}_w\,[exec] \bullet l.datatype \sqsubseteq \mathsf{trace}_w\,[exec](i)$$

Execution traces are monotonic, so if the datatype appears at any point in the trace, it must also appear in every subsequent step — specifically, the last step.

$$l.datatype \sqsubseteq (last\ \mathsf{trace}_w\,[exec])$$

Links from a source node to a transformation. Let $l.to \neq \mathsf{sink}$ and $l.from = \mathsf{source}$. Since the link starts at one of the workflow's source nodes, Theorem 7.12 tells us that its datatype must appear in the first element of the execution trace:

$$l.datatype \sqsubseteq (head\ \mathsf{trace}_w\,[exec])$$

The indices of a sequence all are at least one; specifically, the index of the link's destination transformation in the execution order must be at least one:

$$1 \leq exec^{\sim}(l.to)$$

Execution traces are monotonic, so if the datatype appears at any point in the trace, it must also appear in every subsequent step:

$$l.datatype \sqsubseteq \mathsf{trace}_w\,[exec](exec^{\sim}(l.to))$$

Links from a transformation to another transformation. Let $l.to \neq \mathsf{sink}$ and $l.from \neq \mathsf{source}$. Since the link does not start at one of the workflow's source nodes, Theorem 7.12 tells us that its datatype must appear in the execution trace immediately after its source transformation is executed:

$$l.datatype \sqsubseteq \mathsf{trace}_w\,[exec](exec^{\sim}(l.from) + 1)$$

Since both ends of the link are transformations, Definition 7.9 ensures that the source transformation appears before the destination transformation in the execution order:

$$exec^{\sim}(l.from) < exec^{\sim}(l.to)$$

This is equivalent to

$$exec^{\sim}(l.from) + 1 \leq exec^{\sim}(l.to)$$

Execution traces are monotonic, so if the datatype appears at any point in the trace, it must also appear in every subsequent step:

$$l.datatype \sqsubseteq \mathsf{trace}_w\,[exec](exec^{\sim}(l.to))$$

□

Determinism of set graphs and set paths

Having provided a formal description of the workflow notation, we now turn our attention to the set notation. One important property of a set graph is that it is *deterministic*. That is, if we start at any node in the set graph, and follow a well-defined sequence of transformations, there is at most one set path that can result. There might be *no* valid set path, which will happen if we come to a node from which we cannot execute the next transformation in the sequence. However, we will never face a choice of which edge to follow next; this will be fully determined by the starting node and the sequence of transformations.

To prove this determinism, we must show three things. First, we can easily show that individual set edges are deterministic. Next, we can show how to construct a set path from a starting position and a sequence of transformations. Finally, we can use these two properties to show that entire set paths are also deterministic.

Theorem 7.14 (Determinism for set edges). *For each node in a set graph, there cannot be multiple outgoing edges for any atomic transformation:*

$$\begin{aligned}&\forall\, g : SetGraph \bullet \forall\, e_1, e_2 : g.edges \bullet \\ &\quad (e_1.from = e_2.from \land \\ &\qquad e_1.transformation = e_2.transformation) \Rightarrow e_1 = e_2\end{aligned}$$

Proof. We are given that the sources and transformations of the two edges are identical:

$$\begin{aligned}&e_1.from = e_2.from \\ &e_1.transformation = e_2.transformation\end{aligned}$$

Since the transformations are identical, the outputs of the transformations must also be equal:

$$\begin{aligned}&g.transformations\,(e_1.transformation).outputs = \\ &\quad g.transformations\,(e_2.transformation).outputs\end{aligned}$$

The *SetEdgeOutputsCreated* constraint from Definition 7.7 tells us that

$$\begin{aligned}&e_1.to = e_1.from \cup g.transformations\,(e_1.transformation).outputs \\ &e_2.to = e_2.from \cup g.transformations\,(e_2.transformation).outputs\end{aligned}$$

Since each respective operand of the unions are equal, the union itself must be, too:

$$e_1.to = e_2.to$$

Finally, since we have shown each element of the edge schemas to be equal, the edges are as well:

$$e_1 = e_2$$

□

Before we can prove that set paths are deterministic, we must define what it means to construct a set path from a sequence of edges.

Definition 7.15 *(Constructing a set path).* A set path is *constructed* by a source node and a sequence of transformations iff the path starts at the source node, and consists of exactly the same sequence of transformations.

$$\begin{array}{l}
\mathsf{stcurtsnoc} : \mathit{SetPath} \to (\mathsf{seq}\ \mathit{Identifier} \times \mathbb{P}\ \mathit{Identifier}) \\
[_, _]\ \mathsf{constructs}\ _ : (\mathsf{seq}\ \mathit{Identifier} \times \mathbb{P}\ \mathit{Identifier}) \leftrightarrow \mathit{SetPath} \\
\hline
\mathsf{stcurtsnoc}\ p = (p.\mathit{source}.\mathit{datatypes}, p.\mathit{edges} \fatsemi (_.\mathit{transformation})) \\
[_, _]\ \mathsf{constructs}\ _ = \mathsf{stcurtsnoc}^\sim
\end{array}$$

With this construction definition, and the proof that individual set edges are deterministic, we can now show that entire set paths are also deterministic.

Theorem 7.16 (Determinism for set paths). *Within a given set graph, there is at most one set path that can be constructed from a source node and a sequence of transformations.*

$$\begin{array}{l}
\forall\ \mathit{xs} : \mathsf{seq}\ \mathit{Identifier};\ \mathit{source} : \mathbb{P}\ \mathit{Identifier};\ p_1, p_2 : \mathit{SetPath} \bullet \\
\quad ([\mathit{xs}, \mathit{source}]\ \mathsf{constructs}\ p_1 \land \\
\quad\ [\mathit{xs}, \mathit{source}]\ \mathsf{constructs}\ p_2 \land \\
\quad\ p_1.\mathit{graph} = p_2.\mathit{graph}) \Rightarrow p_1 = p_2
\end{array}$$

Proof. We are given that the *graph* elements of the path schemas are equal; we must therefore show that the other three elements are also equal. We do this separately for empty and non-empty paths.

Empty paths. Let $\#\mathit{xs} = 0$. Definition 7.15 tells us that

$$\begin{array}{l}
\mathit{source} = p_1.\mathit{source}.\mathit{datatypes} \\
\mathit{source} = p_2.\mathit{source}.\mathit{datatypes}
\end{array}$$

By substitution, it is obvious that

$$p_1.\mathit{source}.\mathit{datatypes} = p_2.\mathit{source}.\mathit{datatypes}$$

Since the *SetNode* schema only contains the *datatypes* element, we have also proved the source nodes to be equal:

$$p_1.\mathit{source} = p_2.\mathit{source}$$

Definition 7.15 also tells us that the length of a path is equal to the length of the transformation sequence that constructs it:

$$\begin{array}{l}
\#p_1.\mathit{edges} = \#\mathit{xs} = 0 \\
\#p_2.\mathit{edges} = \#\mathit{xs} = 0
\end{array}$$

The empty sequence is the only sequence of length zero:

$$\begin{array}{l}
p_1.\mathit{edges} = \langle\rangle \\
p_2.\mathit{edges} = \langle\rangle
\end{array}$$

Therefore, by substitution,

$$p_1.\mathit{edges} = p_2.\mathit{edges}$$

Finally, the *SetPathSourceSinkConsistency* constraint from Definition 7.8 tells us that the source and sink of an empty path must be identical:

$$\begin{array}{l}
p_1.\mathit{source} = p_1.\mathit{sink} \\
p_2.\mathit{source} = p_2.\mathit{sink}
\end{array}$$

Since we have already shown that $p_1.source$ equals $p_2.source$, then by substitution, the sinks are equal as well:

$$p_1.sink = p_2.sink$$

We have shown that each respective element of p_1 and p_2 are equal; thus, the paths are equal as well:

$$p_1 = p_2$$

Non-empty paths. Let $\#xs \neq 0$. Using the same reasoning as for empty paths, we can show that the path sources are equal:

$$p_1.source = p_2.source$$

To prove that the path's edges are equal, we must prove inductively that each element of the sequence is equal.

Base case. The *SetPathSourceSinkConsistency* constraint from Definition 7.8 tells us the first edge of the path must start at the path's source node.

$$(head\ p_1.edges).from = p_1.source$$
$$(head\ p_2.edges).from = p_2.source$$

We have already shown that the path sources are equal, so by substitution,

$$(head\ p_1.edges).from = (head\ p_2.edges).from$$

Definition 7.15 tells us that a path's edges must match the transformation sequence that constructs it:

$$(head\ p_1.edges).transformation = head\ xs$$
$$(head\ p_2.edges).transformation = head\ xs$$

Therefore, by substitution,

$$(head\ p_1.edges).transformation = (head\ p_2.edges).transformation$$

We have shown that the first edge in each path have equivalent sources and transformations; Theorem 7.14 tells us that the edges themselves must also be equal:

$$head\ p_1.edges = head\ p_2.edges$$

Inductive case. Assuming that $p_1.edges\,(i) = p_2.edges\,(i)$, we need to show that $p_1.edges\,(i{+}1) = p_2.edges\,(i+1)$. Because the current edges in the sequence are equal, we know that they have the same destination node:

$$p_1.edges\,(i).to = p_2.edges\,(i).to$$

The *SetPathEdgeConsistency* constraint from Definition 7.8 tells us that the next edges must start where the current edges ended:

$$p_1.edges\,(i+1).from = p_1.edges\,(i).to$$
$$p_2.edges\,(i+1).from = p_2.edges\,(i).to$$

By substitution, we therefore know that the next edges must have the same source node:

$$p_1.edges\,(i+1).from = p_2.edges\,(i+1).from$$

Definition 7.15 tells us that a path's edges must match the transformation sequence that constructs it:

$$p_1.edges\,(i+1).transformation = xs\,(i+1)$$
$$p_2.edges\,(i+1).transformation = xs\,(i+1)$$

Therefore, by substitution,

$$p_1.edges\,(i+1).transformation = p_2.edges\,(i+1).transformation$$

We have shown that the next edges have equivalent sources and transformations; Theorem 7.14 tells us that the overall edges themselves must also be equal:

$$p_1.edges\,(i+1) = p_2.edges\,(i+1)$$

The induction that we have just proved shows that each edge in the two paths are respectively equal; therefore, the edge sequences are equal, as well.

$$p_1.edges = p_2.edges$$

Since we know that the final edge of the two paths are equal, we know that their destination nodes are equal, too:

$$(last\ p_1.edges).to = (last\ p_2.edges).to$$

The *SetPathSourceSinkConsistency* constraint from Definition 7.8 tells us that the last edge in each path must end at the path's sink node:

$$(last\ p_1.edges).to = p_1.sink$$
$$(last\ p_2.edges).to = p_2.sink$$

By substitution, the path sinks must be equal:

$$p_1.sink = p_2.sink$$

We have shown that each respective element of p_1 and p_2 are equal; thus, the paths are equal as well:

$$p_1 = p_2$$

□

Linking workflow executions and set paths

With formal specifications for the two notations, we can now show that they are equivalent. We do this by defining a bijection between workflow executions and set paths, thereby validating step three of the polyadic discovery algorithm described in Section 7.2.2.

At first glance, it might seem that we need to provide a bijection between *workflows* and set paths, since these are the constructs that represent compound transformations in the two notations. However, this bijection does not exist. As we have already mentioned, a workflow might have several executions, since the dataflow links only induce a partial order on the workflow's transformations (Figure 7.15). Further, the same sequence of transformations can result in multiple workflows: if a particular datatype is generated more than once, a transformation input using that datatype might be satisfied by several different dataflow links (Figure 7.19).

Neither of these issues matter, however. In the first case, the pathfinding algorithm gives us a set path, which defines a total order on the underlying atomic transformations. We therefore know exactly which execution order we will be using to construct the corresponding workflow. In the second case, it does not matter if the execution order can yield multiple workflows; their only difference will be that some of the transformation inputs can receive their values from multiple sources. Our interpretation of a datatype, as stated in Section 3.1.3, means that these multiple values will all be equivalent and interchangeable. This allows us to choose one of the resulting workflows arbitrarily.

To start the proof, we can say that a workflow and set graph *correspond* to each other, denoted by the $\doteq$ symbol, if they both represent the same abstract polyadic transformation graph: i.e., when they both encode the same set of datatypes and atomic transformations.

Definition 7.17 *(Correspondence of workflows and set graphs).*

$$
\begin{array}{|l}
\_ \doteq \_ : \mathit{Workflow} \leftrightarrow \mathit{SetGraph} \\
\hline
\forall\, w : \mathit{Workflow};\ sg : \mathit{SetGraph} \bullet \\
\quad w \doteq sg \iff \\
\qquad w.\mathit{datatypes} = sg.\mathit{datatypes} \land \\
\qquad w.\mathit{transformations} = sg.\mathit{transformations}
\end{array}
$$

Next, we can prove the bijection between workflow executions and set paths.

Theorem 7.18 (Equivalence of workflow executions and set paths). *Each execution of a workflow has exactly one analogous set path in the corresponding set graph.*

$$
\begin{array}{l}
\forall\, w : \mathit{Workflow};\ sg : \mathit{SetGraph} \mid w \doteq sg \bullet \\
\quad \forall\, \mathit{exec} : \mathsf{executionsOf}\ w \bullet \\
\qquad \exists_1\, p : \mathit{SetPath} \bullet p.\mathit{graph} = sg \land [\mathit{exec}, w.\mathit{source}]\ \mathsf{constructs}\ p
\end{array}
$$

This theorem requires two separate proofs: one for the existence of the set path, and one for the uniqueness of it.

Proof of existence. We prove the existence of the path by providing a recipe for its construction, and then showing that it is a valid path that is correctly constructed by the workflow execution.

We start by providing an abbreviated name for the execution's trace, as we will be using it frequently in the proof.

$$trace == \mathsf{trace}_w\,[exec]$$

Next we construct the sequence of set edges that will form the body of the set path. It will have the same length as the workflow execution.

$$\begin{array}{|l} edges : \mathsf{seq}\ SetEdge \\ \hline \#edges = \#exec \end{array}$$

Each edge in the path is an execution of the respective transformation from the workflow execution. The execution's trace tells us which datatypes are available immediately before and immediately after this transformation is executed; we use these as the source and destination nodes for the edge.

$$\begin{array}{l} \forall\, i : \mathsf{dom}\ exec \bullet \\ \quad edges\,(i).transformation = exec\,(i) \land \\ \quad edges\,(i).from = \mathsf{set}\ trace\,(i) \land \\ \quad edges\,(i).to = \mathsf{set}\ trace\,(i+1) \end{array}$$

We can now construct the final set path. The workflow defines which datatypes are given to us initially; we use this as the source node of the path. The last element of the execution trace tells us which datatypes are available after all of the transformations have executed; we use this as the sink node of the path.

$$\begin{array}{l} path == \langle\!| graph \rightsquigarrow sg,\ edges \rightsquigarrow edges, \\ \qquad\qquad source \rightsquigarrow \langle\!| datatypes \rightsquigarrow w.sources |\!\rangle, \\ \qquad\qquad sink \rightsquigarrow \langle\!| datatypes \rightsquigarrow \mathsf{set}\ last\ trace |\!\rangle\ |\!\rangle \end{array}$$

We must now show that *path* is a valid *SetPath* and that it is constructed by the workflow execution. The construction proof is trivial, since we can see the following equalities by inspection:

$$\begin{array}{l} path.source.datatypes = w.sources \\ \#exec = \#path.edges \\ \forall\, i : \mathsf{dom}\ exec \bullet exec\,(i) = path.edges\,(i).transformation \end{array}$$

These are exactly the properties needed by Definition 7.15 to show that

$$[exec, w.source]\ \mathsf{constructs}\ path$$

However, we must also show that the path that we have constructed satisfies all of the constraints defined in the *SetPath* schema.

Edge consistency. To satisfy the *SetPathEdgeConsistency* constraint, we must show that each edge ends at the same node that the following edge starts from. By choosing carefully which indices we look at, we can see from the construction of the path that

$$\begin{array}{l} \forall\, i : 1\,..\,\#exec - 1 \bullet \\ \quad path.edges\,(i+1).from = \mathsf{set}\ trace\,(i+1) \land \\ \quad path.edges\,(i).to = \mathsf{set}\ trace\,(i+1) \end{array}$$

By substitution, therefore,

$$\forall\, i : ..\#path.edges - 1 \bullet path.edges\,(i+1).from = path.edges\,(i).to$$

Source and sink consistency, empty execution. If the execution is empty, then we must satisfy the *SetPathSourceSinkConsistency* constraint by showing that the path's source and sink nodes are equal. From the definition of our path,

$$\begin{aligned} &path.source.datatypes = w.sources \\ &path.sink.datatypes = \text{set } last\ trace \end{aligned}$$

Since the execution is empty, its trace has only one element.

$$path.sink.datatypes = \text{set } trace\,(1)$$

According to Definition 7.11, the first element of an execution trace is the set of source nodes in the workflow:

$$trace\,(1) = \text{bag } w.sources$$

By substitution and simplification,

$$\begin{aligned} path.sink.datatypes &= \text{set (bag } w.sources) \\ &= w.sources \\ &= path.source.datatypes \end{aligned}$$

Since the *SetNode* schema contains only one element, we have therefore shown that the path's source and sink are equal.

$$path.source = path.sink$$

Source and sink consistency, non-empty execution. If the execution is not empty, then we must satisfy the *SetPathSourceSinkConsistency* constraint by showing that the path's first edge starts from the path's source, and that the path's final edge ends at the path's sink. From the definition of our path,

$$\begin{aligned} &path.source.datatypes = w.sources \\ &path.edges\,(1).from = \text{set } trace\,(1) \end{aligned}$$

According to Definition 7.11, the first element of an execution trace is the set of source nodes in the workflow:

$$trace\,(1) = \text{bag } w.sources$$

By substitution and simplification,

$$\begin{aligned} path.edges\,(1).from &= \text{set (bag } w.sources) \\ &= w.sources \\ &= path.source.datatypes \end{aligned}$$

Next we show that the path's sink is consistent. From the definition of our path,

$$path.sink.datatypes = \text{set } last\ trace$$

Also, if we set $i = \#path.edges$, the path definition tells us that

$$path.edges\,(\#path.edges).to = \text{set}\ trace\,(\#path.edges + 1)$$

Since the length of the trace is $\#path.edges + 1$, we can simplify this to

$$(last\ path.edges).to = \text{set}\ last\ trace$$

Finally, by substitution,

$$(last\ path.edges).to = path.sink.datatypes$$

Node definedness. To satisfy the first part of the *SetPathStructure* constraint, we must show that each of the set nodes mentioned in the path are actually defined in the graph. The definition of the path tells us that

$$path.source.datatypes = w.sources$$

The *WorkflowDefinedness* constraint from Definition 7.4 tells us that

$$w.sources \subseteq \text{dom}\ w.datatypes$$

Because we are given that the underlying graphs of the workflow and set graph are the same, we know that $sg.datatypes$ and $w.datatypes$ are equal. Substituting for both sides, we see that

$$path.source.datatypes \subseteq \text{dom}\ sg.datatypes$$

Similarly, the path definition tells us that

$$path.sink.datatypes = \text{set}\ last\ trace$$

From Definition 7.11, we can see that every datatype mentioned in an execution trace must have been defined in the execution's workflow.

$$\forall\ b : \text{ran}\ trace \bullet \forall\ dt : \text{set}\ b \bullet dt \in \text{dom}\ w.datatypes$$

Since this constraint holds for every element of the trace, it specifically holds for the last element:

$$\forall\ dt : \text{set}\ last\ trace \bullet dt \in \text{dom}\ w.datatypes$$

We can simplify this to

$$\text{set}\ last\ trace \subseteq \text{dom}\ w.datatypes$$

and substitute to get

$$path.sink.dataypes \subseteq \text{dom}\ sg.datatypes$$

We have now shown that both $path.source.datatypes$ and $path.sink.datatypes$ are subsets of the set graph's datatypes. The *SetNodeCompleteness* constraint from Definition 7.7 tells us that the set graph contains a node for every subset of datatypes in the graph.

$$\begin{array}{l}\forall\ dts : \mathbb{P}\ \text{dom}\ sg.datatypes \bullet \\ \quad \exists\ n : sg.nodes \bullet n.datatypes = dts\end{array}$$

We can apply this specifically to the path's source and sink nodes:

$$\exists\, n : sg.nodes \bullet n.datatypes = path.source.datatypes$$
$$\exists\, n : sg.nodes \bullet n.datatypes = path.sink.datatypes$$

The *SetNode* schema only contains one element, so we can simplify this to:

$$\exists\, n : sg.nodes \bullet n = path.source$$
$$\exists\, n : sg.nodes \bullet n = path.sink$$

Finally, we can eliminate the quantifications as follows:

$$path.source \in sg.nodes$$
$$path.sink \in sg.nodes$$

Edge definedness. To satisfy the second part of the *SetPathStructure* constraint, we must show that each of the edges in the set path are actually defined in the graph. The *SetEdgeCompleteness* constraint of Definition 7.7 requires the existence of an edge in certain situations; our goal is to show that each edge in the path satisfies these criteria, thereby ensuring that the edge exists. The first criterion is that the path's edge must refer to a transformation defined in the graph.

The definition of the path tells us that

$$\forall\, i : \mathrm{dom}\ path.edges \bullet$$
$$path.edges\,(i).transformation = exec\,(i)$$

Definition 7.9 tells us that every transformation in the execution order has appears in the workflow's execution set:

$$exec\,(i) \in w.executions$$

The *WorkflowDefinedness* constraint of Definition 7.4 tells us that every transformation in a workflow's execution set is defined in the graph:

$$w.executions \subseteq \mathrm{dom}\ w.transformations$$

Since anything that is an element of a subset must also be an element of the superset,

$$exec\,(i) \in \mathrm{dom}\ w.transformations$$

Because we are given that the underlying graphs of the workflow and set graph are the same, we know that *sg.datatypes* and *w.datatypes* are equal. Substituting for both sides, we see that

$$path.edges\,(i).transformation \in \mathrm{dom}\ sg.transformations$$

This shows that each edge's transformation is defined in the graph. The second criterion requires us to show that each edge's source contains all of the inputs needed to execute its transformation.

We have just shown previously that each edge's transformation is in the workflow's execution set:

$$\forall\, i : \mathrm{dom}\ path.edges \bullet$$
$$exec\,(i) \in w.executions$$

The *AllInputsSatisfied* constraint from Definition 7.4 tells us that any transformation in this execution

set must have incoming links for all of its inputs:

$$\begin{array}{l}\forall\, input : w.transformations\,(exec\,(i)).inputs \bullet \\ \quad \exists\, l_{IN} : w.links \bullet \\ \quad\quad l_{IN}.to = exec\,(i) \land l_{IN}.datatype = input\end{array}$$

Theorem 7.13 tells us that each input link's datatype must appear in the execution trace immediately before its destination transformation is executed.

$$l_{IN}.datatype \sqsubseteq trace\,(exec^{\sim}(l_{IN}.to))$$

Since $l_{IN}.to = exec\,(i)$, we can simply as follows:

$$\begin{array}{l}l_{IN}.datatype \sqsubseteq trace\,(exec^{\sim}(exec\,(i))) \\ l_{IN}.datatype \sqsubseteq trace\,(i)\end{array}$$

By rewriting, we see that we have an existential quantification that can be eliminated:

$$\begin{array}{l}\exists\, l_{IN} : w.links \bullet \\ \quad l_{IN}.datatype = input \land l_{IN}.datatype \sqsubseteq trace\,(i) \\ input \sqsubseteq trace\,(i)\end{array}$$

By rewriting again, we find a universal quantification that can be eliminated and simplified:

$$\begin{array}{l}\forall\, input : w.transformations\,(exec\,(i)).inputs \bullet \\ \quad input \sqsubseteq trace\,(i) \\ w.transformations\,(exec\,(i)).inputs \sqsubseteq trace\,(i) \\ w.transformations\,(exec\,(i)).inputs \subseteq \mathrm{set}\, trace\,(i)\end{array}$$

Finally, by rewriting and substituting, we get:

$$\begin{array}{l}\forall\, i : \mathrm{dom}\, path.edges \bullet \\ \quad w.transformations\,(exec\,(i)).inputs \subseteq \mathrm{set}\, trace\,(i) \\ \quad sg.transformations\,(path.edges\,(i).transformation).inputs \\ \quad\quad \subseteq path.edges\,(i).from\end{array}$$

This shows that each edge's source node correctly contains all of the inputs needed by its transformation. By assigning variables as follows,

$$\begin{array}{l}id == path.edges\,(i).transformation \\ source == path.edges\,(i).from\end{array}$$

the *SetEdgeCompleteness* constraint from Definition 7.7 tells us that:

$$\begin{array}{l}\exists\, e : sg.edges \bullet \\ \quad e.from = path.edges\,(i).from \land \\ \quad e.transformation = path.edges\,(i).transformation\end{array}$$

Since their source nodes and transformations are equal, Theorem 7.14 tells us that the edges themselves are equal, too, by the determinism of set edges:

$$\begin{array}{l}\exists\, e : sg.edges \bullet \\ \quad e = path.edges\,(i)\end{array}$$

We can eliminate the existential quantification as follows:

$$\forall\, i : \operatorname{dom} path.edges \bullet \\ path.edges\,(i) \in sg.edges$$

Lastly, we can eliminate the universal quantification as follows:

$$\operatorname{ran} path.edges \subseteq sg.edges$$

This shows that *path* is a valid path according to Definition 7.8, and that it is correctly constructed by the workflow execution.

□

We have shown that each workflow execution has *some* equivalent set path; now we must show that this set path is unique.

Proof of uniqueness. Luckily, the uniqueness proof is much simpler than the existence proof. If we assume that there are two paths, p_1 and p_2, that satisfy the conditions of the theorem, we have:

$$\forall\, w : Workflow;\ sg : SetGraph \mid w \mathrel{\hat{=}} sg \bullet \\ \forall\, exec : \mathsf{executionsOf}\ w \bullet \\ (\exists\, p_1 : SetPath \bullet \\ p_1.graph = sg \land [exec, w.source]\ \mathsf{constructs}\ p_1) \land \\ (\exists\, p_2 : SetPath \bullet \\ p_2.graph = sg \land [exec, w.source]\ \mathsf{constructs}\ p_2)$$

However, Theorem 7.16 tells us that set paths are deterministic: two paths constructed from the same source node and transformation sequence must be identical. Therefore,

$$p_1 = p_2$$

□

These proofs show the correctness of the polyadic discovery algorithm described in Section 7.2.2. The construction rules for a set graph allow us to create the corresponding set notation version of a polyadic graph. We can then use a simple pathfinding algorithm to find a set path solution. Finally, since set paths and workflow executions are equivalent, we can correctly construct a workflow that corresponds to the set path solution. Furthermore, since a corresponding workflow and set path consist of the same transformations, their costs are identical. Thus, an optimal set path must also be an optimal workflow.

7.4 Complexity analysis

In the previous section, we proved the correctness of our polyadic transformation discovery algorithm. Now we can analyze the algorithm's complexity. It is easy to see that the complexity is dominated by the size of the set graph, since it includes a node for every set of datatypes in the transformation graph. This requires a set graph with $O(2^D)$ nodes, where D is the number of datatypes in the graph. The pathfinding algorithm that we use to find a solution runs in time polynomial to the

size of the set graph. Thus, it is obvious that our set-based algorithm will run in time *exponential* to the number of datatypes — highly undesirable.

Of course, this does not imply that the underlying problem is inherently "difficult"; we might just have given a poor solution. In this section, we develop yet another notation for polyadic graphs, and use it to show that polyadic transformation discovery is *NP*-hard. This shows that the problem is fundamentally hard, and implies that there is no efficient, polynomial-time solution.

7.4.1 Hypergraphs

The notation used in this section is based on the *hypergraph* [10, 11, 4]. A (directed) hypergraph is similar to a graph, but its edges are allowed to have multiple source and destination nodes. For comparison, Figure 7.24 shows a polyadic transformation graph represented both as a workflow and as a hypergraph. Nodes in the hypergraph represent datatypes, just as with unary transformation graphs. Each atomic transformation is represented by a *hyperedge*, with a source node for each input datatype and a sink node for each output datatype. The γ transformation, for instance, has a hyperedge connecting the A and B nodes to the D node.

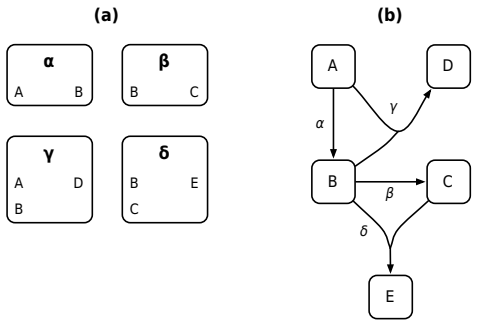


Figure 7.24: A transformation graph in the workflow and hypergraph notations

In a unary graph, a compound transformation is represented by a path; for polyadic graphs, we will use the corresponding notion of a *hyperpath*. Figure 7.25 shows the compound transformation that generates datatypes C and D from A, as both a workflow and a hyperpath.

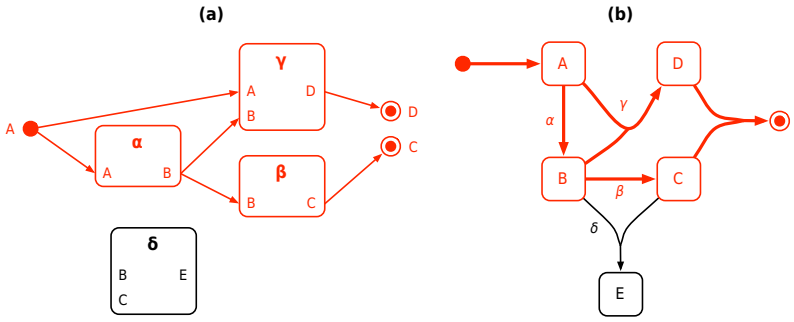


Figure 7.25: A compound transformation as a workflow and a hyperpath

Because of the possibility of branching and merging, a hyperpath is a much subtler construction than the linear path found in a unary graph. Briefly, a hyperpath is a subset of the hyperedges in a

hypergraph. Like hyperedges, a hyperpath has a source and a sink. Some treatments of the subject allow the source and sink to be a set of nodes; others require them to be single nodes. All of the hypergraphs that we will consider will have a single source and a single sink, so this distinction will not concern us.

Not just any subset of hyperedges is a valid hyperpath. At least one of the hyperedges must end at the hyperpath's sink. Further, the hyperpath must contain a "subhyperpath" to each of that hyperedge's sources. The hyperpath in Figure 7.25(b), for instance, must contain a hyperedge ending at the sink node. This hyperedge has datatypes C and D as its sources, so there must also be subhyperpaths from the source to each of these datatypes, as shown in Figure 7.26. We can repeat this process recursively, including smaller subhyperpaths at each stage, until we have worked our way back to the original source node. This forms a *hyperpath tree* of hyperedges, rooted at the original sink node. Figure 7.26 also shows the corresponding tree for our example hyperpath. More details on hyperpaths and hyperpath trees can be found in [6] and [45].

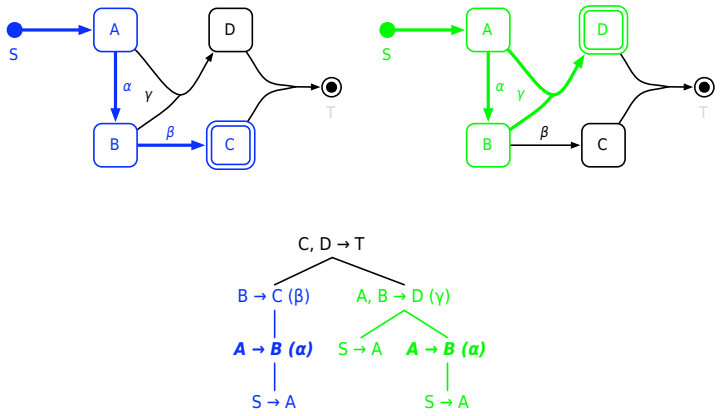


Figure 7.26: Two subhyperpaths, both including the α hyperedge

7.4.2 Hyperpath-finding efficiency

Having described how hypergraphs and hyperpaths can be used to represent transformation graphs and compound transformations, we can now analyze the efficiency of transformation discovery in this model. For unary graphs, this was equivalent to the shortest path problem; for polyadic graphs, we will examine the analogous shortest hyperpath problem. However, the extra complexity of hypergraphs and hyperpaths adds several wrinkles that we did not have to consider in the unary case.

Like edges, the hyperedges in a hypergraph can be *weighted*. For unary graphs, it is obvious how to calculate the weight of a path: it is simply the sum of the weights of the path's edges. For hyperpaths, though, there is not a single obvious way to calculate a hyperpath's weight from the weights of its hyperedges. As we can see from Figure 7.26, hyperedges can appear multiple times in a hyperpath tree. For instance, as highlighted in the hyperpath tree, both of the C and D subhyperpaths contain the α transformation's hyperedge. Intuitively, this is because α's result — the B datatype — is required by both the β and γ transformations. However, even though α appears in the hyperpath tree multiple times, it is only contained in the overall hyperpath once. Intuitively, this is because α is only executed once, regardless of how many times its output is used. These two interpretations

allow us to calculate two different weights for the hyperpath: the first by including α's weight twice (since it is "used" twice in the overall hyperpath), the second by including it exactly once (regardless of how many times it is used).

These weighting functions, and many others, are examined in [59], [6], and [7]; a summary can be found in [5]. Using the terminology from [7], the first weight function (where α is counted twice) is called the *traversal cost* of the hyperpath; the second (where every hyperedge is counted at most once) is called the *cost*. Intuitively, we can distinguish these two metrics by whether they consider the hyperedges of a hyperpath to belong to a bag or a set. Figure 7.27 shows a hypergraph with two possible hyperpaths between the source and sink nodes, along with their hyperpath trees. Path (a) has no repeated hyperedges, so its cost and traversal cost are both 7. Path (b), on the other hand, has a repeated edge (from the source node *S* to datatype *A*); its cost is 6, while its traversal cost is 8. As we can see, the choice of weighting function can easily affect which hyperpath should be considered "shortest". If we use traversal cost as our metric, path (a) is the optimal solution; if we use cost, path (b) is optimal.

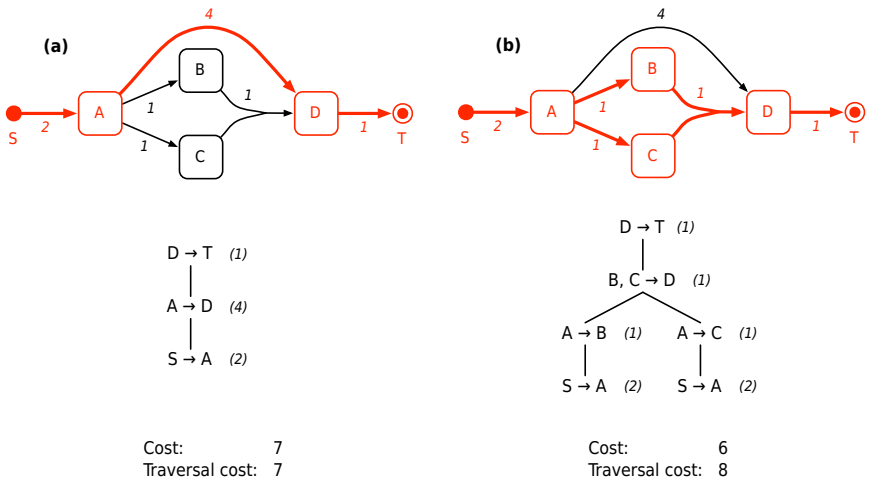


Figure 7.27: Examples of different hyperpath weight functions

Traversal cost is an example of a *value-based function*; these functions have simple recursive definitions that depend solely on the weights of their subhyperpaths. The traversal cost of the hyperpath in Figure 7.25(b), for instance, is simply the sum of the traversal costs of the subhyperpaths in Figure 7.26. The cost of the hyperpath, however, is not as easy to calculate, since summing the subhyperpath costs would result in α being included twice. We would need to analyze the structure of the two subhyperpaths to discover this and produce the correct hyperpath cost.

This means that the two weighting functions have quite different time complexities: each recursive step in the traversal cost function requires exactly one operation, whereas each recursive step in the cost function requires time proportional to the size of the recursive subhypergraphs. As [59] and [6] show, there is an efficient polynomial-time algorithm for finding shortest hyperpaths if a value-based function is used to calculate hyperpath weights. For more complex functions like cost, however, the shortest hyperpath problem is *NP*-hard. An *NP*-hard algorithm is one that we consider to be fundamentally "difficult". There is a large class of problems, known as *NP* (for *nondeterministically polynomial*), for which there are no known polynomial solutions. Moreover, it is accepted

wisdom — though it has not yet been proved — that no polynomial solutions exist. An *NP*-hard problem, then, is at least as difficult as all of the problems in *NP*; if we were ever to find an efficient, polynomial solution to an *NP*-hard problem, this could be used to build an efficient solution to every *NP* problem.

Returning to transformation graphs, these two weight functions correspond to different assumptions about how a compound transformation will be executed. If we use traversal cost, we imply that each time the output of a transformation is used, we need to re-execute that transformation. However, the data that the transformation generates will usually be reusable, so re-executing the transformation is not necessary. The cost metric correctly takes this into account by including each transformation's weight exactly once.

Of course, nothing would require us to actually execute a repeated transformation multiple times. It might be tempting to use the efficient traversal cost metric to find a solution, and then execute each transformation only once, regardless of how many times it appears in the traversal cost solution. Unfortunately, as we have already shown with Figure 7.27, the choice of weighting function affects which solution the discovery algorithm finds. Using a weighting function that does not match how the transformations will actually be executed can cause the discovery algorithm to find suboptimal solutions.

This means that if we want to discover polyadic compound transformations via a hypergraph, we must use cost as the weighting metric for the hyperpath-finding algorithm. As with the set notation, the resulting solution is not an efficient, polynomial-time algorithm. However, with the hypergraph solution, we have gone one step further and shown that the solution is *NP*-hard. Though it has not yet been proved, our current intuition is that the *P* and *NP* complexity classes are distinct. This strongly implies that there is no polynomial time algorithm for polyadic transformation discovery, and that the underlying problem is fundamentally difficult.

Summary

In this chapter we have investigated *polyadic transformation graphs*, which differ from the transformation graphs seen previously in that their atomic transformations can have more than one input and output datatype. We introduced an intuitive *workflow notation* for polyadic graphs that focuses on each of these atomic transformations as opaque units of computation. Unfortunately, this notation does not lend itself to an obvious discovery algorithm for compound transformations. We therefore defined a *set notation*, in which compound transformations are once again represented by paths. This yields a simple discovery algorithm: translate the polyadic transformation graph into the set notation, use a pathfinder to find a compound transformation, and translate the resulting set path into a workflow.

Next, we proved that this algorithm is sound. To do this, we developed a formal specification of the workflow and set notations, and showed that there is a bijection between workflow executions and set paths. This shows that a set path found by the pathfinder can be correctly used to construct an equivalent workflow.

Though provably correct, our algorithm is very inefficient, due to its exponential space requirements. We then showed that it is not just our solution which is inefficent; rather, the underlying problem is fundamentally difficult. To show this, we developed a *hypergraph notation*, in which

compound transformations are represented by hyperpaths and are discovered using an analogous hyperpath-finding algorithm. The efficiency of a hyperpath-finding algorithm depends strongly on the function used to calculate a hyperpath's weight given the weights of its constituent hyperedges; unfortunately, the weighting function needed for polyadic transformation discovery yields an algorithm that is *NP*-hard. This implies that there is no efficient, polynomial-time algorithm for polyadic transformation discovery. However, this intractability is a worst-case bound. In the next chapter, we will investigate the algorithm's complexity further, to see if there are situations where transformation discovery is reasonably efficient.

8 Efficiency concerns

In the previous chapter, we described how transformation graphs can be extended to support polyadic transformations — those with multiple inputs and outputs. Unfortunately, as we showed, the transformation discovery algorithm in this extended model is provably *NP*-hard. Ideally, this intractability will be a worst-case bound, with the hope that "normal" transformation graphs will lend themselves to discovering compound transformations more efficiently. In this chapter, we investigate this hypothesis. To this end, we develop a rapid prototype of the algorithm, allowing us to quickly test many simple transformation graphs, looking for features that lend themselves to efficient discovery. There are many tools that could be used for this purpose; we use the Communicating Sequential Processes (CSP) process algebra [52, 93] to develop the prototype implementation, and the FDR refinement checker [92, 95] to explore the problem's complexity space. Analyzing the results, we can highlight certain optimizations that FDR makes while performing its refinement checks, which we then relate to the underlying transformation discovery problem. This shows several analogous optimizations that we can make to the algorithm presented in the previous chapter. The contents of this chapter have been published previously in [31].

8.1 CSP implementation

In this section, we describe a prototype implementation of the transformation discovery problem described in the previous chapter, written in CSP. The usual strategy for working with CSP specifications is to define two processes: one providing a specification of what the system should do, and the other describing a particular implementation of the system. One then uses a refinement checker such as FDR to verify that the implementation refines, and therefore satisfies, the specification.

We will follow a similar approach, though our CSP processes will not describe a "specification" and "implementation", per se. Instead, we will use the first process to describe the structure of the graph, and the basic rules about when transformations can be executed. We will use the second process to describe the specific property that we are looking for in a solution: that, given instances of a particular set of *initial datatypes*, there is some sequence of transformations that can be executed that will yield instances of a different set of *desired datatypes*. Note that we do not describe *how* to find solutions; we only provide a declarative description of the problem structure and of valid

solutions.

8.1.1 Graph structure

We start by declaring the CSP types needed for the specification. The *Datatype* type represents a single datatype from the transformation graph. (The overloading of the term "datatype" is unfortunate but unavoidable; we will use "type" to refer to the syntactic concept in the CSP language, and "datatype" to refer to a node in a transformation graph.) A *Transformation* has a unique identifier, and is defined by two sets of datatypes: one for its inputs and one for its outputs. A particular transformation graph can be encoded by providing concrete values for the *Datatype* type and the *Transformations*, *GivenTypes*, and *DesiredTypes* variables. The *Transformations* variable contains all of the transformations in the graph. The *GivenTypes* variable specifies which datatypes we are given instances of, while *DesiredTypes* specifies which datatypes need to be generated by the discovered compound transformation.

datatype $Datatype, XformID$
nametype $Transformation = XformID \times (\mathbb{P}\, Datatype) \times (\mathbb{P}\, Datatype)$
variable $Transformations, GivenTypes, DesiredTypes$

Next we define the event channels that will be used in the specification. The *have* channel signals when a datatype has become available, regardless of how it was obtained. The *given* channel is used to notify other processes which datatypes are given. The *execute* channel signals that a particular atomic transformation has executed. The *produce* channel indicates that a datatype has been produced as the output of some transformation. Finally, the *finish* channel signifies that a datatype has been used as the final result of the compound transformation.

channel $given, have, produce, finish : Datatype$
channel $execute : XformID$

Now we can construct the CSP process that represents the structure and rules of a transformation graph. We follow the standard approach of declaring subprocesses for each of the individual properties or constraints of the system, which we then compose together into a final specification using parallel composition.

We first define a *MakeAvailable* process that is responsible for generating *have* messages whenever a datatype instance becomes available. This can happen in one of two ways: we can be given the instance (in which case we match a *given* message), or it can be generated by the execution of a transformation (in which case we match a *produce* message).

$$\alpha(MakeAvailable) = \{\!|\, given, produce, have \,|\!\}$$
$$MakeAvailable =$$
$$\quad given?t \rightarrow have!t \rightarrow MakeAvailable$$
$$\quad \Box$$
$$\quad produce?t \rightarrow have!t \rightarrow MakeAvailable$$

Next we define a *Given* process that generates the initial *given* messages for the datatypes that we start with. The alphabet of this process contains *all* *given* messages, even though only certain *given* messages are created; this ensures that CSP events only appear for those datatypes that actually are

given to us.

$$\alpha(Given) = \{\!|\, given \,|\!\}$$
$$Given = |||\ t : GivenTypes \bullet given!t \rightarrow Stop$$

Next we define a process to handle the *finish* messages. We keep track of which datatypes we have; when one of the *DesiredTypes* becomes available, we allow a *finish* event for it. We do not want to generate multiple *finish* events for any datatype, so we must also keep track of the datatypes that have already been *finish*ed. This means that we only allow a *finish* event if the datatype is one of the desired outputs, it is available, and we have not already generated a *finish* event for it.

$$\begin{array}{l}
\alpha(Finish) = \{\!|\, finish, have \,|\!\} \\
Finish = \\
\quad \text{let} \\
\quad\quad Have\,(avail, finished) = \\
\quad\quad\quad have?t \rightarrow Have\,(avail \cup \{t\}, finished) \\
\quad\quad\quad \Box \\
\quad\quad\quad finish?t : (avail \setminus finished) \cap DesiredTypes \rightarrow Have\,(avail, finished \cup \{t\}) \\
\quad \text{within} \\
\quad\quad Have\,(\varnothing, \varnothing)
\end{array}$$

Our next process is responsible for preventing a particular transformation from executing before all of its inputs are satisfied. We define it similarly to the *Finish* process: we keep track of which input datatypes we have; once all of them are available, we allow any number of *execute* events to occur for this transformation. The process alphabet contains a *have* event for each input datatype, and the *execute* event for the transformation.

$$\begin{array}{l}
\alpha(XformPrereq\,(\,(id, inputTypes, outputTypes)\,)) = \\
\quad \{execute.id\} \cup \{\, t : inputTypes \bullet have.t \,\} \\
XformPrereq\,(\,(id, inputTypes, outputTypes)\,) = \\
\quad \text{let} \\
\quad\quad Have\,(avail) = \\
\quad\quad\quad (avail = inputTypes)\ \&\ execute!id \rightarrow Have\,(avail) \\
\quad\quad\quad \Box \\
\quad\quad\quad have?t : inputTypes \rightarrow Have\,(avail \cup \{t\}) \\
\quad \text{within} \\
\quad\quad Have\,(\varnothing)
\end{array}$$

For the transformation graphs described in this chapter, we assume that every datatype is *reusable*: that any instance of the datatype can be used multiple times without penalty. We do not allow single-use datatypes as described in Section 7.1.1. For this reason, we do not remove any elements from the set of available datatypes in the *Finish* and *XformPrereq* processes. If desired, we could use a more complicated definition for these processes to limit the number of times that a particular datatype could be consumed.

The above process verifies that the prerequisites are satisfied for a single transformation. We must use parallel composition to combine them together: since multiple transformations might be waiting for the same datatype to satisfy an input, they must be notified of its availability simultaneously. This means that they must synchronize on the corresponding *have* event. This parallel composition yields

the *Prereqs* process, which verifies the prerequisites of each atomic transformation simultaneously.

$$\alpha(Prereqs) = \bigcup \{ xf : Transformations \bullet \alpha(XformPrereq\,(xf)) \}$$
$$Prereqs = \|\ xf : Transformations \bullet XformPrereq\,(xf)$$

Next we define a process that describes what happens when a particular transformation is executed. The process is fairly straightforward: it waits for the appropriate *execute* event, after which it outputs *produce* events for each of the transformation's output datatypes. We use replicated interleaving to allow the *produce* events to occur in any order. The overall process then ends in *Skip*. The process alphabet does not contain any extra events — only the *execute* and *produce* messages appropriate to the transformation.

$$\alpha(ExecuteOneOnce\,(\,(id, inputTypes, outputTypes)\,)) =$$
$$\quad \{execute.id\} \cup \{\ t : outputTypes \bullet produce.t\ \}$$
$$ExecuteOneOnce\,(\,(id, inputTypes, outputTypes)\,) =$$
$$\quad execute!id \rightarrow (|||\ t : outputTypes \bullet produce!t \rightarrow Skip)$$

The *ExecuteOneOnce* process is parameterized on the definition of a transformation; now we instantiate this process for each of the actual transformations in the graph. The *ExecuteAnyOnce* process allows the environment to execute any one transformation. Its alphabet includes *all* of the *produce* messages, since we want to prevent datatypes that do not play a part in some transformation from being produced.

$$\alpha(ExecuteAnyOnce) = \{\!|\, execute, produce \,|\!\}$$
$$ExecuteAnyOnce = \Box\ xf : Transformations \bullet ExecuteOneOnce\,(xf)$$

With the *ExecuteAnyOnce* process, we have allowed the environment to execute a single transformation. Now we allow it to execute a sequence of them. Since the *ExecuteAnyOnce* process ends with a *Skip* (due to it being defined in terms of *ExecuteOneOnce*), we can accomplish this with a recursive sequential composition. The *Execute* process allows *any* sequence of transformations to be executed; it does not take into account whether a transformation has its inputs satisfied. This constraint is handled by the *Prereqs* process, and so it will be introduced automatically when we compose together all of the graph processes.

$$\alpha(Execute) = \alpha(ExecuteAnyOnce)$$
$$Execute = ExecuteAnyOnce\ ;\ Execute$$

Finally, we can merge together all of the previous processes using parallel composition. This yields an overall *Graph* process that satisfies the constraints introduced by each of its constituent parts. We also provide a view of the graph (*GraphOutputs*) that hides everything except for the *finish* channel; this allows us to only concern ourselves with which final datatypes can actually be produced, without worrying about the details of which transformations were executed.

$$\alpha(Graph) = \{\!|\, given, have, execute, produce, finish \,|\!\}$$
$$Graph = MakeAvailable \parallel Given \parallel Finish \parallel Prereqs \parallel Execute$$

$$\alpha(GraphOutputs) = \{\!|\, finish \,|\!\}$$
$$GraphOutputs = Graph \setminus (\alpha(Graph) \setminus \{\!|\, finish \,|\!\})$$

8.1.2 Transformation discovery process

Next we construct the CSP process that tests whether all of the desired datatypes are eventually produced by some compound transformation. When we only have a single desired type, this is exceedingly simple. Since we have hidden everything except for the *finish* message for our desired type, we just want to ensure that this message occurs. This property is given by the $Want'_T$ process, which has exactly two traces:

$$Want'_T(\{t\}) = finish!t \rightarrow Stop$$
$$traces(\llbracket\ Want'_T(\{t\})\ \rrbracket) = \{\langle\rangle, \langle finish.t\rangle\}$$

The empty sequence is a trace of every process. We are hoping that $\langle finish.t\rangle$ will be a trace of *GraphOutputs*, since this would imply that there is some sequence of transformations that produces the desired datatype. If this is true, the traces of $Want'_T$ will be a subset of the traces of *GraphOutputs*. Therefore, a traces refinement check will provide us with a solution:

$$\text{assert } GraphOutputs \sqsubseteq_T Want'_T(DesiredTypes)$$

We can use a similar traces check when we have many desired output datatypes. We can construct a $Want_T$ process that allows the appropriate *finish* events in any order:

$$Want_T(\varnothing) = Stop$$
$$Want_T(types) = |||\ t : types \bullet finish!t \rightarrow Stop$$
$$traces(\llbracket\ Want_T(\{t_1, t_2\})\ \rrbracket) =$$
$$\{\langle\rangle, \langle finish.t_1\rangle, \langle finish.t_2\rangle, \langle finish.t_1, finish.t_2\rangle, \langle finish.t_2, finish.t_1\rangle\}$$

If the transformation graph can generate all of these datatypes, the *GraphOutputs* process will output exactly one *finish* message for each. Further, since the *finish* messages are not coupled to the order in which the atomic transformations are executed, *GraphOutputs* will be able to output these *finish* messages in any order. Thus, the traces of $Want_T$ will be a subset of the traces of *GraphOutputs*. (In fact, because neither process has any other visible events, they will be traces-equivalent.)

On the other hand, if the graph *cannot* generate each desired datatype, then the *GraphOutput* process will not have any trace containing every *finish* event. Since $Want_T$ does contain such a trace, the traces of $Want_T$ will *not* be a subset of the traces of *GraphOutputs*. This means that a valid compound transformation exists iff the following refinement holds:

$$\text{assert } GraphOutputs \sqsubseteq_T Want_T(DesiredTypes)$$

Unfortunately, while this correctly tells us if a compound transformation *exists*, it does not tell us what the transformation *is*. Luckily, we can find this information with only slight modifications. We create a new $Want_F$ process as follows:

$$Want_F(\varnothing) = Stop$$
$$Want_F(\{t\}) = Stop$$
$$Want_F(types) = \sqcap\ t : types \bullet finish!t \rightarrow Want_F(types \setminus \{t\})$$

This differs from $Want_T$ in two respects. First, we use internal choice instead of interleaving to establish each permutation of the *finish* events. Second, for each of these permutations, we only accept *all but one* of the *finish* events, refusing the final one.

With these changes, we can use the *stable failures* model of CSP instead of the previous traces model. If there is a valid compound transformation, the *GraphOutputs* process must allow every *finish* message to occur, in any permutation. The $Want_F$ process, however, only accepts all but one of these events; there is no situation where it will accept every *finish* event. Thus, the stable failures of *GraphOutputs* are *not* a subset of the stable failures of $Want_F$.

If, on the other hand, no compound transformation is possible, then there must be at least one *finish* event that *GraphOutputs* refuses. Further, it will refuse this *finish* event at every point during its execution. $Want_F$ can also refuse this event at any point: either because there are other *finish* events for the internal choice to fall back on, or because it is the final remaining *finish* event, which we always refuse. Thus, the stable failures of *GraphOutputs* *are* a subset of the stable failures of $Want_F$. We can now check the following negated refinement:

$$\text{assert } Want_F(DesiredTypes) \not\sqsubseteq_F GraphOutputs$$

There are two important points to note. First, the order of the operands in the refinement check has been reversed. As we will see in our time complexity analysis, this causes a noticable improvement in efficiency on its own. Second, our choice of semantic model is important. The *Graph* process can execute the same transformation repeatedly forever, which causes the *GraphOutputs* process to diverge. By using the stable failures model instead of the failures-divergences model, we ignore these situations.

When we check this refinement, there are two possible outcomes. If the refinement check succeeds, then we know that there is no valid compound transformation. If it fails, then the compound transformation exists, and FDR will provide a counterexample to the refinement. By examining this counterexample, we will find the sequence of *execute* events that defines the execution order of the compound transformation solution. Furthermore, since FDR uses a breadth-first search to perform the refinement check, the counterexample returned will be the one with the fewest events in its trace. Since this corresponds to the compound transformation with the fewest atomic transformations, the result of the refinement check will be the optimal transformation solution.

8.2 Analysis using FDR

In the previous section we presented a prototype implementation of the polyadic discovery algorithm using the CSP process algebra. By casting the problem as a suitable refinement test between two processes, we can use the FDR refinement checker to search for compound transformations. In this section, we run this refinement check over many different transformation graphs, of varying shapes and sizes, recording how efficiently FDR can find solutions (in both space and time). Doing so gives us an empirical view into the complexity space of the problem, with the hope of finding regions of transformation graphs for which the discovery algorithm is more efficient than the *NP*-hard worst-case bound. Ideally, these regions will correspond to the kinds of transformation graphs that are more likely to appear in practice, suggesting that polyadic discovery can still be a useful tool.

The obvious way to measure the space and time complexity of our prototype would be to record the maximal amount of memory used by FDR, and the amount of wallclock or actual processor time needed to perform the refinement check. However, we use a different metric: all measurements are

made with respect to the underlying labeled transition system (LTS) that FDR creates for a compiled CSP process. Because of *supercompilation* [47], FDR will usually not have to store the process's entire abstract LTS in memory. We measure the space complexity as the size of this smaller supercompiled LTS. The refinement check, however, must be performed on the full abstract LTS, which requires *explicating* the supercompiled LTS into its full form. (The explicated LTS nodes are allocated and deallocated as they are needed, so as to avoid storing the full LTS in memory at once.) We therefore use the number of explicated LTS states visited during the refinement check as a measure of the time complexity.

We measure the space and time complexity in this way because these measurements depend only on the definition of the CSP process. The space complexity metric is fully deterministic, since FDR will always compile a CSP process into the same LTS. The time metric is fully deterministic, as well, since FDR will perform the same search for any particular refinement check. Our measurements, therefore, do not depend on the speed or load of the machine used to perform the refinement check, and are more reproducible.

All of the figures mentioned in this section begin on page 141.

8.2.1 Space complexity

Our first experiment is to measure the space complexity of the constructed graph representation. Initially, we only consider how the graph size is affected by the number of datatypes in the graph, so we consider graphs containing a varying number of datatypes and no transformations. Figure 8.1 shows the size of the labeled transition system that FDR constructs for each transformation graph process. As the figure uses a logarithmic scale, we can see that the graph size grows exponentially; graphs with more than twenty datatypes took over an hour to compile on a reasonably fast workstation.

The problem is with the *Finish* and *XformPrereq* processes, specifically with their internal *Have* subprocesses. These subprocesses maintain the set of available datatypes as a state parameter. Unfortunately, sets require exponential space; since FDR is compiling this subprocess into a low-level operator tree, the *Have* process's LTS also requires exponential space. Luckily, we can modify the *Finish* process as follows:

$$
\begin{array}{l}
Finish = \\
\quad \text{let} \\
\quad\quad DontHave(t) = have!t \rightarrow Have(t) \\
\quad\quad Have(t) = \\
\quad\quad\quad (t \in DesiredTypes)\ \&\ finish!t \rightarrow Finished(t) \\
\quad\quad\quad \Box \\
\quad\quad\quad have!t \rightarrow Have(t) \\
\quad\quad Finished(t) = have!t \rightarrow Finished(t) \\
\quad \text{within} \\
\quad\quad ||| \ t : Datatype \bullet DontHave(t)
\end{array}
$$

We can make a similar modification to *XformPrereq*:

$$
\begin{array}{l}
XformPrereq((id, inputTypes, outputTypes)) = \\
\quad \text{let}
\end{array}
$$

$$
\begin{aligned}
&\alpha(DontHave(t)) = \{execute.id, have.t\} \\
&DontHave(t) = have!t \rightarrow Have(t) \\
&Have(t) = (execute!id \rightarrow Have(t)) \,\Box\, (have!t \rightarrow Have(t)) \\
&\text{within} \\
&\quad \|\, t : inputTypes \bullet DontHave(t)
\end{aligned}
$$

Here we have redefined the internal subprocesses to only keep track of a single datatype. We then create copies of these internal subprocesses for each of the datatypes, and use a composition operator to combine them. For the *Finish* process, we can use interleaving, since the subprocess alphabets are disjoint. In the *XformPrereq* process, on the other hand, the subprocesses for each input datatype must synchronize on the *execute* event, since all of the inputs must be available before the transformation can proceed. We must therefore use alphabetized parallel for the composition.

FDR will compile the subprocesses into low-level operator trees; however, since they no longer maintain exponential state, these trees will be small. The composition of these smaller processes is far more efficient than the original exponential LTS; Figure 8.2 shows the same space measurements for a graph constructed with the modified *Finish* and *XformPrereq* processes. With this modification, we are easily able to represent graphs with hundreds of datatypes. This optimization works because the availability of any one datatype is independent of the rest. As we will show later, this modification to the CSP process will also suggest a similar modification to our naïve discovery algorithm.

Next we show how the size of the graph process is affected by the number and arrangement of transformations in the graph. For this experiment, we fix the number of datatypes in the graph, and examine four situations, as shown in Figure 8.3. First, as a control, we again examine the graph with no transformations. Second, we introduce a single directed cycle of transformations that encompasses all of the datatypes in the graph. Third, we consider a graph with two directed cycles, pointing in opposite directions. Finally, we consider the fully-connected graph, where a transformation directly connects every possible pair of datatypes, including self-cycles.

Figures 8.4 and 8.5 show how the number of LTS states and transitions, respectively, vary based on the shape of the graph and the number of datatypes in it. The None, Cycle, and Double-cycle graphs all have linear growth, whereas the All-pairs graph has quadratic growth. This can be easily explained by Figure 8.6 — the All-pairs graph has a transformation connecting each pair of datatypes, which gives a quadratic growth for the number of transformations as compared to the number of datatypes. This suggests that we should also compare the size of the LTS to the number of transformations, as is shown in Figure 8.7. In this case, each style of transformation graph yields linear growth for its LTS size. Part of the overall growth comes from the datatypes, and part comes from the transformations; Figure 8.8 combines everything together into a three-dimensional graph to show this relationship. The contour lines show that the resulting surface is planar, yielding an $O(D + T)$ overall size for a graph's LTS. One can imagine the graph from Figure 8.6 lying on the XY plane; these curves are then projected up onto the growth plane to yield a single growth curve for a particular style of transformation graph.

8.2.2 Time complexity

Next we examine the time complexity of the algorithm. We again look at four different "shapes" of transformation graph, this time shown in Figure 8.9. In all cases, we are seeking a transformation

between the source datatype S and the destination datatype D. The shapes differ in the number of additional datatypes in the graph, and in how the datatypes are connected. In part (a), we have a single sequence of datatypes A_1 through A_n, with a single path through the graph from S to D. In part (b), we have the same sequence of datatypes A_1 through A_n, but in this case, they are not needed to transform from S to D. In part (c), we have two sequences of datatypes, A_1 through A_n and B_1 through B_n, between S and D. Either one can be used as a valid transformation path. Finally, in part (d), we again have two sequences of datatypes between S and D, but we introduce crosslinks as well, allowing the algorithm to jump from the A datatypes to the B datatypes at any point in the sequence. In this graph, there are $n + 2$ valid transformations between S and D.

The results of this analysis are shown in Figures 8.10 through 8.13. Logarithmic scales are used in these figures when the growth rates are especially pronounced. Several important conclusions can be drawn from this data. In most cases, the number of LTS states and transitions that must be examined during the discovery algorithm is much greater than the number needed to represent the graph itself. This implies that with our more efficient process definition, FDR is not initially instantiating the entire structure of the graph; rather, the graph process is encoding a recipe for dynamically instantiating the graph as needed. This corresponds with our understanding of FDR's use of supercompilation to distinguish between low- and high-level operator trees: the exponential state is "hidden" by the high-level parallel compositions. We must still examine many of these exponential states during the refinement check, taking time, but it is not necessary to store them all in memory at once, saving space.

Next, we can see that the execution time for the discovery algorithm is almost entirely dependent on the number of transformation paths that must be checked. This is most apparent in shape B (Figure 8.11), where regardless of the number of datatypes in the graph, there is a constant size transformation solution. Once the discovery algorithm finds this path, no more processing is required. In this case, the order of the operands in our stable failures refinement is beneficial, since FDR must *normalize* the left-hand side before performing the check. In this case, the left-hand side is $Want_F$ — whose LTS size depends only on the number of *desired* datatypes, and not on the size of the total transformation graph. For our earlier traces refinement, on the other hand, the left-hand side was *GraphOutputs*, whose size increases as new datatypes and transformations are added, even if they are not needed in the discovered solution. Thus, for the failures refinement, the normalization happens much faster, since it operates on a much smaller CSP process.

Figure 8.14 shows the relationship between the complexity curves of shapes A (Figure 8.10), C (Figure 8.12), and D (Figure 8.13). Shape C seems to be a simple modification to the graph from shape A, only adding a single additional possible transformation path. However, since FDR is performing the equivalent of a breadth-first search, it will try to make progress down both paths simultaneously. Only after reaching the end of both intermediary sequences will FDR discover that only one was needed to reach the destination. Worse, it must consider every interleaving of the transformations in the two paths while advancing through the graph. Shape D exacerbates this problem by introducing crosslinking edges. Now, instead of having to consider all possible permutations of two transformation paths, FDR must consider permutations of $n+2$ paths. The growth is much more pronounced, and approaches the worst-case exponential growth; whereas with shape A we were able to consider graphs with 150 datatypes in a reasonable amount of time, shape D quickly becomes infeasible after only twenty datatypes.

Finally, it is important to point out we have not truly eliminated the exponential growth curve of the algorithm's running time; it is still exponential for pathological inputs. This might seem to be a discouraging result at first, but it is in fact still useful in practice. As we will discuss in the next section, real-world transformation graphs tend not to contain a large number of datatypes and transformations, so any improvement in the efficiency of the discovery algorithm for smaller graphs will be helpful.

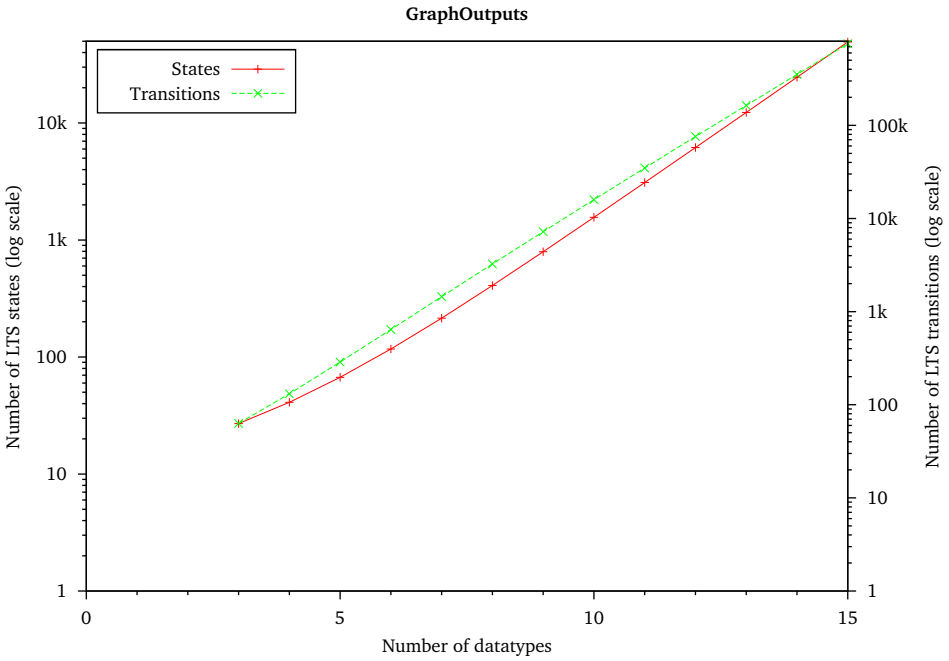


Figure 8.1: Space required for the original transformation graph process

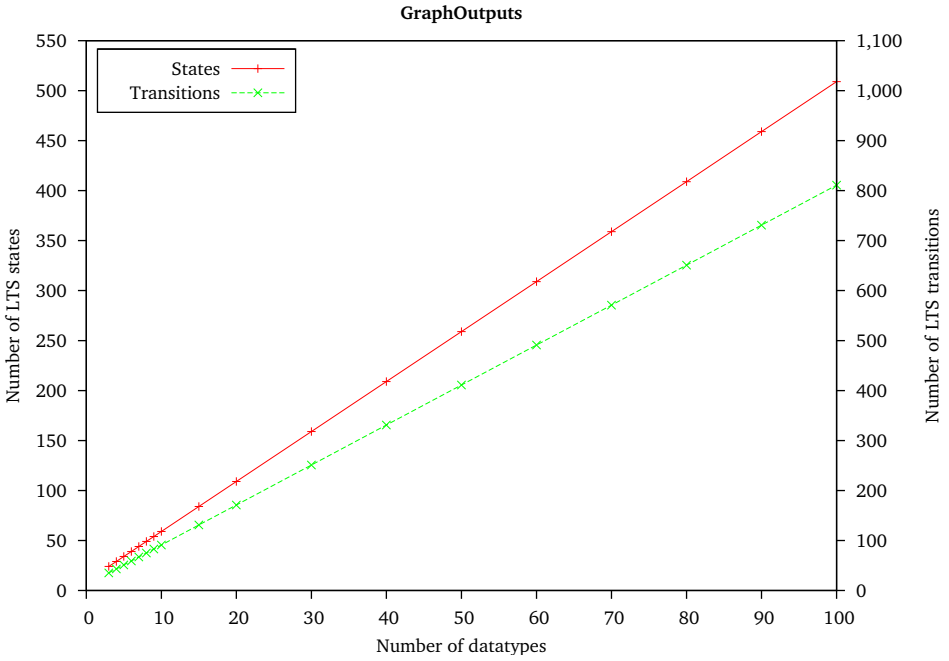


Figure 8.2: Space required for the modified transformation graph process

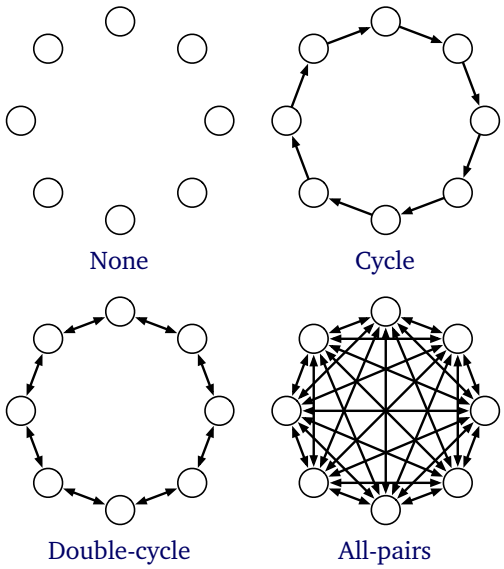


Figure 8.3: The different transformation graph shapes used in the space analysis

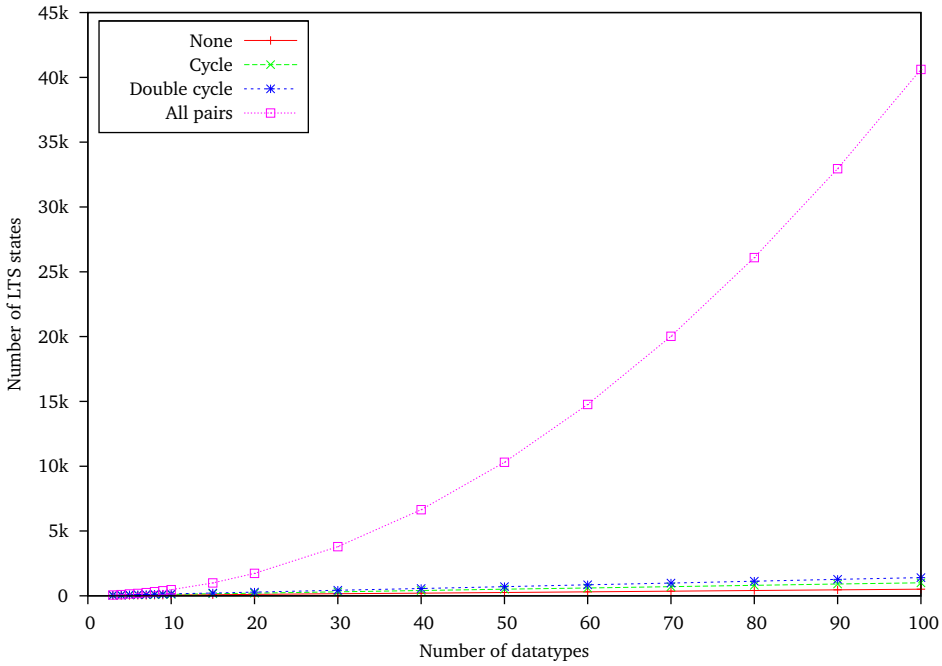


Figure 8.4: Number of LTS states compared to number of datatypes

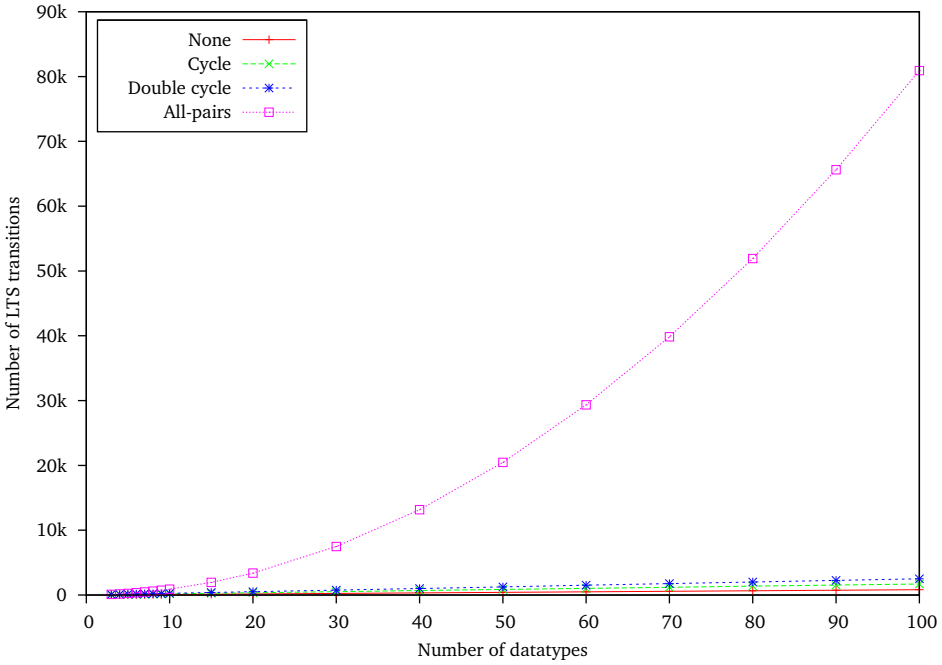


Figure 8.5: Number of LTS transitions compared to number of datatypes

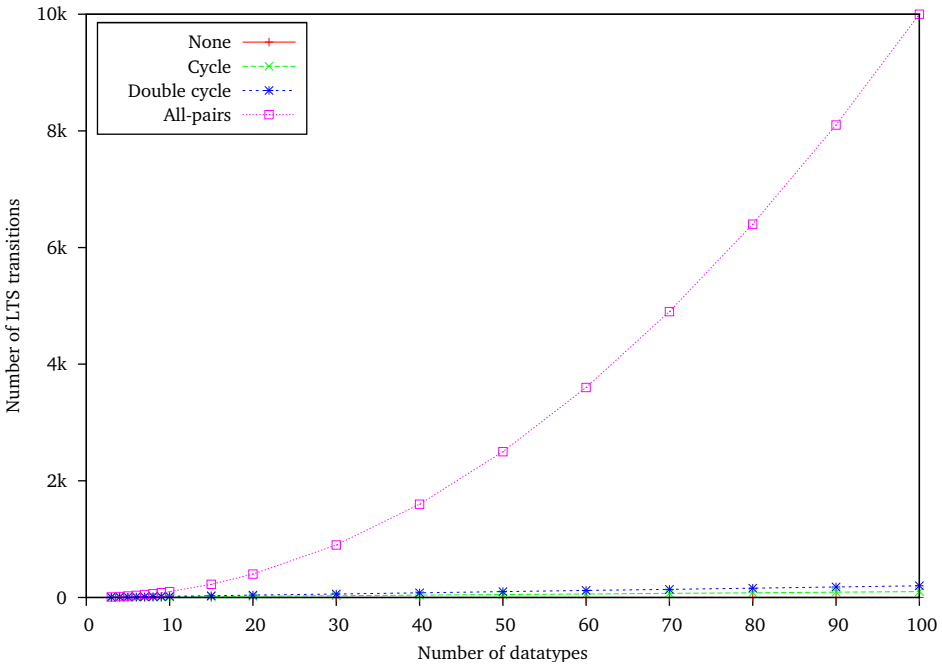


Figure 8.6: Number of transformations compared to number of datatypes

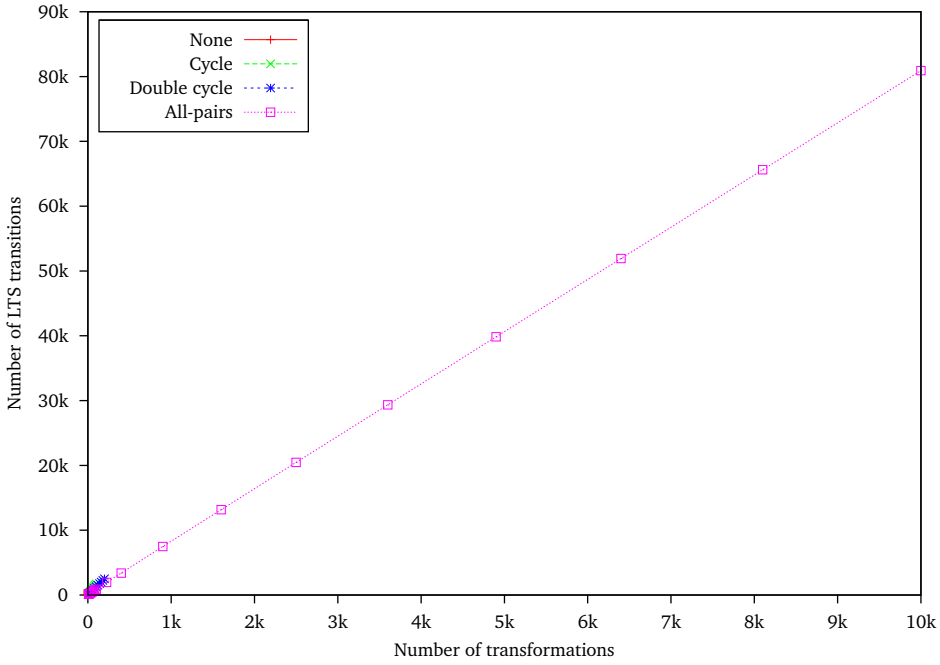


Figure 8.7: Number of LTS transitions compared to number of transformations

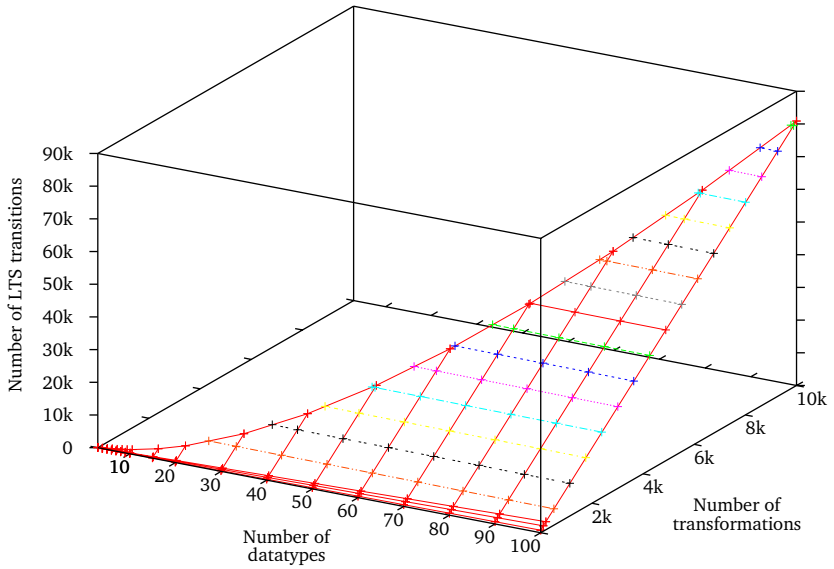


Figure 8.8: Overall relationship between graph size and LTS size

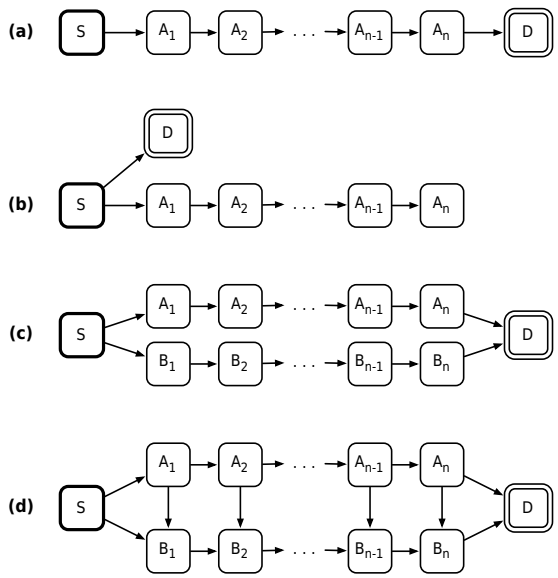


Figure 8.9: The different transformation graph shapes used in the timing analysis

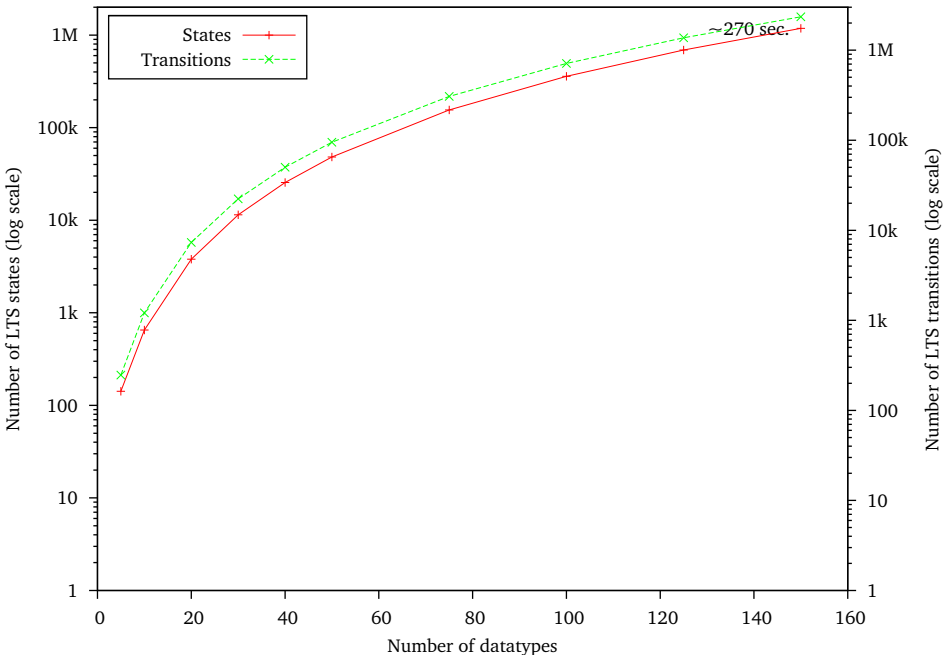


Figure 8.10: Number of states checked for shape A

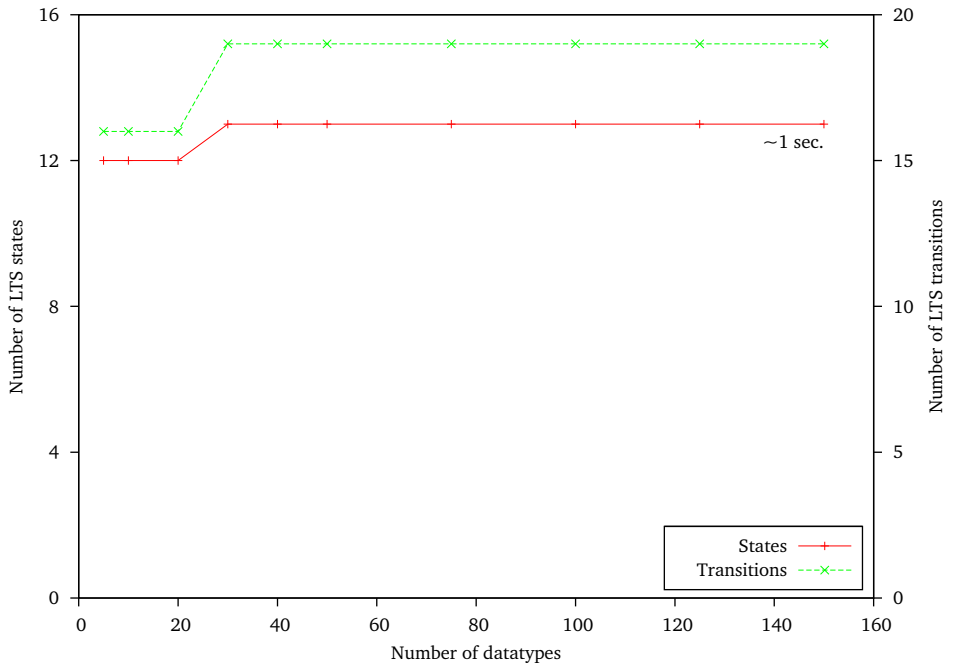


Figure 8.11: Number of states checked for shape B

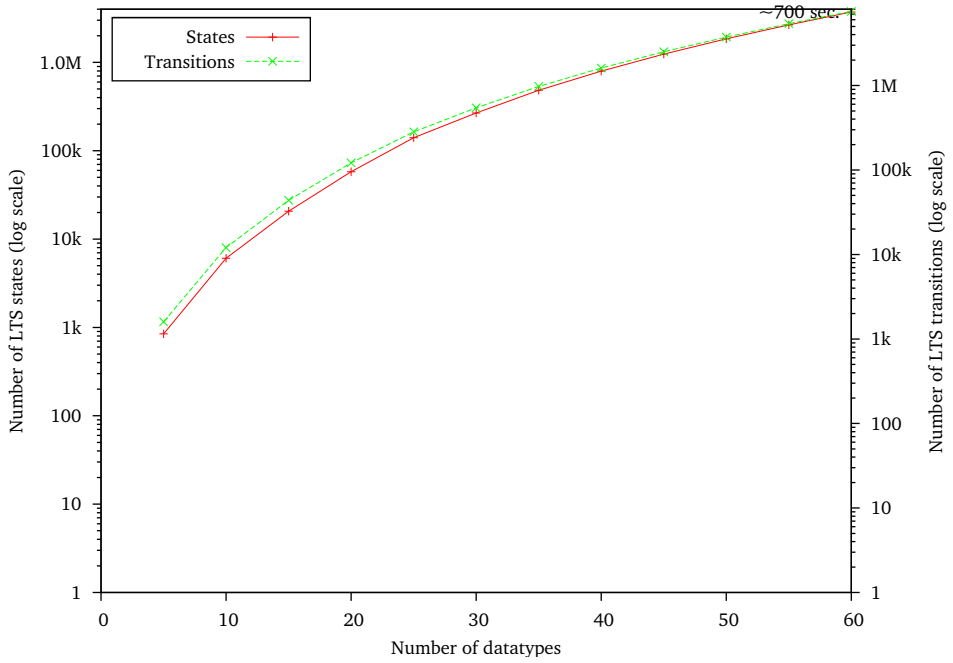


Figure 8.12: Number of states checked for shape C

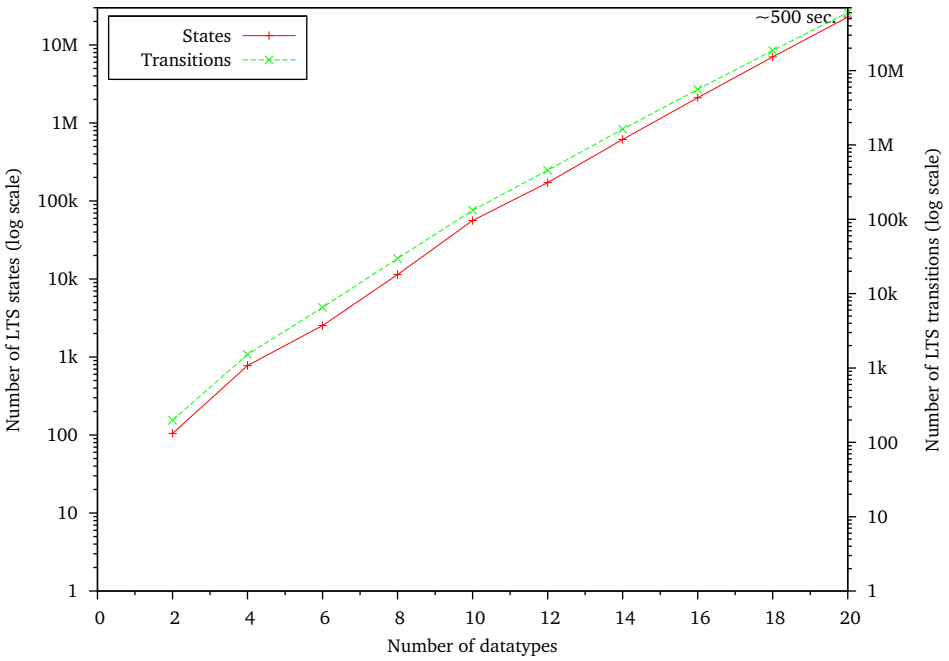


Figure 8.13: Number of states checked for shape D

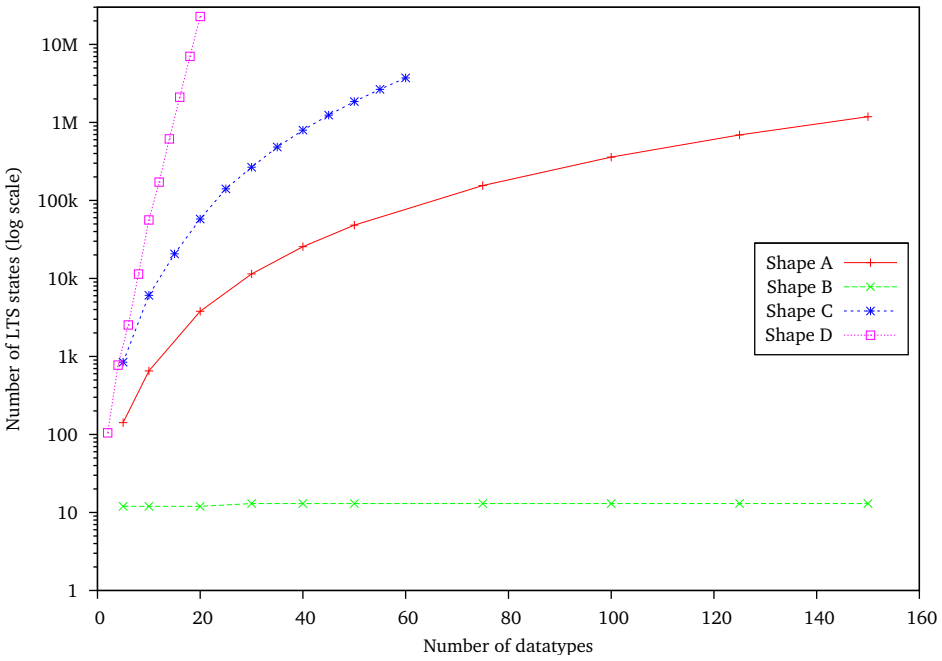


Figure 8.14: Number of states checked for all shapes

8.3 Interpreting and exploiting the results

Having expressed the polyadic discovery problem as a CSP process, and analyzed the complexity space of this process, we can now show how the insights from this analysis can be used with the set graph algorithm from the previous chapter. First, we interpret the kinds of transformation graphs that were more efficient, identifying the features of those graphs that led to efficiency gains, and showing why "real-life" transformation graphs will tend to have those features. Then, we show how the particular technique that FDR uses — lazy evaluation — can be incorporated into the discovery algorithm described in Chapter 7.

8.3.1 Causes of the efficiency gains

In this section, we review the classes of transformation graph for which FDR was able to find a solution more efficiently. The example graphs that we used were fairly abstract, so we extrapolate these results to describe the "real-world" graph features that would lead to similar efficiency gains. Finally, we argue that these features are exactly those that will tend to appear in practical transformation graphs, increasing the utility of our polyadic discovery algorithm.

According to our analysis, the space complexity for the discovery algorithm is fairly static, determined only by the number of datatypes and transformations in the graph. This implies that reducing the number of datatypes in a graph can be an effective way to improve efficiency. In practice, this strategy should prove useful, since large transformation graphs tend to be easily separated into connected components. Intuitively, this is because the datatypes in the graph will tend to form "clumps", where a datatype can be transformed into anything in its clump, but not into anything outside of it. For instance, we might add datatypes to the graph in Figure 4.5 to represent the contact details (such as a postal address) of the scientist collecting a particular microscope image. However, we will never be able to transform the image itself into a postal address, and vice versa. The image formats will form one connected component, and the contact detail datatypes will form another. By treating these connected components as separate transformation graphs, we reduce the number of datatypes and the space required to represent the graph.

The time complexity, on the other hand, depends much more on the "shape" of the graph. As suspected, certain input graphs provide much more efficient executions of the discovery algorithm. The major factor in determining the running time of the algorithm is the number of possible transformation paths that must be checked. The time required by FDR grew dramatically as edges were added to the graph, especially when those edges added new transformation paths without making any new datatypes reachable. This yields a portion of the graph where many different interleavings of transformation sequences must be considered. Each of these sequences will eventually yield the same set of available datatypes, but will require different intermediary sets to get there.

In practice, transformation graphs will usually avoid this inefficiency, since the clumps of datatypes in a graph will not be highly interconnected. Indeed, the entire reason for using this graph-based approach is to limit the need to write direct transformations between datatypes. Instead, transformation graphs tend to follow an "intermediary format" pattern, as shown in Figure 8.15. In this pattern, one or more datatypes (the shaded types in the center of the graph) become de facto or official standards within different communities of users and developers. Individual application de-

velopers then write transformations between their datatypes (the unshaded types on the periphery of the graph) and one of these standards. These star-like graphs form trees, where there is exactly one path between any pair of nodes. While a real-world transformation graph is not likely to be a perfect tree (and might therefore allow multiple paths between a certain pair of datatypes), it can still provide full connectivity between all datatypes without providing an overabundance of transformation paths to check.

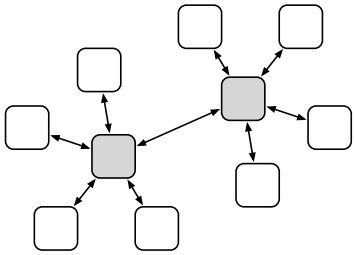


Figure 8.15: Using intermediary datatypes to simplify a transformation graph

A similar factor affecting the algorithm's time complexity is whether the compound transformation that we are seeking actually exists. FDR is able to find a correct transformation much faster than it is able to prove that no transformation exists. Intuitively, this makes sense; once FDR has found a solution, it does not need to consider any of the remaining possibilities and can stop processing. If there is no solution, FDR must check every possible transformation path to prove this. In practice, a program or user invokes the discovery algorithm because they know (or can reasonably assume) that the desired compound transformation exists. For real-world use cases, therefore, the time complexity will tend to be much more efficient than the worst case.

8.3.2 Modifying the algorithm

Having showed in the previous section how the efficiency gains found by FDR should be useful in practice, we now turn our attention to modifying our discovery algorithm to exploit these improvements. All of the efficiency gains in our examples stemmed from one major feature: lazy evaluation. This was only possible with the modified *Finish* and *XformPrereq* processes described in Section 8.2. With the reduced state space of the internal *Have* subprocesses, FDR did not need to store the entirety of a set-based graph. Instead, it encoded a recipe for lazily deriving the appropriate portions of the graph as they become relevant.

This explains why the initial space requirements are fairly independent of the connectivity of the graph: while there might be many more paths that need to be checked, the recipe used to describe the graph remains roughly the same size. The time complexity, however, is highly dependent on the connectivity; when there are more edges, a larger fraction of the set-based graph must be checked to find a solution.

We can illustrate this by revisiting the example set graph from Section 7.2.2, repeated here in Figures 8.16 and 8.17. Examining the set graph closely, we see that only a small fraction of the nodes are reachable from the $\{A, B\}$ source node. In fact, if we look at the connected components of the set graph, as shown in Figure 8.18, we see that the number of reachable nodes (and therefore the number of possible set paths) will always be relatively small. The connected components are defined

entirely by the atomic transformations in the graph, so this is true regardless of the source and sink nodes used.

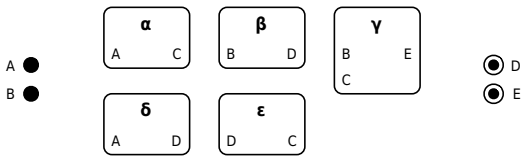


Figure 8.16: An example transformation graph for the discovery algorithm

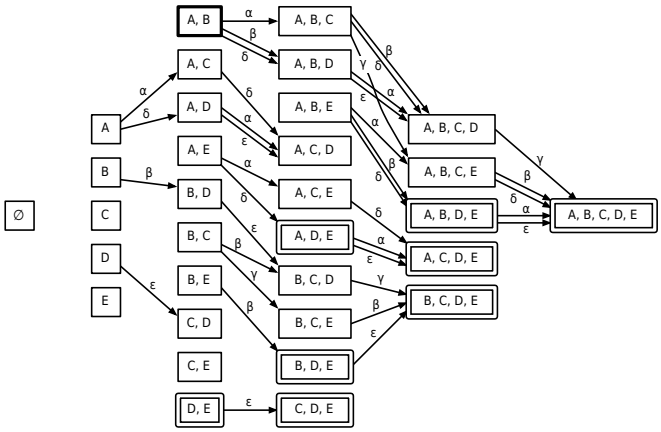


Figure 8.17: The corresponding set graph

By incorporating FDR's lazy evaluation strategy, we can exploit this knowledge. For reference, we repeat the naïve version of the discovery algorithm, as presented in Chapter 7:

1. Translate the polyadic transformation graph into the set notation.
2. Find a shortest path from the set graph's source node to *any* of its sink nodes.
3. Use this path to add dataflow links to the transformation graph's workflow.

To support lazy evaluation in FDR, we had to rewrite the *Have* subprocesses to store a recipe for constructing a set graph, rather than the set graph in its entirety. To support lazy evaluation in our algorithm, we will need a similar recipe. Luckily, we already have one: the original workflow notation encodes this in its representation of atomic transformations as "black boxes" of logic or code. We can therefore interleave the set graph construction of step one with the pathfinding of step two, as follows:

1. Add the source node, and its outgoing transformation edges, to the set graph.
2. Run the pathfinding algorithm, as before. As new nodes are encountered, add them, along with their outgoing transformation edges, to the set graph. As soon as we encounter a shortest path to any sink node (i.e., to any node that is a superset of our desired output types), stop processing.

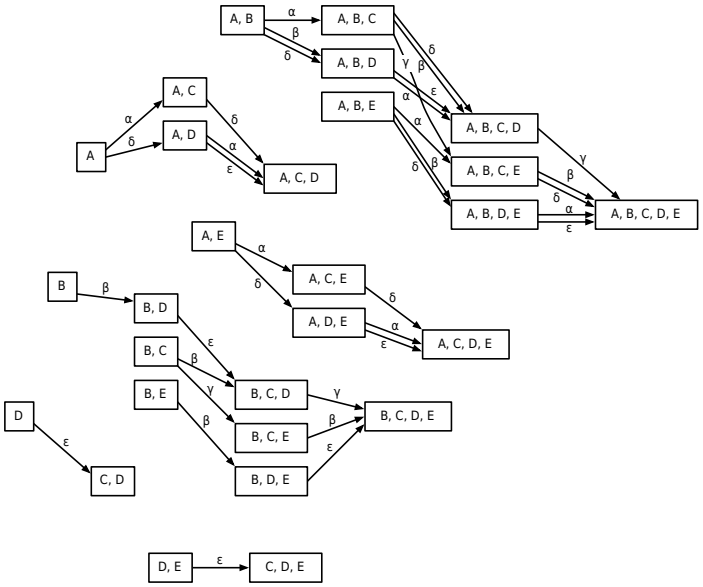


Figure 8.18: The connected components of the example set graph

3. If any such set path is found, use it to construct a corresponding workflow.

With these modifications, we ensure that only set nodes that are actually reachable from the source are instantiated in the set graph. Moreover, nodes that are reachable, but are "further" away than a sink node, will also not be instantiated, since we stop processing as soon as a set path is found. As always, we have not improved the worst-case efficiency of the algorithm: if the set graph is fully connected, for instance, all $O(2^D)$ nodes will be reachable from the source, instantiated in the set graph, and visited by the pathfinder. However, our analysis using FDR leads us to believe that these pathological cases will be rare in practice, and that the lazy instantiation strategy will often be a helpful optimization.

8.4 Limitations of this technique

The technique described in this chapter allowed us to empirically analyze the complexity space of the polyadic transformation discovery algorithm. This technique can be used in a similar way for any algorithm or problem that can expressed as a refinement between two CSP processes, by finding the inputs that can be solved more efficiently by FDR, and searching for common features of those inputs. This would then hopefully provide insights into how the algorithm can be made more efficient for those cases.

One limitation of this technique is that it is highly dependent on our selection of inputs and on the tool that we use. We must choose an appropriate sampling of inputs if we want a true view of the problem's complexity space. More importantly, even if we choose an appropriate suite of test inputs, there is no guarantee that FDR will find all of the efficient solutions that are possible. When FDR finds a compound transformation inefficiently, for instance, this does not mean that this input graph has no efficient solution; instead, it might be that FDR's refinement checking strategies cannot reproduce the necessary optimizations. If we were to use a different modeling formalism and model checker, or

even a different CSP refinement checker, we might get different results for our analysis. Because of this, we cannot establish a tight bound on the problem's complexity using this technique. However, we *can* establish an *upper* bound. Put another way, negative results are not indicative, in general. Positive results, on the other hand, represent real optimizations that can be exploited, though even these results might not be fully optimal.

Our general strategy was to develop a rapid prototype of the algorithm in a declarative style, without having to choose a particular execution strategy for a low-level implementation. If we want more confidence in our view of the problem's complexity space, we could use several prototypes, each using a different underlying language, hoping that each efficient input would be found by at least one of them. SAT solvers, in particular, would be a good choice for an additional prototype; being the earliest and most visible *NP*-hard problem [28], Boolean satisfiability has inspired research into many sophisticated optimization techniques [48, 109, 32, 39].

Summary

In the previous chapter, we presented an algorithm for polyadic transformation discovery and showed that it was *NP*-hard, implying that the underlying problem is inherently difficult. However, this is only a worst-case bound on the algorithm's efficiency; the hope is that for practical examples of transformation graphs, the discovery algorithm would execute more efficiently. To this end, we have developed a declarative CSP description of the discovery problem, allowing us to use the FDR refinement checker to analyze the complexity space of the problem. By using FDR to find compound transformations in graphs of varying sizes and shapes, we have identified several classes of transformation graph that are more efficient. By examining the high-level patterns that these efficient transformation graphs have in common, we identify several design criteria that lead to efficient discovery. Luckily, the transformation graphs that meet these criteria are indeed the ones that are likely to appear in practice.

Given this analysis, we then modified the naïve discovery algorithm from the previous chapter to incorporate the primary optimization used by FDR: lazy evaluation. By only instantiating the nodes in the graph as they are encountered during the pathfinding algorithm, and by stopping the processing once a shortest path has been discovered, we can exploit the same efficiencies in our pathfinding algorithm that FDR exploited in its refinement check. Further, as we have shown, these efficiencies are likely to occur exactly in those transformation graphs that tend to appear as practical, real-world examples. Thus, while we cannot improve the algorithm's worst-case intractability, we have been able to empirically show that a slight modification to the algorithm allows practical transformations to be discovered efficiently and effectively.

9 Discussion and conclusions

In this chapter we summarize the contributions of this thesis. First, we present an overview of the data mismatch problem, highlighting its importance in today's highly connected software systems. We show how existing techniques are not general enough: while being useful *parts* of an overall solution, they do not solve the problem in its entirety. Next, we present our graph-based transformation framework, which can be used to solve the data mismatch problem in a truly generic way. Then, we show how we can extend the expressiveness of this graph model to support more sophisticated use cases, at the cost of sacrificing the efficiency of the transformation discovery algorithm. Next, we use a novel application of the CSP process algebra to empirically analyze the complexity space of the extended discovery algorithm. By doing so, we find graph features that are executed faster than the worst-case upper bound; luckily, these are features that will tend to appear often in practical transformation graphs. Finally, we examine several related research areas, comparing them with the topics covered in this thesis, and we present several remaining open questions, suggesting areas for future work in this area.

9.1 Motivation

As software engineers and computer scientists, we now find ourselves in a world that is heterogeneous in a fundamental way that was not the case even a decade ago. Before, software systems were usually designed and used in isolation. Decisions about architecture, algorithms, and data models could be made strictly based on the needs of the one application. Communication and interoperability were sometimes necessary requirements of a system, but this was the exception, not the rule.

Now, on the other hand, the Internet is ubiquitous, and most software systems are expected to communicate with their peers. As the global network connection becomes more established as a given in our computing infrastructure, the amount and sophistication of this communication can only be expected to increase. While the primary criteria for each design decision is still the benefit to our particular application, we must now be aware of the ramifications of those decisions in the larger context brought about by the Internet.

One particular issue that we must worry about when considering interoperability is how the application's data is modeled and encoded, both internally and externally. Even within a particular

problem domain, where different applications deal with logically similar data, the representations of the data will likely be different. Moreover, these differences can occur at many levels, from high-level semantic differences in how the data is interpreted, through medium-level differences in how data elements are structured and organized, to low-level syntactic differences in how the data is concretely encoded. This leads to the *data mismatch problem*: for different applications to communicate and coexist effectively in today's highly networked environment, we must somehow reconcile these differences, preferrably in an automatic and generic way.

One way to approach the problem is to consider the different *equivalences* that exist between the data models, in addition to the differences between them. Like data mismatches, these equivalences will also appear at many conceptual levels. Two datatypes can be semantically equivalent if they describe the same real-world concepts. They can be structurally equivalent if they are organized using the same primitive structures. They can be syntactically equivalent if they are both physically encoded into a stream or buffer of bytes in the same way. Our task, then, is to find a way to translate between two datatypes, overcoming some mismatch, while at the same time maintaining some equivalence.

Apart from the naïve and tedious task of creating custom translations for each pair of applications, the most obvious solution is to develop and mandate an intermediary format to serve as a *lingua franca* between systems. Applications would be responsible for supporting this intermediary format in addition to whatever custom data models and formats were needed. Unfortunately, while this provides a simple technical solution to mismatched data models, it relies on the developers, users, and other stakeholders of each system to agree on and adhere to the standard. Often this does not happen, and the community balkanizes into several groups centered around competing standards.

Thus, while we need to promote the standardization process whenever possible, we cannot rely on globally mandated standards as a solution. Instead, we must accept that there could be a multitude of data models and formats to support, with various differences and incompatibilities between them. Our solution, then, should be able to incorporate and exploit any standardized intermediaries when they exist, while also gracefully handling situations where they do not.

9.2 Solution

While the data mismatch problem is becoming more urgent with the advent of the ubiquitous Internet, it has been acknowledged as a valid concern for quite some time. As such, it is not surprising that there are many existing approaches that try to reconcile data mismatches automatically, to varying degrees of success. However, these previous approaches all follow the general strategy of requiring a precise description of each datatype — either explicitly written by the developer or derived from actual data. From these descriptions, the relationships between two datatypes are inferred, providing a recipe for translating between them.

Unfortunately, these approaches suffer from two main drawbacks. First, the descriptions supported by a particular technique can only be provided for certain kinds of datatypes: for instance, they might require that the data be stored in relational tables, or be described using an XML schema. Any datatype that does not meet these requirements would not be supported without some initial manual translation. Second, inferring the relationships from the datatype descriptions is not trivial, often requiring some form of natural language processing or manual verification of the results. While this certainly does not render these techniques useless, it can make them more difficult to use in the

larger context of application development.

We take a different approach, and abandon this precise description of the datatypes. Instead, we require that *atomic transformations* be written between certain pairs of datatypes. Obviously, a description of the datatypes — whether more formal, like a relational or XML schema, or more informal, like documentation — will be necessary to write an atomic transformation. However, once an atomic transformation is written, this information can be discarded, since it is not needed by the higher-level transformation framework. Instead, the detailed knowledge of the datatype is encapsulated into the atomic transformation, which is treated as a "black box" of logic or code.

With several of these atomic transformations, we can construct a *transformation graph*, with nodes representing datatypes and edges representing atomic transformations. While the transformation graph gives us a nice model for representing translations between datatypes, they can sometimes be repetitive. Every XML datatype, for instance, will usually have similar transformations for translating between its various internal and external encodings. We can use constructs like *declaration patterns* to make these situations less tedious.

Since the underlying atomic transformations are composable, each path in a transformation graph represents a *compound transformation* between arbitrary datatypes. There are many efficient path-finding algorithms that can then be used to discover compound transformations. Moreover, we can support different use cases, with different criteria for which compound transformations are optimal, by allowing the user to assign numeric weights to the atomic transformations dynamically.

Thus, we have a solution that is automatic and generic. It is not fully automatic, in that the atomic transformations must be written by hand; however, since each atomic transformation can be written in whichever programming or transformation language is most suitable to it, existing transformation code can often be incorporated into a transformation graph as-is. Further, this amount of manual work is much less than would be required to connect each pair of applications directly, or to incorporate the results of existing (and limited) automated techniques into a larger solution.

Our graph-based approach is also *fully* generic — since we do not require a precise description of the datatypes, having encapsulated this knowledge into the atomic transformations that operate on them, there are absolutely no restrictions placed on the datatypes that we support. Relational databases, XML documents, and proprietary binary formats are all supported, as long as someone is able to provide an atomic transformation that can translate each datatype into something else. Our approach even works for abstract datatypes and for infinite datatypes.

9.3 Extension

Our simple graph-based approach is useful, and can be used to solve many interesting transformation use cases. However, it only supports *unary transformations*. It would be helpful to allow the more expressive *polyadic transformations* — those with multiple inputs and outputs — if possible. Polyadic transformations would be especially useful when the internal structure of a datatype can vary over time or use case; by exposing some of this structure in the transformation graph, we can use the transformation discovery algorithm to handle and mitigate some of these differences. Polyadic transformations allow us to expose this detail in a controlled manner, while still retaining the benefits of the encapsulation and opacity of datatypes.

There are several notations that we can use to describe a polyadic transformation graph. The

workflow notation presents an intuitive view of the graph, representing the available atomic transformations as black boxes of code or logic, and showing the flow of data (which might be reusable) through a compound transformation. The *set notation* focuses on which datatypes are available at each step of a compound transformation. This gives us an obvious discovery algorithm, which we can prove finds correct compound transformation solutions; unfortunately, it does this very inefficiently, due to the space requirements for generating and storing a set graph. Finally, the *hypergraph notation* is logically concise, and very familiar due to its similarities with simple unary transformation graphs. We can use this hypergraph notation to show that polyadic discovery is *NP*-hard — rather than that our set-based algorithm was just poorly designed.

9.4 Analysis

The fact that polyadic discovery is *NP*-hard might seem like a discouraging result at first, since it implies that the discovery algorithm cannot possibly execute in a reasonable amount of time for non-trivial transformation graphs. Luckily, this intractability is only a worst-case bound. There are many problems that are similarly intractable, but only for pathological inputs; for "real-world" inputs, they can be much more efficient. The hope is that polyadic discovery is one of these problems.

To investigate this hypothesis, we use a novel technique for analyzing the complexity of the problem. First, we provide a formal, declarative description of the problem as a refinement between two CSP processes. We can then use the FDR refinement checker as a prototype implementation. By performing the refinement check with a wide variety of inputs, we get an empirical view of the complexity space of the problem, looking for possible optimizations. Some of these optimizations require modifications to the CSP definitions, while others are found automatically by FDR's search algorithm.

By examining the inputs that lead to more efficient solutions in FDR, we can identify and classify common features of those input graphs. This is useful for two reasons. First, we can determine which particular graph features lead to efficiency gains, looking at how those features correspond to real-world transformation graphs. In our case, these efficient graphs are ones that will tend to appear more often in practice, lending support to our hypothesis that real-world transformation graphs are not pathologically intractable. Second, by understanding which input graphs are leading to efficient discovery, we can infer which optimizations — such as lazy evaluation — FDR is using in its search. We can then incorporate these same optimizations into our own set-based discovery algorithm. Together, these show that polyadic transformation graphs can still be useful in practice, even if we are unable to remove the worst-case *NP*-hard bound on the discovery algorithm.

9.5 Related work

Having presented a summary of the contributions of this thesis, we can now examine how it relates to other similar research topics. We look at other approaches for solving our problem of interest, as well as other applications of the technique that we use. We examine three research areas in particular: schema matching, and the automated composition of both workflows and Semantic Web Services.

9.5.1 Schema matching

Schema matching refers to another approach for solving the data mismatch problem. Several schema matching techniques were described previously in Chapter 2. The primary difference between schema matching and the approach described in this thesis is that schema matching techniques try to infer relationships between different datatypes from a detailed model of the data. This model might come from a formal description of the data, such as a database or XML schema, or it could be constructed by examining samples of the data. Moreover, the model might focus on the natural-language names of the elements of a datatype (*nominal typing*) or on the relationships between its elements (*structural typing*). However, regardless of its source or form, these techniques require some kind of model to discover a translation between datatypes.

This reliance causes an interesting tradeoff for schema matching techniques. First, it is beneficial for the description of each datatype to be as detailed as possible, since this provides more information from which to infer a translation. However, the extra detail greatly increases the complexity of the search, making it more inefficient to successfully discover a translation. Many of the research efforts in this area seek to reconcile these competing interests.

Our technique avoids this tradeoff, since we do not use a description of the datatypes to infer a translation. Instead, we rely on small units of transformation to be provided by the user of our framework; we then provide a means of combining these small pieces together in a way that requires no deep understanding of their internals. This gives us an effective abstraction barrier between the individual atomic transformations and our composition technique. This abstraction barrier removes the conflicting concerns inherent in schema matching. Our atomic transformations can benefit from as much detail as can be provided for the datatypes involved. However, none of this information is used by the high-level composition logic, preventing the extra detail from affecting the efficiency of the search.

9.5.2 Workflow composition

Having looked at other solutions to the data mismatch problem, we can now examine other similar uses of composition in the field. First, we look at the composition of workflows in the context of Web Services.

In the Web Service world, workflows can be used to describe higher-level interactions between Web Services. Many languages — including the Business Process Execution Language [60], the Business Process Modeling Notation [76], and the Web Service Choreography Interface [3] — have been proposed for describing these workflows at varying levels of abstraction and complexity. There are existing proposals [88, 19, 97] for automatically discovering and generating new workflows, and for combining existing workflows, based on declarative descriptions of the goals of the application. One would hope that these existing techniques could be used to discover translations between datatypes, since data transformations are a simple specialization of the generic class of computation supported by Web Services.

However, Web Service workflows are too complex for our needs. Web Services are very generic, and support a huge range of interactions. This requires a correspondingly sophisticated workflow language to be able to model the rich interactions between services. They need to represent complex sequences of messages and operations, between multiple parties, with non-trivial control flow

semantics. Along these lines, UML activity diagrams have been proposed [100] as one possibility for modeling and expressing these complex interactions. Regardless of their particular form, any Web Service language for describing and composing workflows will approach the same level of complexity and sophistication as a full-fledged process algebra like CSP.

Our transformation technique operates in a much simpler model, even when considering polyadic transformations. Atomic transformations are functional in nature, having no side effects. They have very simple interfaces, taking the data to translate as input and generating the translation as output. This allows us to model the composition of transformations using only dataflow links; no complex control flow structures are needed. We can exploit this simple structure to reduce the search space we must examine when looking for compound transformations, greatly increasing the efficiency of our technique compared to Web Service composition frameworks. In the extreme case of unary transformations, the dataflow diagram reduces to a simple directed graph, giving us a very efficient pathfinding algorithm for discovering transformations.

9.5.3 Semantic Web Service composition

One relatively recent addition to the Web Service stack of standards and protocols attempts to integrate the ontological descriptions of the Semantic Web with the lower-level transport and service interface descriptions of Web Services. Dubbed *Semantic Web Services* [72, 105], this paradigm assumes that a Web Service interface description will describe not only the low-level details of a service, such as its name and the XML types of its inputs; it will also provide a formal description, expressed in an extension to OWL [70, 101, 69], of the semantics of those inputs, and of how its outputs logically relate to its inputs. One application of these machine-readable semantics is to support composing these services automatically with respect to some larger specification [71, 23]. If we could use Semantic Web primitives to express the translation behavior required of a solution to the data mismatch problem, we could possibly use these semantics-based composition techniques to discover data transformations.

In one sense, our technique also claims to include "semantic meaning" in our transformation solutions, since they purportedly operate on datatypes defined across the full range of the S classification. We have given several examples of datatypes that we must translate between, whose differences exist solely at the semantic level — the raw and deconvolved images from Section 3.1.3 being a prime example. One of the main inefficiencies in Semantic Web-based algorithms is the need to encode this semantic meaning in a form that is, at least to some extent, "understandable" by a machine. Because of our strict adherence to datatype opacity, we do not suffer from this inefficiency. In our framework, it is perfectly acceptable for this semantic meaning to live solely in the minds of the human developers who write and use the transformations in our system. While not directly understandable by (or even necessarily represented in) a machine, this semantic meaning can still inform the decisions made about the datatypes and transformations, which can have a visible influence on the results produced by our high-level transformation discovery logic. As with most other aspects of our system, the information is available for use in the low-level transformation definitions, while encapsulation prevents the extra detail from reducing the efficiency of our algorithms.

Semantic Web-based service descriptions are also much more expressive, since they have the full range of propositional logic available for describing the semantics of a service operation. Services are

usually modeled as implications, where the truth of an operation's precondition implies the truth of its postcondition. The composition of services is modeled as a conjunction of the two corresponding implications, with an appropriate renaming of variables to model the sharing or passing of data between the two. The hypothesis that a valid composite service exists is then naturally modeled as an existential quantification. One can also add other conjunctive clauses to the quantification to provide further details of the requirements of the desired composite service. This approach requires the use of a theorem prover to show that this quantification can be satisfied.

Our approach also supports complex specifications of the desired compound transformation, but as usual, we add several constraints to make the discovery more efficient. Specifically, our pathfinding-based approach requires that the declarative specification of the output be expressible as a simple induction over the atomic transformations that make up the solution. This is easiest to see in the case of data transformation, where a compound transformation maintains any equivalence that is maintained by all of its constituent members. However, our discovery algorithm works for any kind of computation whose semantics can be described as an induction. This is analogous to the different cost metrics that can be used to find hyperpaths: as long as one correctly limits the amount of information considered at each step, the problem is tractible.

9.6 Discussion

In this section, we discuss some interesting features, false starts, and personal observations that occurred while developing this thesis.

When first developing the polyadic extension in Chapter 7, the author examined many other graph problems as a possible basis before settling on the workflow and set notations. For instance, it seemed possible for quite some time to cast polyadic transformation discovery as an example of a maximum flow problem, with the data itself being the commodity flowing through the graph. In its simplest incarnation, the nodes in the graph would represent datatypes. An atomic transformation would be represented by a gadget of nodes and edges that would draw some amount of flow from the transformation's input nodes, and deposit the flow in its output nodes. Of course, this formulation did not work. The fundamental issue has already been described previously — we must require *all* of a transformation's inputs to be available before the transformation can execute, which is difficult to require of a flow graph. Furthermore, the outputs may or may not be needed, depending on the downstream elements of the transformation graph. Not all of these issues are insurmountable, but overcoming them makes the flow graph more complex than it needs to be; casting the problem in terms of the workflow, set, and hypergraph notations provides a much more intuitive view of the problem.

Another interesting feature of our transformation discovery framework is that it does not rely on a standardization process, or on a single formalism or data modeling technique. Many research projects provide formalisms that can be used to model any feasible application data model, which is certainly a necessary part of any generic framework. However, while one can find some instantiation of the formalism that is equivalent to a particular data model, it is not necessarily a direct representation of the data model itself. In the case of data transformations, this is not an acceptable restriction — we cannot claim to support transformations for a particular datatype if the user must first manually translate that datatype into some other representation. Similarly, we cannot rely on a centralized

standards body to provide a solution, since we cannot impose this solution on all interested parties. They may or may not have valid technical reasons for avoiding the standardized solution. While it is tempting to not support those parties who do not buy into the standardization process, we feel that this is an unnecessary restriction, since their reasons might be perfectly valid. In the ideal case, the framework would work even with clients and users that cannot, or choose not to, directly use or support it. As pointed out several times throughout this thesis, we consider this to be a key feature of a framework that aims to be useful in today's highly networked world.

Finally, the author found the algorithm analysis of Chapter 8 to be a particularly exciting application of the CSP process algebra. This use of CSP and FDR falls well outside the scope of its original intended use: the specification of complex parallel and distributed systems. The concept of refinement, which the theory of CSP is based on, is useful in a wide range of other areas, though, and it was gratifying to be able to show that the existing tool support could extend so easily to such a radically different problem domain. This is a testament to the robustness and soundness of Hoare's theory and Roscoe's tool. The only drawback to this line of inquiry was due to the proprietary nature of the FDR program. Being at Oxford, the author was privileged to have access to several of the developers and designers behind FDR, who were an invaluable aid while examining the transformation discovery algorithm. Other researchers without this access might have been hard-pressed to correctly interpret the results.

This analysis was also an interesting example of what one of the examiners termed "experimental computer science". Especially in the area of algorithm design, most work of this nature is firmly grounded in the realm of theory. New algorithms are implemented in order to help show their utility, but the efficiency of an algorithm is usually shown formally as a mathematical proof. We have followed this strategy ourselves in Chapter 7, by using a reduction to hypergraphs to show that the polyadic discovery problem is *NP*-hard. It is more rare to perform an empirical analysis of an algorithm, as we have done in Chapter 8. Part of this could be due to aesthetic reasons, where a formal proof seems to carry more weight due to its finality and rigor, but there are legitimate logistical reasons, too — in order to empirically test an algorithm in this way, one must have at least a prototype implementation of it. Especially in the initial stages of the development of a new algorithm, when certain design decisions have not been finalized, it can be difficult to implement a prototype that will be true to the end result — meaning that any measurements we make will be inaccurate. Moreover, if the development of the prototype is time-consuming in its own right, it can be too expensive to devote time to it. This makes it very difficult to follow an iterative design approach if we want to incorporate this kind of empirical analysis into the process.

In our case, the analysis was not cumbersome because of the approach we took. The prototype "implementation" was not really an implementation as such; because we used a process algebra to model the problem, our description of it was very declarative in nature. While this did require a working knowledge of CSP, which can be a subtle and difficult language to master, the resulting formal description of the problem is very concise, and maps very well to an English description of the problem. More importantly, though, the formal description is *directly executable*, due to the existence of the FDR refinement checker. This makes it very easy to experiment with different design decisions in a very agile way, while allowing us to have immediate feedback about the relative efficiency of those decisions. Of course, as we mentioned in Chapter 8, we are not able to find a tight bound on the efficiency, since FDR might not always find optimal solutions, but it is helpful nonetheless.

9.7 Future work

In this final section, we consider the many tasks and open research questions that remain. The most obvious is that, while we have described a theoretical framework for our graph-based approach to transformation discovery, there is currently only a basic prototype implementation of this technique. To be useful in a wider context, we would need to create a well-engineered library implementing these ideas. For unary graphs, this should be fairly simple, as the theory only relies on well-established, well-known graph algorithms. The most difficult engineering task, in fact, would likely be supporting atomic transformations written in a wide range of programming and transformation languages. The polyadic discovery algorithm is slightly more complex, in that its implementation focuses on the new set notation, though it should still be relatively straightforward to develop.

There are several open research questions that involve our empirical analysis of the polyadic discovery algorithm. First, as we mentioned in Chapter 8, we can repeat our experiments using another declarative description of the problem (expressed, for instance, as a Boolean satisfiability [28] problem), along with another corresponding prototype implementation. This would ideally give us more confidence that our view of the problem's complexity space is accurate. Second, there has been a lot of research into compression and optimization techniques for CSP processes. The supercompilation approach described in [47] is integral to the lazy evaluation strategy that we have already exploited. The hierarchical compression functions described in [94] seem promising, as well. It would be fruitful to see if any of these CSP compressions could be used to obtain further optimizations. Finally, this technique could be similarly used for any algorithm or problem that can expressed as a refinement of CSP processes, by finding the inputs that are solved more efficiently by FDR, and searching for common features of those inputs. This would then hopefully provide insights into how the algorithm could be made more efficient for those cases. One could verify the general utility of this technique by applying it to several well-known *NP*-hard problems, seeing if it can reproduce existing results and lead to new insights.

There are also open questions that pertain directly to our graph-based transformation approach. First, our use of a transformation graph to encode the known translations between datatypes is centered within the context of a single application. This might seem counterintuitive, since the purpose of our framework is to support communication betweeen multiple applications; however, as a simplification, we take the view that only one of the applications — the one with the transformation graph — is responsible for mediating this communication. The situation becomes more complicated when both communicating applications have transformation graphs at their disposal.

A simple solution would be to retain the restriction that only one application mediates the communication. If both applications have transformation graphs available, then they use some negotiation protocol (or a predefined set of rules) to determine which application's transformation graph is used. The other application would then act as if it could only communicate using its own application-specific datatypes. This solution might not be optimal, however, since there is no guarantee that either transformation graph on its own will be able to find the "best" transformation between the two native datatypes.

A more interesting solution would attempt to merge the two transformation graphs together; this assumes that the combined graph could find a more optimal compound transformation. (It could certainly do no worse, since it would fully contain both individual graphs, and therefore retain all of

the possible solutions of each.) We would probably need to ensure that any compound transformation crosses the "boundary" between the two constituent graphs only once, representing the fact that we only want to send a single data instance across the wire between the two applications — though it might also be interesting to see if we can model the tradeoffs between a "chatty" connection on the wire and the efficiency of the individual atomic transformations. Other interesting topics in this area include how to combine the two weighting functions if the applications have different optimality criteria for the compound transformation; and how to efficiently assemble the combined transformation graph when the applications are separated by a (possibly slow or faulty) network connection.

One final research question concerns how a transformation framework like this fits into larger architectural design paradigms. The various flavors of service-oriented architecture (SOA) seem the most complementary, as they are also designed to support communication between decoupled, heterogeneous software systems. The idea of a multitude of datatypes, which we can dynamically and automatically transform between, seems to fit especially well with the REST notion of different representations of a resource [40]. One can easily see a REST client using a transformation graph to automatically translate the resource representation provided by the server into something more understandable; equally, one can see a server using a transformation graph to seamlessly support a wider range of output datatypes.

There are two slight inconsistencies, neither of which are insurmountable, with how REST datatypes are defined in terms of the MIME standard [43, 44]. First, MIME types are mostly — and preferrably — assigned by a central authority, which does not match well with our very decentralized notion of datatype definitions. The frameworks would be more consistent if REST used URIs [14] (or any other decentralized, globally unique identifier scheme) for datatype names. The second difference is that MIME types tend to focus only on the syntactic and structural levels of the S classification — with `image/jpeg` and `application/xml`, which are purely syntactic, being examples of common MIME types. Structural types are also possible, the most common being schema-specific types for different XML formats. However, MIME types cannot really support more subtle distinctions between datatypes, such as structurally and syntactically identical datatypes that must be interpreted according to different semantics; nor do they support our policy of using separate datatypes to represent any distinguishable differences between datatypes. Again, though, these differences are not fundamental inconsistencies, and could easily be overcome if one were to add transformation graph capabilities to a REST application.

Similar issues arise when considering transformation graphs in the context of Web Services. One could envision using WSDL [22] to describe a service that accepts or generates many possible datatypes by incorporating a transformation graph within it. As a more intriguing example, one could see a Web Service client using a transformation graph to seamlessly connect to a wide range of services, each with a slightly different contract regarding input and output datatypes, bringing us closer to the dream of truly dynamic (and automatic) service discovery. As with REST, there are some slight inconsistencies, most notably with how the Web Service protocols assume that every datatype will be syntactically encoded using XML. Again, though, none of these issues seem insurmountable.

Summary

In this thesis we have presented a framework for discovering data transformations that is automated and fully generic. Instead of relying on precise descriptions of each datatype, we require a small number of atomic transformations to be written manually. These can be formed into a graph, in which compound transformations between arbitrary datatypes are represented by paths. Thus, we can use any efficient pathfinding algorithm to discover arbitrary data transformations. This approach is very efficient, and works well when the datatypes in question are relatively static. If the internal structure of the datatypes can vary quickly with time or over use case, though, it can be helpful to expose some of this internal structure to the transformation discovery layer. We can model this, and other sophisticated transformation graphs, using polyadic transformations, which have multiple inputs and outputs. At the cost of efficiency, we can extend our graph model to support polyadic transformations. In this extended model, we can create a naïve exponential-space discovery algorithm; moreover, we can show that the problem is *NP*-hard, and therefore inherently difficult. By developing a declarative description of the problem in CSP, we can empirically analyze the complexity space of the problem. This shows that there is an interesting class of inputs, which luckily correspond to "real-world" polyadic transformation graphs, where polyadic discovery is more efficient. We therefore have a transformation framework that is very efficient when we only have transformations with one input and output; when we must support transformations with multiple inputs and outputs, our framework is worst-case intractable, though still useful in practice.

APPENDIX

Z utility library

This appendix contains definitions of several Z types, schemas, and functions that are used in formalizations throughout this thesis.

A.1 Bags and sets

In this section we provide two simple functions for converting between bags and sets. Any bag can be converted into a set by removing duplicates; since bags are represented in Z by mapping each element to the number of occurrences of that element, the domain of the bag is the same as the bag's corresponding set.

Definition A.1 *(Converting bags to sets).*

$$[X]$$
$$\mathrm{set}\ \_ : \mathrm{bag}\ X \rightarrow \mathbb{P}\ X$$
$$\forall\ xs : \mathrm{bag}\ X \bullet \mathrm{set}\ xs = \mathrm{dom}\ xs$$

Similarly, since every element of a set occurs exactly once, a set can be converted into a bag by simply mapping each element of the set to one.

Definition A.2 *(Converting sets to bags).*

$$[X]$$
$$\mathrm{bag}\ \_ : \mathbb{P}\ X \rightarrow \mathrm{bag}\ X$$
$$\forall\ xs : \mathbb{P}\ X \bullet \mathrm{bag}\ xs = \{\ x : xs \bullet x \mapsto 1\ \}$$

A.2 Binary strings

In this section, we present a formalization of binary strings; these functions are used in the formal description of the integer datatypes described in Chapter 3.

Any description of binary data must first define *bits*. Bits are simple — there are exactly two of them: 0 and 1. We can also define a *bit string*, which is an ordered sequence of bits.

Definition A.3 *(Bits).*

$$Bit ::= 0 \mid 1$$
$$BitString == \mathrm{seq}\ Bit$$

We will often refer to bit strings of a particular length. We can define a Bits function to help with this, which returns the set of all bit strings of the specified length.

Definition A.4 *(Bit strings of bounded length).*

$$\mathrm{Bits}\ \_ : \mathbb{N} \rightarrowtail \mathbb{P}\ BitString$$
$$\mathrm{Bits}\ n = \{\, b : BitString \bullet \#b = n \,\}$$

We will also often need to translate a binary string into its integer equivalent. We assume that the most-significant bit is the first element of the bit string's sequence, so that the bit string appears in an intuitive order when rendered on the page. (This is not to be confused with interpreting integer datatypes; this is a low-level helper function to get the decimal value of a base-2 integer.)

Definition A.5 *(Bits as integers).*

$$\mathrm{int}_{\mathrm{Bit}}\ \_ : Bit \rightarrowtail \mathbb{N}$$
$$\mathrm{int}_{\mathrm{Bit}}\ 0 = 0$$
$$\mathrm{int}_{\mathrm{Bit}}\ 1 = 1$$

Definition A.6 *(Bit strings as integers).*

$$\mathrm{int}_{\mathrm{Bits}}\ \_ : BitString \to \mathbb{N}$$
$$\mathrm{int}_{\mathrm{Bits}}\ \langle\rangle = 0$$
$$\forall\ b : Bit;\ bin : BitString \bullet \mathrm{int}_{\mathrm{Bits}}\ bin \frown \langle b \rangle = (\mathrm{int}_{\mathrm{Bits}}\ bin) \times 2 + (\mathrm{int}_{\mathrm{Bit}}\ b)$$

$$\mathrm{int}_{\mathrm{Bits}}\ \langle 10010011 \rangle = 147$$

This allows us to define a *byte*, which is an 8-bit value, and a *byte string*, which is an ordered sequence of bytes. We also define a Bytes function which returns the set of all byte strings of a given length.

Definition A.7 *(Bytes and byte strings).*

$$Byte == \mathrm{Bits}\ 8$$
$$ByteString == \mathrm{seq}\ Byte$$

Definition A.8 *(Byte strings of bounded length).*

$$\mathrm{Bytes}\ \_ : \mathbb{N} \rightarrowtail \mathbb{P}\ ByteString$$
$$\mathrm{Bytes}\ n = \{\, b : ByteString \bullet \#b = n \,\}$$

When referring to literal byte strings, one of two notations will be used. When expressing the bytes in the string by their numeric (specifically hexadecimal) values, the ⟪48 69⟫ notation will be used. When expressing the bytes by their corresponding ASCII characters, the "Hi" notation will be used.

$$\langle\!\langle 48\ 69 \rangle\!\rangle = \text{“Hi”} = \langle \langle 01001000 \rangle, \langle 01101001 \rangle \rangle$$

The integer representation of a byte string is more complicated, because we must contend with signedness and endianness issues. The simplest case to define is the unsigned, big-endian version, since we can use distributed concatenation to turn the big-endian byte string into an equivalent big-endian bit string.

Definition A.9 *(Unsigned integers).*

$$\begin{array}{|l} \mathsf{unsigned} : \mathit{ByteString} \to \mathbb{N} \\ \hline \forall\, b : \mathit{ByteString} \bullet \mathsf{unsigned}\ \ b = \mathsf{int}_{\mathsf{Bits}}\ (\frown\!\!/\ b) \end{array}$$

Definition A.10 *(Big-endian integers).*

$$\begin{array}{|l} \mathsf{bigEndian} : \mathit{ByteString} \to \mathit{ByteString} \\ \hline \mathsf{bigEndian} = \mathsf{id}\ \mathit{ByteString} \end{array}$$

For signed integers expressed in two's complement, the most-significant bit (which is the first bit in the sequence) will have the value -2^{n-1} instead of 2^{n-1}, but all of the other bits will have the same value as in the unsigned case.

Definition A.11 *(Signed integers).*

$$\begin{array}{|l} \mathsf{signed} : \mathit{ByteString} \to \mathbb{N} \\ \hline \mathsf{signed}\ \langle\rangle = 0 \\ \forall\, b : \mathit{ByteString} \mid \#b > 0 \bullet \\ \quad \textbf{let}\ \mathit{bits} == \frown\!\!/\ b \bullet \\ \qquad \mathsf{signed}\ \ b = \\ \qquad\quad (\mathsf{int}_{\mathsf{Bits}}\ (\mathit{tail}\ \mathit{bits})) - (\mathsf{int}_{\mathsf{Bit}}\ (\mathit{head}\ \mathit{bits})) * 2^{(\#\mathit{bits}-1)} \end{array}$$

Finally, for little-endian byte strings, we can just reverse the byte string before passing it on to the unsigned or signed functions.

Definition A.12 *(Little-endian integers).*

$$\begin{array}{|l} \mathsf{littleEndian} : \mathit{ByteString} \to \mathit{ByteString} \\ \hline \mathsf{littleEndian} = \mathit{rev} \end{array}$$

With these definitions, we can evaluate a particular byte string in all four cases:

$$\begin{array}{l} \langle\!\langle 93\ E8 \rangle\!\rangle \qquad\qquad\ = \langle \langle 10010011 \rangle, \langle 11101000 \rangle \rangle \\ \mathsf{bigEndian}\ \langle\!\langle 93\ E8 \rangle\!\rangle \ = \langle \langle 10010011 \rangle, \langle 11101000 \rangle \rangle \\ \mathsf{littleEndian}\ \langle\!\langle 93\ E8 \rangle\!\rangle = \langle \langle 11101000 \rangle, \langle 10010011 \rangle \rangle \\ \\ \frown\!\!/\ \mathsf{bigEndian}\ \langle\!\langle 93\ E8 \rangle\!\rangle \ = \langle 1001001111101000 \rangle \\ \frown\!\!/\ \mathsf{littleEndian}\ \langle\!\langle 93\ E8 \rangle\!\rangle = \langle 1110100010010011 \rangle \\ \\ \mathsf{unsigned}\ \mathsf{bigEndian}\ \langle\!\langle 93\ E8 \rangle\!\rangle \ = 37864 \\ \mathsf{unsigned}\ \mathsf{littleEndian}\ \langle\!\langle 93\ E8 \rangle\!\rangle = 59539 \\ \mathsf{signed}\ \mathsf{bigEndian}\ \langle\!\langle 93\ E8 \rangle\!\rangle \ = -27672 \\ \mathsf{signed}\ \mathsf{littleEndian}\ \langle\!\langle 93\ E8 \rangle\!\rangle \ = -5997 \end{array}$$

Bibliography

[1] Adobe Systems. *TIFF: Revision 6.0*. 1992. https://partners.adobe.com/public/developer/en/tiff/TIFF6.pdf.

[2] G Alonso, F Casati, H Kuno, and V Machiraju. *Web Services: Concepts, architectures and applications*. Springer-Verlag, Berlin, 2004.

[3] A Arkin, A Askary, S Fordin, et al., editors. *Web Service Choreography Interface (WSCI) 1.0*. Worldwide Web Consortium, August 2002. http://www.w3.org/TR/wsci/.

[4] G Ausiello, A D'Atri, and D Saccà. Graph algorithms for functional dependency manipulation. *Journal of the ACM*, **30**(4), pp. 752–766, October 1983.

[5] G Ausiello, PG Franciosa, and D Frigioni. Directed hypergraphs: Problems, algorithmic results, and a novel decremental approach. In *7th Italian Conference on Theoretical Computer Science*, *LNCS* **2202**, pp. 312–328, Berlin, October 2001. Springer-Verlag.

[6] G Ausiello, GF Italiano, and U Nanni. Optimal traversal of directed hypergraphs. Technical Report TR-92-073, ICSI, Berkeley, CA, September 1992.

[7] G Ausiello, GF Italiano, and U Nanni. Hypergraph traversal revisited: Cost measures and dynamic algorithms. In *23rd Int'l Symposium on the Mathematical Foundations of Computer Science*, *LNCS* **1450**, pp. 1–16, Berlin, August 1998. Springer-Verlag.

[8] D Beckett, editor. *RDF/XML Syntax Specification*. Worldwide Web Consortium, February 2004. http://www.w3.org/TR/rdf-syntax-grammar/.

[9] R Bellman. On a routing problem. *Quarterly of Applied Mathematics*, **16**(1), pp. 87–90, 1958.

[10] C Berge. *Graphs and hypergraphs*, *North-Holland Mathematical Library* volume 6. Elsevier, Amsterdam, 1973.

[11] C Berge. *Hypergraphs: Combinatorics of finite sets*, *North-Holland Mathematical Library* volume 45. Elsevier, Amsterdam, 1989.

[12] T Berners-Lee, R Cailliau, JF Groff, and B Pollermann. World-Wide Web: The information universe. *Electronic Networking: Research, Applications and Policy*, **1**(2), pp. 74–82, 1992.

[13] T Berners-Lee, R Cailliau, A Luotonen, HF Nielsen, and A Secret. The World-Wide Web. *Communications of the ACM*, **37**(8), pp. 76–82, August 1994.

[14] T Berners-Lee, RT Fielding, and L Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986. Internet Engineering Task Force, January 2005. http://tools.ietf.org/html/rfc3986.

[15] T Berners-Lee, J Hendler, and O Lassila. The Semantic Web. *Scientific American*, **284**(5), pp. 28–37, May 2001.

[16] J Boyer. *Canonical XML*. Worldwide Web Consortium, March 2001. http://www.w3.org/TR/xml-c14n/.

[17] J Boyer, DE Eastlake, and J Reagle. *Exclusive XML Canonicalization*. Worldwide Web Consortium, July 2002. http://www.w3.org/TR/xml-exc-c14n/.

[18] T Bray, J Paoli, CM Sperberg-McQueen, E Maler, and F Yergeau, editors. *Extensible Markup Language*. Worldwide Web Consortium, February 2004. http://www.w3.org/TR/REC-xml/.

[19] F Casati, S Ilnicki, L Jin, V Krishnamoorthy, and MC Shan. Adaptive and dynamic service composition in eFlow. In *12th Int'l Conference on Advanced Information Systems Engineering*, *LNCS* **1789**, pp. 13–31, Berlin, 2000. Springer-Verlag.

[20] S Castano, VD Antonellis, and S De Capitani de Vimercati. Global viewing of heterogeneous data sources. *IEEE Trans. on Knowledge and Data Engineering*, **13**(2), pp. 277–297, March/April 2001.

[21] C Catley and M Frize. Design of a health care architecture for medical data interoperability and application integration. In *2nd Joint EMBS/BMES Conference*, volume 3, pp. 1952–1953. IEEE, October 2002.

[22] E Christensen, F Curbera, G Meredith, and S Weerawarana. *Web Services Description Language (WSDL) 1.1*. Worldwide Web Consortium, March 2001. http://www.w3.org/TR/wsdl/.

[23] SA Chun, Y Lee, and J Geller. Ontological and pragmatic knowledge management for Web Service composition. In *9th Int'l Conference on Database Systems for Advanced Applications*, *LNCS* **2973**, pp. 365–373, Berlin, February 2004. Springer-Verlag.

[24] J Clark, editor. *XSL Transformations (XSLT)*. Worldwide Web Consortium, November 1999. http://www.w3.org/TR/xslt/.

[25] J Clark, editor. *RELAX NG Compact Syntax*. OASIS, November 2002. http://relaxng.org/compact-20021121.html.

[26] J Clark and M Murata, editors. *RELAX NG Specification*. OASIS, December 2001. http://relaxng.org/compact-20011203.html.

[27] EF Codd. A relational model of data for large shared data banks. *Communications of the ACM*, **13**(6), pp. 377–387, June 1970.

[28] SA Cook. The complexity of theorem-proving procedures. In *3rd ACM Symposium on the Theory of Computing*, pp. 151–158, New York, 1971. ACM Press.

[29] DA Creager and AC Simpson. A fully generic, graph-based approach to data transformation discovery. In *Graph Computation Models (GCM06)*, September 2006.

[30] DA Creager and AC Simpson. Towards a fully generic theory of data. In *Int'l Conf. on Formal Engineering Methods (ICFEM06)*, *LNCS* **4260**, pp. 304–323, November 2006.

[31] DA Creager and AC Simpson. Empirical analysis and optimization of an *NP*-hard algorithm using CSP and FDR. In *Brazilian Symposium on Formal Methods*, August 2007. To appear.

[32] M Davis, G Logemann, and D Loveland. A machine program for theorem-proving. *Communications of the ACM*, **5**(7), pp. 394–397, July 1962.

[33] DCMI Usage Board. DCMI Metadata Terms, January 2005. http://dublincore.org/documents/dcmi-terms/.

[34] EW Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, **1**(1), pp. 269–271, December 1959.

[35] A Doan, P Domingos, and AY Halevy. Reconciling schemas of disparate data sources: A machine-learning approach. *ACM SIGMOD Record*, **30**(2), pp. 209–520, June 2001.

[36] A Doan, P Domingos, and A Levy. Learning source descriptions for data integration. In *Proceedings of WebDB Workshop*, pp. 81–86, 2000.

[37] D Eastlake, J Reagle, and D Solo, editors. *XML-Signature Syntax and Processing*. Worldwide Web Consortium, February 2002. http://www.w3.org/TR/xml-dsigcore/.

[38] ECMA International. *ECMA 376: Office Open XML File Formats*. ECMA International, Geneva, December 2006.

[39] N Eén and N Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing*, *LNCS* **2919**, pp. 502–518. Springer-Verlag, Berlin, 2004.

[40] RT Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[41] RT Fielding, J Gettys, J Mogul, H Frystyk, L Masinter, P Leach, and T Berners-Lee. *Hypertext Transfer Protocol — HTTP/1.1*. RFC 2616. Internet Engineering Task Force, June 1999. http://tools.ietf.org/html/rfc2616.

[42] LR Ford, Jr. and DR Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[43] N Freed and N Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part one: Format of Internet message bodies*. RFC 2045. Internet Engineering Task Force, November 1996.

[44] N Freed and N Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part two: Media types*. RFC 2046. Internet Engineering Task Force, November 1996.

[45] G Gallo, G Longo, and S Pallottino. Directed hypergraphs and applications. *Discrete Applied Mathematics*, **42**(2), pp. 177–201, April 1993.

[46] IG Goldberg, PK Sorger, JR Swedlow, et al. The Open Microscopy Environment (OME) data model and XML file: Open tools for informatics and quantitative analysis in biological imaging. *Genome Biology*, **6**(5), May 2005. http://genomebiology.com/2005/6/5/R47.

[47] M Goldsmith. Operational semantics for fun and profit. In AE Abdallah, CB Jones, and JW Sanders, editors, *Communicating Sequential Processes: The First 25 Years*, *LNCS* **3525**, pp. 265–274. Springer-Verlag, 2005.

[48] J Gu, PW Purdom, J Franco, and BW Wah. *Algorithms for the satisfiability (SAT) problem: A survey*. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1997.

[49] M Gudgin, M Hadley, N Mendelsohn, JJ Moreau, HF Nielsen, A Karmarkar, and Y Lafon, editors. *SOAP Version 1.2 Part 1: Message framework*. Worldwide Web Consortium, April 2007. http://www.w3.org/TR/soap12-part1/.

[50] M Gudgin, M Hadley, and T Rogers, editors. *Web Services Addressing 1.0 — Core*. Worldwide Web Consortium, May 2006. http://www.w3.org/TR/ws-addr-core/.

[51] M Hardwick, DL Spooner, T Rando, and KC Morris. Data protocols for the industrial virtual enterprise. *IEEE Internet Computing*, **1**(1), pp. 20–29, January 1997.

[52] CAR Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[53] innoQ Deutschland GmbH. Web Services standards as of Q1 2007. Online poster, version 3.0, February 2007. http://www.innoq.com/soa/ws-standards/poster/.

[54] International Organization for Standardization. *ISO/IEC 10918-1:1994: Information technology — Digital compression and coding of continuous-tone still images: Requirements and guidelines*. ISO, Geneva, 1994.

[55] International Organization for Standardization. *ISO 4217:2001: Codes for the representation of currencies and funds*. ISO, Geneva, 2001.

[56] International Organization for Standardization. *ISO/IEC 8824-1:2002: Abstract Syntax Notation One (ASN.1): Specification of basic notation*. ISO, Geneva, 2002.

[57] International Organization for Standardization. *ISO/IEC 19501:2005: Information technology — Open distributed processing — Unified Modeling Language (UML) version 1.4.2*. ISO, Geneva, 2005.

[58] International Organization for Standardization. *ISO/IEC 26300:2006: Information technology — Open Document Format for Office Applications*. ISO, Geneva, 2006.

[59] GF Italiano and U Nanni. On-line maintenance of minimal directed hypergraphs. In *3rd Italian Conf. on Theoretical Computer Science*, pp. 335–349. World Scientific Co., 1989.

[60] D Jordan and J Evdemon, editors. *Web Services Business Process Execution Language: Version 2.0*. OASIS, April 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[61] G Klyne and JJ Carroll, editors. *Resource Description Framework (RDF): Concepts and Abstract Syntax*. Worldwide Web Consortium, February 2004. http://www.w3.org/TR/rdf-concepts/.

[62] Laboratory for Optical and Computational Instrumentation (LOCI), University of Wisconsin. OME-TIFF. http://loci.wisc.edu/ome/ome-tiff.html.

[63] A Le Hors, P Le Hégaret, L Wood, G Nicol, J Robie, M Champion, and S Byrne. *Document Object Model (DOM) Level 3 Core Specification*. Worldwide Web Consortium, April 2004. http://www.w3.org/TR/DOM-Level-3-Core/.

[64] WS Li and C Clifton. Semantic integration in heterogeneous databases using neural networks. In *20th Int'l Conference on Very Large Data Bases (VLDB)*, pp. 1–12, San Francisco, 1994. Morgan Kaufmann.

[65] WS Li and C Clifton. SemInt: A tool for identifying attribute correspondences in heterogeneous databases using neural networks. *Data and Knowledge Engineering*, **33**(1), pp. 49–84, April 2000.

[66] WS Li, C Clifton, and SY Liu. Database integration using neural networks: Implementation and experiences. *Knowledge and Information Systems*, **2**(1), pp. 73–96, March 2000.

[67] J Madhavan, PA Bernstein, and E Rahm. Generic schema matching with Cupid. In *27th Int'l Conference on Very Large Data Bases (VLDB)*, pp. 49–58, San Francisco, 2001. Morgan Kaufmann.

[68] F Manola. Interoperability issues in large-scale distributed object systems. *ACM Computing Surveys*, **27**(2), pp. 268–270, June 1995.

[69] D Martin, editor. *OWL-S: Semantic markup for Web Services*. Worldwide Web Consortium, November 2004. http://www.w3.org/Submission/OWL-S/.

[70] DL McGuinness and F van Harmelen, editors. *OWL Web Ontology Language Overview*. Worldwide Web Consortium, February 2004. http://www.w3.org/TR/owl-features/.

[71] S McIlraith and T Son. Adapting Golog for composition of Semantic Web Services. In *Conference on Knowledge Representation and Reasoning*, 2002.

[72] SA McIlraith, TC Son, and H Zeng. Semantic Web Services. *IEEE Intelligent Systems*, **16**(2), pp. 46–53, March–April 2001.

[73] T Milo and S Zohar. Using schema matching to simplify heterogeneous data translation. In *24th Int'l Conference on Very Large Data Bases (VLDB)*, pp. 122–133. Morgan Kaufmann, 1998.

[74] P Mitra, G Wiederhold, and J Jannink. Semi-automatic integration of knowledge sources. In *2nd Int'l Conference on Information Fusion*, 1999.

[75] A Nadalin, C Kaler, R Monzillo, and P Hallam-Baker, editors. *Web Services Security: SOAP message security 1.1*. OASIS, February 2006. http://docs.oasis-open.org/wss/v1.1/.

[76] OMG Business Modeling and Integration DTF. *Business Process Modeling Notation (BPMN) 1.1*. OMG, January 2008. http://www.omg.org/spec/BPMN/1.1/.

[77] N Orlov, J Johnston, T Macura, C Wolkow, and IG Goldberg. Pattern recognition approaches to compute image similarities: Application to age-related morphological change. In *3rd IEEE Int'l Symp. on Biomedical Imaging*, pp. 1152–1155. IEEE, April 2006.

[78] AM Ouksel and A Sheth. Semantic interoperability in global information systems. *ACM SIGMOD Record*, **28**(1), pp. 5–12, 1999.

[79] L Palopoli, D Saccà, G Terracina, and D Ursino. A unified graph-based framework for deriving nominal interscheme properties, type conflicts and object cluster similarities. In *Int'l Conf. on Cooperative Information Systems*, pp. 34–45, September 1999.

[80] L Palopoli, D Saccà, and D Ursino. An automatic technique for detecting type conflicts in database schemes. In *Int'l Conf. on Information and Knowledge Management*, pp. 306–313, New York, 1998. ACM Press.

[81] L Palopoli, D Saccà, and D Ursino. Semi-automatic, semantic discovery of properties from database schemas. In *Database Engineering and Applications Symposium*, pp. 244–253, Washington, D.C., 1998. IEEE Computer Society Press.

[82] J Postel. *User Datagram Protocol*. RFC 768. Internet Engineering Task Force, August 1980. http://tools.ietf.org/html/rfc768.

[83] J Postel. *Internet Control Message Protocol: DARPA Internet Program Protocol Specification*. RFC 792. Internet Engineering Task Force, September 1981. http://tools.ietf.org/html/rfc792.

[84] J Postel, editor. *Internet Protocol: DARPA Internet Program Protocol Specification*. RFC 791. Internet Engineering Task Force, September 1981. http://tools.ietf.org/html/rfc791.

[85] J Postel, editor. *Transmission Control Protocol: DARPA Internet Program Protocol Specification*. RFC 793. Internet Engineering Task Force, September 1981. http://tools.ietf.org/html/rfc793.

[86] D Raggett, A Le Hors, and I Jacobs. *HTML 4.01 Specification*. Worldwide Web Consortium, December 1999. http://www.w3.org/TR/html.

[87] E Rahm and PA Bernstein. A survey of approaches to automatic schema matching. *The Very Large Data Base Journal*, **10**(4), pp. 334–350, December 2001.

[88] J Rao and X Su. A survey of automated Web Service composition methods. In *1st Int'l Workshop on Semantic Web Services and Web Process Composition*, *LNCS* **3387**, pp. 43–54, Berlin, 2005. Springer-Verlag.

[89] SA Renner. A "community of interest" approach to data interoperability. In *Federal Database Colloquium*, San Diego, August 2001.

[90] SA Renner, AS Rosenthal, and JG Scarano. Data interoperability: Standardization or mediation. In *IEEE Metadata Workshop*, Silver Springs, MD, April 1996. Poster presentation.

[91] L Richardson and S Ruby. *RESTful Web Services*. O'Reilly, first edition, May 2007.

[92] AW Roscoe. Model-checking CSP. In AW Roscoe, editor, *A classical mind: Essays in honour of C. A. R. Hoare*, pp. 353–378. Prentice-Hall, 1994.

[93] AW Roscoe. *The theory and practice of concurrency*. Prentice-Hall, 1998.

[94] AW Roscoe, PHB Gardiner, MH Goldsmith, JR Hulance, DM Jackson, and JB Scattergood. Hierarchical compression for model-checking CSP: How to check 10^{20} dining philosophers for deadlock. In *1st Int'l Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, *LNCS* **1019**, pp. 133–152, Berlin, 1995. Springer-Verlag.

[95] JB Scattergood. *The semantics and implementation of Machine-Readable CSP*. D.Phil. dissertation, Oxford University, 1998.

[96] DA Schiffman, D Dikovskaya, PL Appleton, IP Newton, DA Creager, C Allan, IS Näthke, and IG Goldberg. Open Microscopy Environment and FindSpots: Integrating image informatics with quantitative multidimensional image analysis. *BioTechniques*, **41**(2), pp. 199–208, August 2006.

[97] H Schuster, D Georgakopoulos, A Cichocki, and D Baker. Modeling and composing service-based and reference process-based multi-enterprise processes. In *12th Int'l Conference on Advanced Information Systems Engineering*, *LNCS* **1789**, pp. 247–263, Berlin, 2000. Springer-Verlag.

[98] A Sheth. Changing focus on interoperability in information systems: From system, syntax, structure to semantics. In MF Goodchild, MJ Egenhofer, R Fegeas, and CA Kottman, editors, *Interoperating Geographic Information Systems*. Kluwer Publishers, 1998.

[99] SSY Shim, VS Pendyala, M Sundaram, and JZ Gao. Business-to-business e-commerce frameworks. *Computer*, **33**(10), pp. 40–47, October 2000.

[100] D Skogan, R Grønmo, and I Solheim. Web Service composition in UML. In *8th IEEE Int'l Conf. on Enterprise Distributed Object Computing*, pp. 47–57. IEEE Computer Society Press, 2004.

[101] MK Smith, C Welty, and DL McGuinness, editors. *OWL Web Ontology Language Guide*. World-wide Web Consortium, February 2004. http://www.w3.org/TR/owl-guide/.

[102] JM Spivey. An introduction to Z and formal specification. *Software Engineering Journal*, **4**(1), pp. 40–50, January 1989.

[103] JM Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International, 1989.

[104] JR Swedlow, IG Goldberg, E Brauner, and PK Sorger. Informatics and quantitative analysis in biological imaging. *Science*, **300**(5616), pp. 100–102, April 2003.

[105] K Sycara, M Paolucci, A Ankolekar, and N Srinivasan. Automated discovery, interaction and composition of Semantic Web services. *Journal of Web Semantics*, **1**(1), pp. 27–46, December 2003.

[106] S Vinoski. Web Services interaction models, Part 2: Putting the "Web" into Web Services. *IEEE Internet Computing*, **6**(3), pp. 89–91, May 2002.

[107] C von Riegen, editor. *UDDI version 2.03 data structure reference*. OASIS, July 2002. http://uddi.org/pubs/DataStructure-V2.03-Published-20020719.htm.

[108] J Woodcock and J Davies. *Using Z: Specification, refinement, and proof*. Prentice Hall, 1996.

[109] L Zhang and S Malik. The quest for efficient Boolean satisfiability solvers. In *14th Int'l Conference on Computer Aided Verification (CAV)*, *LNCS* **2404**, pp. 641–653, Berlin, August 2002. Springer-Verlag.