UNIVERSITY OF SOUTHAMPTON

Programming Languages and Principles for Read–Write Linked Data

by

Ross J. Horne

A thesis submitted in partial fulfillment for the
degree of Doctor of Philosophy

in the

Faculty of Physical and Applied Sciences
School of Electronics and Computer Science

November 2011

ABSTRACT

This work addresses a gap in the foundations of computer science. In particular, only a limited number of models address design decisions in modern Web architectures. The development of the modern Web architecture tends to be guided by the intuition of engineers. The intuition of an engineer is probably more powerful than any model; however, models are important tools to aid principled design decisions. No model is sufficiently strong to provide absolute certainty of correctness; however, an architecture accompanied by a model is stronger than an architecture accompanied solely by intuition lead by the personal, hence subjective, subliminal ego.

The Web of Data describes an architecture characterised by key W3C standards. Key standards include a semi-structured data format, entailment mechanism and query language. Recently, prominent figures have drawn attention to the necessity of update languages for the Web of Data, coining the notion of Read–Write Linked Data [22]. A dynamic Web of Data with updates is a more realistic reflection of the Web.

An established and versatile approach to modelling dynamic languages is to define an operational semantics. This work provides such an operational semantics for a Read–Write Linked Data architecture. Furthermore, the model is sufficiently general to capture the established standards, including queries and entailments. Each feature is relative easily modelled in isolation; however a model which checks that the key standards socialise is a greater challenge to which operational semantics are suited. The model validates most features of the standards while raising some serious questions.

Further to evaluating W3C standards, the operational semantics provides a foundation for static analysis. One approach is to derive an algebra for the model. The algebra is proven to be sound with respect to the operational semantics. Soundness ensures that the algebraic rules preserve operational behaviour. If the algebra establishes that two updates are equivalent, then they have the same operational capabilities. This is useful for optimisation, since the real cost of executing the updates may differ, despite their equivalent expressive powers. A notion of operational refinement is discussed, which allows a non-deterministic update to be refined to a more deterministic update.

Another approach to the static analysis of Read–Write Linked Data is through a type system. The simplest type system for this application simply checks that well understood terms which appear in the semi-structured data, such as numbers and strings of characters, are used correctly. Static analysis then verifies that basic runtime errors in a well typed program do not occur. Type systems for URIs are also investigated, inspired by W3C standards. Type systems for URIs are controversial, since URIs have no internal structure thus have no obvious non-trivial types. Thus a flexible type system which accommodates several approaches to typing URIs is proposed.

Contents

Nomenclature

$\otimes$ or `JOIN`	The tensor product or the join keyword
$\oplus$ or `CHOOSE`	The additive disjunction, external choice or the choose keyword
$\bigvee$ or `SELECT`	The additive existential quantification or the select keyword
$\bigwedge$ or `LOCAL`	The additive universal quantification or the blank node quantifier
$*$ or `DO`	Iteration or the do keyword
$\parr$ or `,`	The multiplicative disjunction or parallel composition
$\bot$ or `NOTHING`	The multiplicative zero or the empty process
$\mathrm{I} \oplus U$ or `OPTIONAL` U	The optional keyword and its encoding
$(.)^\bot$ or `DELETE`	Linear negation or delete
(.) or `INSERT`	Insert
`FILTER`	Explicit embedding of Boolean algebras
I or `true`	The multiplicative unit or the Boolean value true
0 or `false`	The additive zero or the Boolean value false
$\wedge$ or `&&`	Classical conjunction
$\vee$ or `||`	Classical disjunction
$\neg$ or `!`	Classical negation
$\mathcal{G}_a$	The named graph modality
`p`	The predicate type constructor
$\top$	The resource type
$\cup$	The union type constructor
#	The container type constructor
$\models$	The satisfiability relation for Boolean algebras
$\rhd$	The atomic commitment relation
$\xrightarrow{l}$	The labelled transition
$\equiv$	Structural congruence
$\sim$	Bisimilarity
$\simeq$	Contextual equivalence
$\vdash$	The typing relation
$\Vdash$	The algorithmic typing relation
$\leq$	The subtype relation and the refinement relation
$\preceq$	Simulation
$\sqsubseteq$	The preorder over names

Acknowledgements

I have been lucky to discuss ideas with many stimulating people. I have tried to accommodate the perspectives of as many of these people as possible. I would like to acknowledge the following people in particular.

Thanks to Vladimiro Sassone for his supervision and broad perspectives. His direction has helped me focus on research that matters. Thanks to the examiners of this thesis, Alessio Guglielmi and Corina Cîrstea, for challenging my basic assumptions.

Thanks to John Colley for discussions on a tolerant philosophy for computer scientists. Thanks to Hugh Glaser for introducing me to Linked Data and steering me clear of ontologies. Thanks to all other members of the Dependable Systems and Software Engineering group for some memorable years.

In loving memory of Claire Horne.

Chapter 1

Introduction

The real goals of this work are human: to mediate between people who do not usually collaborate. To set the tone for this challenge, three themes are discussed. The first theme is the motivational issue of the broad setting of this work. Clarifying the setting emphasises the pressing need for the investigation which is embarked upon. The second theme deals with expectation. The balanced nature of this work means that a completely fulfilling subjective or objective truth will not be achieved, since such goals are fundamentally at odds with each other. The third theme deals with misunderstandings due to language. Language problems occur when similar words are used by different people in different contexts, and are exasperated when both people claim expertise. Thus the form of the subsequent chapters should be less of a surprise.

1.1 Architectures for Every-day Applications

This work was first inspired by some recent advances which are now evident in most daily lives. The advances were enabled by the adoption of a Model-View-Controller architecture for Web applications.

The View in this architectural style provides the user interface for an application. The key advance to enable the View was Ajax, which allows messages to be passed asynchronously between the client and the server [52]. This simple extension demonstrated that most every-day user interfaces could be ported to the Web.

The Model in the Model-View-Controller architecture consists of some semi-structured data which represents the content of an application. The data is delivered by a protocol, which allows the content to be read and sometimes updated. The Model uses a standardised format, so that content can be shared across multiple applications. In this architecture, the Model and the View are independent.

The Controller is a program which implements an application by coordinating the Model and the View to fulfil the requirements of the application. The Model-View-Controller architecture has been widely adopted in industry with minimal input from computer science.

In computer science, exactly one generation has passed. Consider just two great figures from the first generation: John Backus (1924–2007) and Peter Landin (1930–2009). Each introduced a corner stone of computer science. The insight of Backus was to adapt the approach of Chomsky, from the syntax of natural languages to the syntax of programming languages [10]. The insight of Landin was adapt approach of Church, from the foundations of mathematics to the foundations of programming languages [86].

The maturity of these established approaches to programming languages is evident in the modern tools. When a grammar is specified, as would be done on a piece of paper, the parser is generated. Similarly, an accurate specification of a transformation matches the corresponding functional program, which then works. These tools are extremely effective only because they are based on elegant well understood models.

Recent advances such as the Model-View-Controller architecture present fresh problems, which challenge traditional models of computing. The recent advances in the decoupling of the View from the Model, along with with advances in commodity portable hardware, demonstrate that isolated desktop computing was an era of little more than a couple of decades.

Concurrency was of course a problem in desktop computing, where it is was sufficient to treat concurrent processes by interleaving sequences of their actions. However, the problems of concurrency are now more subtle. Applications can now be delivered by distributed server farms. Each server farm consists of many machines. Each machine has many processors. This set up does not fit the old interleaving model of concurrency. The interleaving model relies on there being one place where one observer witnesses everything happening sequentially. Where would an observer stand in the set up of modern computing?

It is clear that an understanding of true concurrency is required. For readers who claim that true concurrency is understood, they are invited to demonstrate tools that match the parser generator or functional programming compiler exemplified above. The absence of these tools is the first indication of the gap to fill.

1.2 Eternal Human Challenges

> There are still people saying that in order to make computer science one essentially needs a soldering iron.
>
> Jean-Yves Girard 1987 [55]

The gap to fill in modern computing is partly human. For each of the advances highlighted in the Model-View-Controller architecture there is the human problem of agreeing standards, which is

a human process with no definitive answers. There are however more basic human challenges in computer science itself.

It appears that there are two polarised views on the role of computer science when on the Web. This first says that the Web is 'practical' and should be tackled by the metaphorical soldering iron. The second says that the Web is abound with buzz and void of substantial 'theory', evoking again the metaphorical soldering iron as the most sophisticated tool. Should a reader approach this work wondering whether this is 'theoretical' or 'practical' work, then the answer is to ask a better question. Each terms suggests a disregard of the subjective truth or objective truth, respectively.

Subjective truth is based on sense experiences of what someone perceives to be reality. Objective truth is based on what someone perceives to be an aesthetic model. In this work we appeal philosophically to both subjective and objective truths, preferring neither. Philosophy is as old as the written word and inseparable from science; thereby we benefit from the maturity of thousands of years of human thinking. Indeed the interplay between subjective and objective truth are is characterised by the Aristotelian and Platonic views respectively. Thus such notions are at the foundation of Western science. Famous counterpoints of the past remain relevant to the problems of today.

1.2.1 Towards an objective model of a subjective problem

The development of a model can be simultaneously approached from two directions. The first direction is to take subjective observations and attempt to construct an objective model which captures the subjective observation. The other approach is to take an objective model and attempt to fit subjective observations to the model. Both approaches are rarely conclusive, thus modelling problems combine both approaches.

In this work we begin with the subject. The initial subject matter encompasses all Web standards which aim to contribute to a Web of Data. The initial question is what standards, and what aspects of the standards, are "in use." A lot of features of standards are either rarely used, are of secondary value so used in only a few applications, or are unfit for their intended purposes. A lot of features can therefore be immediately ignored. However this approach depends mainly on the opinions of people, which are based on their personal experiences.

Having subjectively selected key standards and features, the next thing is construct an ad-hoc model of the standards. Each standard is relatively easy to model in isolation but the models are often most easily expressed using different modelling frameworks. This results in several informally connected models. A framework must be found in which the standards which work directly together can be expressed. Some features socialise as intended while others do not. Thus further features of standards can be constrained or entirely ignored. This attaches a weak objective justification to design decisions.

A model exposes glimpses of the objective nature of standards. Parts of logical systems with familiar rules appear, where they were not expected. But if a full shift is made to any such objective model, then the subject matter is immediately blurred. Some features are lost and some new features appear which would be too surprising to present subjectively. Thus a full objective truth, or aesthetic external reality, is never expected to be discovered.

It is difficult to reject outright the existence of an objective truth, even if it cannot be found in any conclusive sense. The objective truth is always there as a tantalising guide. Perhaps the most promising glimpses that are exposed by this work are in algebraic properties, since similar algebraic properties have arisen in searches for objective models of nature, through physics and linguistics.

But the models presented stop short of exploiting these potential tantalising links. Instead the focus is returned to the subject matter where opinions still dominate. But this will never change as even Poincaré experienced when he stepped out of his objective reality to consider a subjective source.

> But it is to the opposite side — the side of nature — against which we must direct the main corps of our army. There we meet the physicist or the engineer who says to us: "Can you integrate for me such a differential equation? I must have it within eight days because of a certain construction which must be finished by that time." "That equation," we reply, "is not of an integrable type; you know there are many like it." "Yes, I know that; but of what use are you then?"
>
> Henri Poincaré (1908) [113]

If even Poincaré experienced hostility to crossing between the object and the subject, then what hope does anyone else have of easing tensions? Perhaps this explains why polarised communities that rarely reach out to each other exist. Any attempt in either direction, to match a subject with an object, or vice versa, tends to dilute one aspect and falls short of expectations. But such human challenges should not halt all communication between communities. Surely the most interesting problems lie where opinions clash.

1.3 The Language Game

> I shall also call the whole, consisting of language and the activities into which it is woven, a "language game."
>
> L. Wittgenstein 1945 [133]

Language itself presents significant challenges. This is not only a reference to programming languages, but also to the natural language in which computer scientists communicate. Challenges

posed by language are tackled in this work by considering the work of communities whose mutual interests are clouded by misunderstandings. Closed communities tend to establish their own language to refer to the concepts they experience.

This work is particularly concerned with Web standards. A standard is not a physical product. It is a document which proposes a standard language to tackle a problem. The language in a standard is created by a community of people with different perspectives on the problem. There is rarely a canonical answer to what the language of a specification should be, so there always remains scope for misunderstandings in the language used.

1.3.1 Types are not types

A significant misunderstanding in the standards concerns the word 'type.' The origin of the problem appears to be in an early version of a Web standard originating from research at Nokia [87]. In this first standardised version of the Resource Description Framework, the type predicate is used to connect a resource to a class. A resource is anything being described; while the class is part of the description of that resource. The original document does not venture much further in defining a type in this context.

The idea of a resource being typed corresponds to many established ideas. Historically, types were introduced to avoid paradoxes in the foundation of mathematics. A mathematical entity cannot just exist in the universe. The universe is too large to discuss. However, given a world with boundaries it becomes more reasonable to discuss the existence of an entity. A world in which an entity exists is a type for the entity.

None of the potential models for types are suggested in the specification of RDF. The result is that different communities have interpreted types in different ways. The slogan which cannot be emphasised enough is the following irreflexive statement: types in RDF are not necessarily types.

1.3.2 Semantics are not semantics

Another killer misunderstanding in the community embroils the word semantics. Misunderstandings surrounding the word semantics are much more severe than those surrounding types. With types there are different models for different scenarios. For some models of types, sets of all entities of a given type can be constructed. For other models types are treated algebraically. However, the idea behind types of controlling the world in which an entity exists is consistent. In contrast, there is no consistent theme to semantics.

Misunderstandings surrounding the word semantics are so severe that the key project has changed its name from the Semantic Web. The Semantic Web project was introduced in an article which enthusiastically describes a future where data is available on the Web [24]. Machines would

know the meaning of data, so would use the data to perform basic tasks making our daily lives easier! The semantics were the meaning of the data which machines would understand.

The Semantic Web project attempted to fix one notion of semantics into which everything can be interpreted. However, even for long established languages, a definitive semantics cannot be fixed. There is a vast volume and diversity of research behind the semantics of languages. Nowhere has a unified framework for semantics ever been established.

The Semantic Web project was revisited several years later by which time the initial proposal was clearly experiencing difficulties [128]. The difficulties experienced included issues associated with the interpretation of semantics. The notion of semantics adopted was that of an ontology, where even the word ontology was not used in its traditional philosophical sense. This resulted in an emphasis in producing 'deep ontologies' which enforce heavy constraints on structures.

The project review suggested that 'shallow ontologies' should be used to achieve the intended levels of scalability. This is the first step towards a shift in emphasis from semantics to data. This change of emphasis was clarified by a sensible change in the name of the project. The project is currently referred to as the Web of Data or the Web of Linked Data to emphasis the rôle of URIs for establishing links [28].

By using the word 'data' the misunderstandings associated with the word 'semantics' are avoided. The following irreflexive statement sums up the issue that the original notion of semantics excludes clearer notions of semantics: semantics in the Semantic Web are not necessarily semantics.

1.3.3 Syntax is syntax

Profound and unavoidable misunderstandings arise in the syntax of any language used to notate the subject matter. To effectively discuss a model at some point a syntax must be introduced. A syntax may be chosen to highlight some aspect of the framework. One syntax may emphasise communication and another may emphasise data.

A syntax may also be chosen to ease the understanding of an idea for a particular community. The language of the foundations of mathematics is quite different from the language of compiler design. However there is a vast overlap between the subject areas. A compiler writer may prefer an ASCII syntax that can be entered into a text editor with recognisable key words. The mathematician may prefer a concise syntax that can be easily manipulated on a sheet of paper with recognisable symbols. A programmer may prefer a sugared syntax which encodes common tasks in a familiar style. No single syntax presents a universal solution. However all syntaxes which play a role in some activity are valid, hence the following statement is reflexive: a choice of syntax is just a choice of syntax.

The variations in syntax for different emphasis and different communities was highlighted in the later work of Wittgenstein. First Wittgenstein rejected the idea that there is an external semantics

associated with language. Instead he emphasises the importance of the language game. The meaning of language is not fixed; it varies with the activity in which the language is used.

Data on the Web can engage in a wide range of activities at different levels of abstraction. Thus a single language cannot be found to communicate all activities. Despite this Web standards attempt to fix a language for global activities. Ultimately standards will always be succeeded by new standards, as activities evolve. For machines however something has to be fixed for the data to be understood, so agreeing standards for data exchange is not futile. Furthermore, to be understood by one community a choice of syntax may be made; while, to convey the intention of a standard to another community a different syntax may be chosen.

1.4 Tensions to be Expected

This work endeavours to play the language games of a number of communities. Each of these communities have a significant rôle in the development of mechanisms for the Web of Data. One community is addressed in Chapter 2; another community is addressed in Chapter 3. Both of these chapters are primarily concerned with subjective issues which are illustrated through examples. A weak objective justification is provided by demonstrating the existence of a concise operational semantics. Chapters 4 and Chapter 5 make steps towards a stronger objective justification for the standards, by investigating the strength of their correspondence with algebra and type systems respectively. The algebra and type systems retain a strong subjective interpretation, or purpose, in the application domain.

Chapter 2

Read–Write Linked Data Standards

This section begins by providing an overview of key ideas behind Linked Data. It provides a discussion of the semi-structured data format, query language and reasoning mechanisms which have been standardised by the W3C. The discussion makes use of examples in the most prevalent syntaxes for these standards, so should be easily understood by users of these standards. The discussion justifies why the standards have been chosen, whilst highlighting issues with design decisions.

The first formal rules are introduced to describe an update language for Linked Data. Such an update language is a requirement for enabling Read-Write Linked Data, as heralded by Tim Berners-Lee. The update language is defined using a ASCII syntax with curly brackets, so that it is similar to the syntax of common engineering languages. The language is then defined using simple logical rules, explained using clear simple examples. The rules of the language are presented without any meta-syntax or theory; only the concise syntax of the language and English are used. The point made is that there is nothing complicated or obscure about this work.

2.1 The Setting of Key Web Standards

> There is an architecture in which a few existing or Web protocols are gathered together with some glue to make a world wide system in which applications (desktop or Web Application) can work on top of a layer of commodity read-write storage. The result is that storage becomes a commodity, independent of the application running on it.
>
> Tim Berners-Lee 2010 [22]

The model presented is a contribution to understanding the principles of the architecture of modern Web applications, which has changed significantly due to recent developments in infrastructure. Web applications can now deliver user interfaces comparable to many traditional

applications. Consequently, mainstream application engineering and Web application engineering are increasingly interlinked. User interface concerns can be isolated in the View of an application. The problem of moving a View onto the Web was solved by presenting the View using Web standards. Web standards are a product of transparent negotiations between industry and standards bodies.

The View is one component in the Model-View-Controller architecture, which is widely adopted for application development. Another component, the Model, provides data which forms the subject of the application. The Controller coordinates interactions with the Model to achieve some objective. Having successfully moved the View onto the Web, standards bodies are tackling the problem of moving the Model onto the Web. The common motive for data standards is that moving the Model onto the Web allows common subject matter to be shared between applications. Evidence of the potential of sharing data on the Web is the ubiquity of feeds, e.g., RSS and Atom [125]. Feeds are now a primary technology used to deliver data on demand between news sources and consumers.

Making data available on the Web gives the potential for data to link across traditional boundaries. This is enabled by using the URI as a standardised naming system for identifiers in data. By naming the identifier of a resource with a URI the resource can be referred to from any other location. Efforts to exploit these links between data sources have resulted in several proposed standards. The common aim of proposed standards is often referred to as establishing a Web of Data [27]. Data which exploits the link structure of the Web is distinguished by the term Linked Data [21]. The Linked Data initiative is supported by W3C recommendations and working drafts, which reflect a consensus on the aims of the initiative [78, 33, 115]. This work draws from key standards for Linked Data and presents an executable model in which the standards coexist.

At a low level, Linked Data is delivered as messages in a semi-structured data format. The Resource Description Format (RDF) is the leading standardised semi-structured data format for Linked Data [78]. At this level, an HTTP request to a URI produces some RDF which describes the resource represented by the URI. No requirements are enforced on how the RDF is produced or what is done with the RDF. Message passing on channels is modelled by many process calculi [98, 31, 1, 37].

At a higher level, Linked Data can be gathered in stores which are accessed using queries. A store responds to queries as prescribed by the SPARQL Query standard [126]. Rich data sources are now published as stores, notably the UK Government Data and DBpedia [28, 80]. These examples gather data, from UK Government Databases and Wikipedia respectively, then prepare the data for queries. No requirements are placed on the method of preparation. SPARQL Query has been modelled as a graph query language and using relational algebra [110, 43].

The executable model presented here is tailored to problems introduced by an update mechanism. Challenges associated with updates are highlighted by Tim Berners-Lee in a note on Read-Write Linked Data. Updates are considered at several levels of granularity. At a coarse

granularity of update the contents of a store are replaced periodically. Periodic updates are adequate when data changes infrequently. An intermediate granularity is achieved by dividing a store into regions, where each region is updated independently. This idea is captured by named graphs for RDF [38]. A protocol for updating named graphs is under development [105]. Feeds and standardised protocols for feeds also work at a similar level of granularity [104, 59].

The primary challenge is to model fine grained updates at the level of triples. Triples are the basic components of RDF which resemble simple sentences in natural language of the form subject–verb–object. Fine grained updates account for exactly the triples required to perform an update. Updates which use disjoint triples may occur concurrently. By using minimal resources, an update causes minimal interruption to a store. This approach avoids regions, which are difficult to design when the long term behaviour cannot be predicted. Fine grained updates are known to present conceptual difficulties, as highlighted by Reynolds in the traditional setting of shared memory [118]. The model is a contribution to the understanding of fined grained updates for Linked Data.

Implementing Read-Write Linked Data is necessary for using Linked Data in modern applications. For instance, in wikis or social media users increasingly write data. In contrast, existing Linked Data applications tend to be limited to reading data. Furthermore, without an update mechanism for the Model, the Model and the Controller in the modern application architecture cannot be decoupled. A Controller instead requires lower level access to the Model to perform updates. This work therefore supports efforts towards a standardised approach to Read–Write Linked Data [53].

2.2 The Suite of W3C Standards

The W3C has introduced many standards to address a range of applications which are delivered over the Web. The standards introduced here are particularly relevant for Linked Data. This section presents examples in the standard formats for the semi-structured data format RDF, the RDFS vocabulary description language, the SPARQL Query language and the OWL ontology description language.

There are many design issues associated with the W3C standards. None of the standards claim to be canonical. Some of the features of the standards are widely accepted to be a good idea and are widely adopted. Some features have technical issues which gives rise to conflicting interpretations and small deviations from the published recommendations. Some features have barely been adopted and rarely appear in implementations. The main purpose of this section is to highlight which features of the languages are core, which features are secondary and which features remain controversial.

Note that the following namespaces abbreviate URIs for readability. *person: eprint: soton: rdf: rdfs: owl: foaf: dc: dc11: xsd: res: postcode: vcard: eg:* .

Most examples in this section are in the Turtle syntax for RDF [16]. The Turtle syntax is a simple format for RDF triple which is designed to be clearly readable. Turtle is the prevalant syntax for RDF, rather than the standardised XML syntax for RDF which has fallen out of favour. The XML syntax is difficult to read and has no XML Schema, which means that few of the benefits of using XML can be exploited.

2.2.1 Overview of the Resource Description Framework

The Web of Linked Data is concerned with resources identified by URIs. The relationship between URIs are indicated using RDF, a standardised loosely structured data format. RDF extends the traditional links of the Web, which can be seen as pairs of URIs, to triples of URIs. Such triples of URIs consist of a subject, predicate and object, where the predicate indicates how the subject and object are related. Triples are built from URIs, literals and blank nodes.

The following RDF triple indicates that one person knows another person. The people are identified by URIs. The predicate which relates the two people is indicated by a URI from a common metadata vocabulary. This example is expressed in the Turtle format for RDF [16], triples are terminated by full stops.

person:9724 *foaf:knows* *person:10511* .

Further to indicating relationships between URIs, triples can also represent relations between URIs and literals. The object of a triple can be a literal. A literal is some basic data, such as a string of characters or a date. In most RDF formats the type of the literal is indicated along with its representation as a string. The following example indicates the date of birth of the subject, where the predicate is drawn from the popular Friend of a Friend (FOAF) vocabulary. The type of the literal is indicated to distinguish the date from a plain string. The range of types for literals is borrowed from the XML Schema Datatypes standard [25].

person:10511 *foaf:birthday* `"1983-06-05"^^`*xsd:date* .

To allow further structure to be encoded in RDF, triples may include blank nodes. Blank nodes are identifiers which are not URIs, but can be used in place of URIs. A blank node is indicated by the prefix `_:` followed by an identifier. A blank node can appear as the subject or the object of a triple. Although, to simplify definitions, this work allows blank nodes to also to appear as a predicate. For instance, the following example represents an address using a blank node.

person:10511 *eg:address* *_:a* .
_:a *eg:city* *res:Southampton* .
_:a *eg:postcode* *postcode:SO171BJ* .

The main difference between a URI and a blank node is that the URI is global whereas the blank node is local. If the same blank node appears in two different datasets, then the two blank nodes are different. Thus one dataset cannot refer directly to blank nodes in another dataset. Blank nodes can be renamed without changing their meaning, which allows datasets to be merged without clashes of blank nodes.

2.2.1.1 Problematic features of RDF.

A controversial feature is reification. Reification allows triples to be described using triples. For instance, the first example in this section can be reified as follows.

_:triple1 rdf:type rdf:Statement .
_:triple1 rdf:subject person:9724 .
_:triple1 rdf:predicate foaf:knows .
_:triple1 rdf:object person:10511 .

Reification introduces a URI for a triple, which allows the triple itself to be described using RDF. This can be useful for assigning properties to triples describing their provenance, or access policy. Reification is however confusing as the triples and the reified triples must be considered separately, adding conceptual difficulty. Reification has been superseded by named graphs which provided named locations for triples, so provenance information and access policies can still be discussed collectively at a coarser more manageable level of granularity [38]. Named graphs are discussed in Sec. 3.3.

The RDF standard introduces a number of data structures for organising resources. In particular list and three types of containers are introduced — bags for unordered collections, sequences for ordered collection and alternatives for collection of resources where only one resource can be chosen. For instance, the following is a representation of a list of two resources in RDF.

_:a rdf:type rdf:List .
_:a rdf:first person:10511 .
_:a rdf:rest _:b .
_:b rdf:type rdf:List .
_:b rdf:first person:9724 .
_:b rdf:rest rdf:nil .

Unfortunately RDF lists are not really lists. There can be multiple heads, tails, cycles and incomplete information, which are forbidden in conventional lists. Furthermore, the type *rdf:List* does not support polymorphism. It is conventional in typed list processing languages to have polymorphic lists, where the parameter indicates the type of data to be found in the list. Similar issues apply to containers.

Steps have been taken to tackle this problem in N3 Logic [23], where constraints are imposed on the structure of lists, using first order logic. Also, lists are primitive in the Turtle RDF format [16], which offers a more conventional solution. It is not clear that lists and containers are widely adopted, except in technical encodings. For many scenarios where containers might be used named graphs can be used. Named graphs introduce clear explicit primitives for organising triples [38].

2.2.2 RDF types and schema

A key feature of the core vocabulary of RDF is the predicate *rdf:type* . The meaning of *rdf:type* is elaborated by the RDF Schema (RDFS) standard [33]. RDFS introduces extra vocabulary for describing the relationships between classes and predicates along with inference rules. The roles of RDF Schema and XML Schema should not be confused. XML Schema is clearly a type system for XML thus constrains the shape of XML data [35]; whereas RDF Schema is used to infer new information.

2.2.2.1 The vocabulary for classes.

Classes are URIs which can be used as the object of the predicate *rdf:type* . Classes are guides for how a URI is intended to be used. Since URIs have no internal structure, RDFS classes are unlike conventional types for data such as XML Schema datatypes [25]. The triple below indicates that the subject is an instance of a the class *foaf:Person* .

person:10511 *rdf:type* *foaf:Person* .

Classes are just URIs which means that triples can be used to describe classes. Thus a class can be described using triples, just like any other resource. The triples associated with a class can guide how the predicate is used, as follows.

foaf:Person *rdfs:comment* `"The class of people."` .

Classes can be structured using the *rdfs:subClassOf* predicate. The relation indicates that any instance of the first class is an instance of the second class. For instance, in the FOAF vocabulary *foaf:Person* the class of people, while in the Dublin Core vocabulary *dc:Agent* is the class of “things that can act.” The Dublin Core vocabulary provides a description, “Examples of Agent include person, organization, and software agent.” So the following sub-class relationship between the two vocabularies can be assumed.

foaf:Person *rdfs:subClassOf* *dc:Agent* .

The sub-class relation is a preorder: it is transitive and reflexive. The specification does not determine whether the relation is irreflexive ($a \leq b$ and $b \leq a$ yields $a = b$), since no explicit equality predicate for classes is provided in the RDFS specification. For instance, do the following triples mean that the classes *foaf:Person* and *dbp:Person* are equal?

foaf:Person rdfs:subClassOf dbp:Person .
dbp:Human rdfs:subClassOf foaf:Person .

The answer to this question depends on the application and choice of model for RDFS.

2.2.2.2 The vocabulary for predicates.

The RDFS standard provides vocabulary to describe URIs which are used as predicates. Inference rules specify how the vocabulary for predicates are interpreted. Two features are enabled. Firstly, predicates can be ordered using the *rdfs:subPropertyOf* predicate. The ordering of predicates is important for interoperability of vocabularies. Secondly, rules are provided to indicate the domain and range of a property. The domain and range of a predicate are intended to infer incomplete type information.

The sub-property predicate is similar to the sub-class predicate. It states that the subject of the predicate is stronger than the object of the predicate. For instance, the following triple indicates that the predicate *eg:colleague* is stronger than the predicate *foaf:knows* .

eg:collegue rdfs:subPropertyOf foaf:knows .

So, suppose that *eg:colleague* appears in a triple, as follows.

person:10511 eg:colleague person:9724 .

Then the sub-property triple can be used to infer that the following triple holds. Which is weaker than the triple above.

person:10511 foaf:knows person:9724 .

The sub-property predicate defines is a preorder over URIs, since it is reflexive and transitive. As with the sub-class predicate, the sub-property predicate does not necessarily form a partial order. This relation will be explicitly formalised as a preorder in this work. Sub-properties are useful for structuring predicates in vocabularies, and are particularly useful for the integration of vocabularies which use different URIs for similar purposes.

The domain and range of a predicate can be indicated using *rdfs:domain* and *rdfs:range* . The *rdfs:domain* predicate indicates the class of URIs which may appear as the subject of a predicate in a triple. Similarly, the *rdfs:range* predicate indicates the class of URIs which can be used as an object. The range can also indicate a datatype when a literal is used as the subject of a predicate. For instance, the triples below indicates that *foaf:knows* is predicate which relates

one *foaf:Person* to another *foaf:Person* .

foaf:knows rdfs:domain foaf:Person .
foaf:knows rdfs:range foaf:Person .

The standardised inference rules for *rdfs:domain* and *rdfs:range* are unconventional. The rules allow the types of URIs to be inferred when they are used in a triple where the domain or range of the predicate of the triple is prescribed. For instance, consider the following triple in the presence of the triples above.

person:10511 foaf:knows person:9724 .

It can be inferred that the following triples hold.

person:10511 rdf:type foaf:Person .
person:9724 rdf:type foaf:Person .

A more conventional type system would work by checking that the domain and the range of the predicate matches the types of the subject and object. In such a type system the above rules would be type inference rules, which infer incomplete type information. However such a type system is completely missing from the specification. In Chapter 5 a candidate type system is suggested, but mismatches cannot be avoided. Elsewhere in this work the rules for the domain and range of predicates are ignored. Thus the domain and range predicates are secondary concerns, whereas sub-properties are essential for interoperability of vocabularies.

2.2.2.3 Top level classes.

RDFS introduces several top level classes notably, *rdfs:Resource* , *rdf:Property* and *rdfs:Class* . Each top level class corresponds to the main roles of a URI or blank node. The class *rdfs:Resource* is the very top level class that contains all URIs, thus all URIs are resources. The classes *rdf:Property* and *rdfs:Class* also range over URIs, so are sub-classes of *rdfs:Resource* . The class *rdf:Property* is class of URIs which are used in the predicate position of a triple. The class of classes, *rdfs:Class* is the class of URIs which appear as the object of the predicate *rdf:type* .

The choice of top level types is controversial. Since *rdfs:Class* is a class, it is of type class. Also, *rdfs:Resource* is a class so is of type class *rdfs:Class* and *rdfs:Class* is a sub class of *rdfs:Resource* . Hence, *rdfs:Resource* is of type *rdfs:Resource* . By interpreting classes as sets of URIs $a \in a$ must hold, but this violates the axiom of foundation in set theory. Therefore classical model theory can never fully capture RDF Schema. Also in type theory, if the top element is a type of itself then problems such as the Burali-Forti paradox arise. Such a paradox was demonstrated, by Girard, to exist in an early formulation of intuitionistic type theory [94],

which also featured nested levels of types. Thus a model of RDF Schema must take some care and liberty when the standard rules of RDFS are interpreted.

Conceptual problems with the standard semantics follow from the lack of distinction between types and terms, which results in an infinite nesting of layers. This is conceptually difficult for both the user and for models to capture, so is considered to be an oversight. This oversight originates in early RDFS working drafts which draw analogies between RDFS and classes in Java. Java classes suffer from similar issues, by mixing concepts conventionally distinguished as terms, types and classes, which makes the full language immune to formal models and difficult to clearly conceptualise [75].

2.2.2.4 Three manageable sub-systems of RDFS.

The RDFS standard is not conclusive. It contains some essential ideas, but a few unhelpful design decisions were made. The problems introduced by the design decisions can be cleaned up in various contexts. Here three different options for clarifying RDFS, with little loss of functionality are explained. The first, due to Pan and Horrocks, is to constrain RDF so that it fits a Tarski-style model theory. The second is to to constrain RDF so that it fits a type system, as pursued in Chapter 5. The third is to forget all but the essential features and treat RDFS as a simple preorder over URIs, as described in Chapter 3.

A Tarski style model theory for RDFS can be provided, which provides an interpretation of features of RDFS in set theory [108]. However, to obtain the model theory the standardised rules of RDFS must be modified. Pan and Horrocks introduce four layers, one for instances, one for classes and other for meta-classes and a fourth top 'meta-meta-class' containing everything. Classes are then mapped to sets of instances, and meta-classes are mapped to sets of classes. Predicates are mapped to pairs of sets of instances. This approach to RDFS works, but the model theory does not easily extend to models of programming languages which use RDFS, such as those considered in this work.

Another approach is to provide a type system which agrees with RDFS. A type system requires the type information, which is indicated by *rdf:type* , *rdfs:domain* and *rdfs:range* , to be separated from other triples. A type system also means that the top level types must be treated differently. The advantage of using a type system is that the sub-class relation can be recovered by a subtype system. Also, the domain and range properties can be recovered using type inference.

The type theoretic approach can be tackled in at least two ways. The more complex but comprehensive approach would be to introduce a higher-order type system which allows nesting of layers of types. Higher-order type systems are excessively complex for this relatively simple application and cannot be expected to be understood by most users [49]. Another approach is to use a simple type system. A simple type system provides a more natural explanation for the concepts in RDFS. Such a simple type system is the basis of Chapter 5.

A much simpler approach is to first ask some fundamental questions. What features of RDFS are really required? Is it useful to be able to infer that any URI is of type *rdfs:Resource* ? Is it helpful when a user goes to a data set asks for the subjects and objects of the predicate *rdf:type* and receives hundreds of results asserting that each URI in the dataset is a resource? Almost certainly not. Also are the domain and range inferences essential? These inferences appear to contribute no more than any other ad-hoc choice of inference mechanism to the usability of Linked Data.

Thus almost all features of RDFS can be ignored. The key features that remain are the sub-class and sub-property relations. These are essential for integrating datasets which use different URIs for predicates and classes which are related. Thus all that is left is two relations which are both preorders over URIs.

The suggestion is that the final ultra light weight approach using preorders over URIs is easy to understand, easy to work with and is all that is required for many applications. Furthermore, the simply typed approach and the preorder approach can coexist without much difficulty. Also, the model theoretic approach can be used when RDFS is used in isolation [107]. Thus a manageable model for RDFS can be chosen depending on the application.

2.2.3 Deep ontologies

The Web Ontology Language (OWL) is intended as a fundamental technology for the Semantic Web. The aims of OWL are to provide a rich but tractable ontology for describing relationships between concepts. Ontologies are often used to place constraints on the shape of data. Ontologies are useful in complex applications such as medical knowledge bases, where an ontology can detect inconsistencies between the data and the ontology [116].

There is a large body of work, which has supported the development of ontologies for the Web. Much of the work is at the convergence of ad-hoc knowledge representation and modal logics, which has resulted in Description Logics. Description Logics are tractable logics over relations, which are generally accompanied by Tarski-style model theory [74]. More recently Description Logics have been provided with a co-algebraic semantics, which is a natural approach to modal logics [58].

A considerable amount of expertise is required to build a rich ontology [128]. As a result, Description Logics have not played such a prominent role in the more recent Linked Data movement, which aims to produce tools for Web developers rather than scientists. Instead a few features of Description Logics which are easy to understand have been applied intuitively.

The most prevalent feature of OWL used by the Linked Data community is the *owl:sameAs* predicate [45, 57]. The *owl:sameAs* predicate is used to indicate that two URIs represent the same resources. However, subtleties mean that *owl:sameAs* is often not used according to the OWL specification. Investigations into the use of *owl:sameAs* in published datasets have

concluded that *owl:sameAs* is not necessarily a transitive, symmetric relation which holds in all contexts [65].

Consider an example where the symmetry of *owl:sameAs* is brought into question. This subtlety can be seen by considering two URIs related by *owl:sameAs* . The triple below indicates that the URI *person:10511* is the same as the URI *soton:10511* .

soton:10511 owl:sameAs person:10511 .

Now suppose that also the following two triples hold.

eprint:21769 dc:author person:10511 .
soton:10511 foaf:knows res:Hosni_Mubarak .

It is reasonable to assume that a further triple, below, can be inferred.

eprint:21769 dc:author soton:10511 .

This inference follows under the assumption that all triples which refer to the object of the *owl:sameAs* predicate are also relevant the subject. However, the owner of the triple *person:10511* may not want all triples of the information related to *soton:10511* (which describes personal information) to be associated with *person:10511* (which describes professional information). By assuming that *owl:sameAs* is not symmetric, this flexibility is permitted.

Consider two examples of asymmetry in practice. Suppose that a redirect, from a source URI to a target URI, produces an *owl:sameAs* triple. The redirect endorses the information about the target resource to be used for the source resource. However, the redirect does not necessarily endorse information about the source resource to be used for the target resource, since the redirect could be performed from anywhere. A different example context may instead insist that *owl:sameAs* is symmetric.

Few examples however would disagree that *owl:sameAs* is reflexive, since a URI is always the same as itself. It is also reasonable to assume that *owl:sameAs* is transitive. For instance, suppose that the following triple also holds further to the triples above.

person:10511 owl:sameAs soton:10511 .

Then inference can be applied twice to obtain the triple.

eprint:21769 dc:author person:10511 .

Under the assumptions of reflexivity and transitivity *owl:sameAs* is a preorder over URIs. Notice that the key features of RDFS are also preorders. So, also treating *owl:sameAs* as a preorder simplifies models for Linked Data. The mechanisms for *owl:sameAs* , *rdfs:subProperty* and *rdfs:subClass* can all be treated simultaneously. For instance, the following triple is often

considered valid if the two URIs are just URIs, but is not valid in a model where the two URIs are a classes.

foaf:Agent owl:sameAs dc:Agent .

Both the rich Description Logic approach and the light intuitive Linked Data approach to OWL are acceptable. This work falls under the lighter Linked Data style approach. This work assumes that the Description Logic approach is being tackled by the relevant community, and does not intend to interfere in that process. As such, OWL does not formally feature in this work, but is acknowledged due to its prominent status in the Semantic Web community.

2.2.4 SPARQL Queries

Perhaps the foremost mechanism for interacting with RDF is SPARQL Query. SPARQL Query stands for SPARQL Protocol and RDF Query Languages. The query language introduces constructs for observing patterns in an RDF store. The protocol describes the HTTP mechanisms and exchange formats for interacting with an endpoint of RDF store, by sending queries and receiving results. SPARQL Query is a major focus for this work. It is used as the basis of a high level language where queries and results can be tightly integrated, which allows the protocol to be hidden from the programmer. This section provides and overview of the standard syntax of queries.

SPARQL Query is based on basic patterns. A basic pattern is just some RDF in which variables may appear in place of URIs and literals. Variables in SPARQL Query are indicated by a question mark prefix to distinguish them from URIs. The syntax for basic patterns is in line with the N3 and Turtle formats for RDF as used in previous sections.

SPARQL Query defines four forms of query. These are ask, select, construct and describe. The ask, select and construct queries differ only in the format of results which are returned. A describe query is expected to return some RDF about a URI. Details of the describe queries are not formally specified, so are dependent on the implementation so ignored in this work. Note that describe queries are similar to dereferencing, which is an important feature of Linked Data. So dereferencing is also ignored in this work. This work focusses on ask and select queries.

2.2.4.1 Ask queries.

Ask queries are queries which can be answered by a straight forward yes or no. A yes answer indicates that a basic pattern can be matched. A no answer indicates that the query has failed to answer a query. For instance, the following query can be answered in the presence of the given data.

```
Data:
 eg:Hamish foaf:name "Hamish" .
```

```
Query:
 ASK { ?a foaf:name "Hamish" }
```

The variable in the query is implicitly existentially quantified. It is asking, "does there exist an ?a such that the basic pattern can be matched?" The existential quantifier will treated explicitly in this work.

2.2.4.2 Select queries.

Select queries return some results instead of just yes. The results returned are the possible bindings for selected variables. For instance, the following query explicitly selects two variables in a basic pattern. The result is two possible bindings for the selected variables.

```
Data:
 eg:Alice foaf:name "Alice" .
 eg:Alice foaf:knows eg:Bob .
 eg:Bob foaf:knows eg:Alice .
 eg:Bob foaf:name "Bob" .

Query:
 SELECT ?namex ?namey WHERE {
  ?x foaf:knows ?y .
  ?x foaf:name ?namex .
  ?y foaf:name ?namey .
 }

Results:
 {?namex -> "Alice", ?namey -> "Bob"}
 {?namex -> "Bob", ?namey -> "Alice"}
```

As with the select query the variables ?x and ?y are implicitly existentially quantified. The variables ?*namex* and ?*namey* are explicitly quantified, as indicated in the first part of the select query. This is also an form of existential quantification, however the explicitly quantified variables are used in the query results. The results of the query are the substitutions required to verify the existential quantification.

In this work, the results of a select query are modelled by immediately passing them as substitutions to some continuation. The continuation represents that program which uses the results. In this way the results of a query are hidden from the programmer. In the example above the two possible bindings are generated. The results can be seen as a choice of possible bindings, where

the result to be used in a continuation process could be chosen either non-deterministically or by some external user interaction.

This work also considers an alternative approach to the results returned, which accounts for the triples used in a query. Accounting for triples reduces repetition in query results, by ensuring that each result draws from distinct data sources. This approach is also natural for concurrency, where separate results are used simultaneously in parallel by continuation processes. This alternative (multiplicative) resource sensitive approach to query results is explained throughout Chapters 3 and 4.

2.2.4.3 Construct queries.

A construct query allows some RDF to be constructed according to a query. This corresponds to using the results of a select query as a substitution in a basic pattern. The result of the substitution is then returned as a result.

```
Data:
 eg:Hamish foaf:name "Hamish" .

Query:
 CONSTRUCT { ?x vcard:FN ?name }
 WHERE { ?x foaf:name ?name }

Result:
 eg:Hamish vcard:FN "Hamish" .
```

Construct and describe queries are be covered implicitly by continuations in this work. A continuation may be a program which uses the RDF returned, such as another query. Thus, as with ask and select, the query results are hidden from the user in a high level language.

2.2.4.4 Features for expressive queries.

More expressive queries can be posed using a number of operations on basic patterns. These include the binary operator `UNION`, and the unary operators `OPTIONAL` and `FILTER`. In this work the `UNION` and `FILTER` operators are primitive, whereas the `OPTIONAL` operator is an abbreviation.

The union operator offers a choice between two patterns. Only one of the patterns is used to produce a result. For instance, the following select query offers a choice between two patterns.

```
Data:
```

```
 eg:book dc11:title "SPARQL Query Tutorial" .

Query:
 SELECT ?title WHERE  {
  { ?book dc10:title  ?title } UNION { ?book dc11:title  ?title }
 }

Result:
 {?book -> eg:book1, ?title -> "SPARQL Query Tutorial"}
```

The keyword FILTER is used to embed constraints in a query. A constraint is something like a regular expression. Constrains are used to constrain literals in a query. For instance the following query uses a constraint to ensure that the variable is a string of characters in which a particular sub string appears.

```
Data:
 eg:book dc11:title "SPARQL Query Tutorial" .

Query:
 SELECT ?title WHERE {
   ?x dc:title ?title
   FILTER regex(?title, "^SPARQL")
 }

Result:
 {?title -> "SPARQL Query Tutorial"}
```

In this work strings are modelled by a Boolean algebra embedded in a query. Some constrains will not be covered, in particular `isBlankNode` which has few applications and complicates the model.

2.2.4.5 Extra features of SPARQL Query.

Some further features of SPARQL Query are considered in this work. The keyword `DISTINCT` ensures that the results of a query produce distinct bindings. This work considers queries which use distinct resources and queries which select names only once. The `LIMIT` keyword sets the maximum number of results returned by a query. Limits are only useful when the results are distinct, so provides further motivation for the control of resources employed in this work. Named graphs are also primitive in SPARQL Query, indicated using the keyword `GRAPH`. Named graphs are considered in Chapters 3 and 5.

The latest working draft of SPARQL Query includes new features. These include negation-as-failure, sub-queries and property paths. The model presented in this work is expressive enough to model negation-as-failure and sub-queries. Property paths can be captured by introducing a fixed point operator. These features were still under debate at the time of writing, so will not be discussed further.

2.3 Introduction to SPARQL Update

In October 2010 the first SPARQL Update W3C working draft with an operational semantics was released [53]. SPARQL Update is a development of an earlier proposal from Hewlett-Packard Labs [126]. The language is intended as a counterpoint to the SPARQL Query language [115], for fine grained updates on an RDF store.

The recommended semantics for SPARQL Query are based on the work of Pérez et al. [110], which provides a set based denotational semantics for idealised queries. In contrast, the semantics presented here for SPARQL Update are operational in nature. The difference between a denotational semantics and an operational semantics is that the former builds an external model (typically using sets), whereas the later is defined directly over the abstract syntax of the language.

There are several advantages of operational semantics. An operational semantics works like an interpreter, so is at an appropriate level for a compiler engineer (the primary target audience). Operational semantics is also suited to ad-hoc features which appear in real programming languages, which SPARQL Update intends to be. Furthermore, operational semantics are suited to specifying the complex long term behaviour of systems, including concurrency as required by servers. Denotational semantics for both application driven ad-hoc features and long term behaviour are notoriously difficult [4]. So operational semantics can easily and insightfully be adapted to SPARQL Query, but denotational semantics do not extend easily to SPARQL Update.

An analogy may help the reader. All readers are familiar with the concept of a regular expression or use tools which involve regular expressions. For instance, the replace tool in your text editor is appropriate for every day updates in text documents. SPARQL Update provides the power of regular expressions generalised appropriately to RDF. For the sake of clarity, here a core language is presented where only the default RDF graph is updated. The model can be extended with named graphs [38]. Also, the model can accommodate updates with respect to entailments, such as those defined in RDFS [33].

2.3.1 An example SPARQL Update.

No official recommendation of the SPARQL Update language exists at the time of writing. The drafts are not yet stable. However a concrete syntax similar to the concrete systax of SPARQL

Query is under development. Here an overview of the current proposed form for fine grained updates is provided.

An update consists of three clauses. The delete clause the insert clause and the where clause. The delete clause specifies the triples to be removed. The insert clause specifies triples to be inerted. The where clause specifies a query which must be answered for the update to be performed. The entire update occurs atomically, which ensures that all clauses are satisfied simultaneously.

The following example is adapted from the current working draft [53]. The update deletes triples where the name Bill is used and inserts a triple where Willian is used instead. The update can only happen if the subject of the triple is of RDF type person.

```
Data before:
 eg:president1 foaf:givenName "Bill" .
 eg:president2 foaf:givenName "Bill" .
 eg:president1 rdf:type foaf:Person .
 eg:president2 rdf:type foaf:Person .

Update:
 DELETE { ?person foaf:givenName "Bill" }
 INSERT { ?person foaf:givenName "William" }
 WHERE { ?person rdf:type foaf:Person }

Data after:
 eg:president1 foaf:givenName "William" .
 eg:president2 foaf:givenName "William" .
 eg:president1 rdf:type foaf:Person .
 eg:president2 rdf:type foaf:Person .
```

The above update can be expressed in an abstract syntax, to be defined in the next section, as follows.

```
DO SELECT ?person {
 DELETE {?person foaf:givenName Bill}
 JOIN
 INSERT {?person foaf:givenName William}
 JOIN
 ASK {?person rdf:type foaf:Person}
}
```

The abstract syntax is more compositional than the concrete syntax. The abstract syntax introduces an explicit join operator which forces parts of a query to occur simultaneously. The explicit join operator allows the three clauses to be defined separately and then joined. The abstract syntax also introduces a select quantifier, which binds the free variables. This clarifies

the scope of variables and leads to a clearer semantics. Finally, the abstract syntax introduces an explicit interation operator. An explicit interation operator indicates when an update is applied repeatedly, and allows some updates to be expressed that cannot be achieved using a single update in the current concrete syntax. The operational semantics are defined over the abstract syntax. A translation from the concrete syntax to the abstract syntax can be provided.

2.4 A Syntax for SPARQL Update

This section presents an abstract syntax used to define an operational semantics for SPARQL Update. The abstract syntax is intended for the purpose of compiler engineering (as opposed to exchange of messages). Three grammars are sufficient to specify an abstract syntax: one for RDF Terms; one for constraints; and a third for SPARQL Updates. Curly brackets are used to resolve ambiguity in examples.

2.4.1 A Syntax for RDF Terms

The following grammar presents an abstract syntax for RDF. Several concrete syntaxes have been proposed for RDF, such as Turtle and N3 for the purpose of tersely presenting RDF to humans [16, 23]. In contrast, the following abstract syntax for RDF Terms is presented at an appropriate level for compiler engineering.

object ::=	`'literal'`	a literal
|	*?variable*	a variable
|	*URI*	a URI

Term ::=	`NOTHING`	the empty term
|	*Term* , *Term*	par
|	`LOCAL` *URI Term*	a local name
|	{*URI URI object*}	a triple

Two forms of triple represent RDF triples, with either a URI or a literal as the object. A variable indicates an unknown literal. Nothing represents the empty graph, which contains no RDF triples. The operator par composes RDF Terms, thus for instance two triples can be composed to form a larger RDF Term. The Local quantifier indicates a local name (which represents a blank node). The local quantifier binds occurrences of a URI in an RDF Term.[1]

The following presents two triples which share a common local name (blank node) as their subject.

[1] Par is a simple syntactic composition, and does not imply any set theoretic composition (union of graphs for instance). Similarly, Local is a simple syntactic binding for resolving blank nodes, and is not necessarily existential quantification. However, graphs may be recovered from this syntax if required by an application [78].

```
LOCAL :a {
  {:a foaf:familyName "Carrol"} ,
  {:a foaf:knows eg:Klyne}
}
```

2.4.2 A Syntax for Constraints

Constraints are defined fully in the SPARQL Query recommendation [115], hence a complete grammar for constraints is not detailed here. The following is enough to suggest that constraints form a Boolean algebra with built in primitives. Constraints may contain variables and URIs.

Constraint ::=	`true`	true
|	`false`	false
|	*Constraint* && *Constraint*	and
|	*Constraint* || *Constraint*	or
|	!*Constraint*	not
|	`regex`(*?variable*, *RegularExpression*)	regular expression
|	...	etc.

A constraint is satisfied if and only if it evaluates to true. The evaluation of constraints is detailed in the SPARQL Query recommendation [115]. Examples of constraints include regular expressions parametrised on a variable and inequality tests on numbers.

2.4.3 A Syntax for SPARQL Update

The following grammar proposes an abstract syntax for SPARQL Updates. A successful update results in an atomic change to an RDF store. This abstract syntax allows constructs to be nested.

Update ::=	`DELETE` *Term*	delete a term
|	`INSERT` *Term*	insert a term
|	`FILTER` *Constraint*	impose a constraint
|	*Update* `CHOOSE` *Update*	choose a branch
|	*Update* `JOIN` *Update*	synchronise updates
|	`SELECT` *URI Update*	select a URI
|	`SELECT` *?variable Update*	select a literal
|	`DO` *Update*	iteratively apply an update

Delete removes the indicated RDF Term from the store. Insert introduces an RDF Term to the store. Filter imposes a constraint on an update. Choose offers the choice of either a left or right update. Join ensures that two updates happen in the same atomic update. Select parametrises an update on either a URI or a literal which is not known in advance. (Note that in this abstract syntax, URIs and literals are distinguished in Selects for clarity.) Iteration (DO) performs an update zero, one, two or more times, in the same atomic update. Without iteration an update is applied once.

Examples of each construct are provided along with the operational semantics for the construct in Section 2.6.

2.4.4 Abbreviations for Common Updates

A number of common updates can be defined using the basic updates above. The use of abbreviations avoids redundancy in the operational semantics.

A unit update is answered trivially without requiring any RDF term. The unit update can be defined using the true constraint, which is always satisfied, as follows.

$$\texttt{SKIP} \triangleq \texttt{FILTER}\,\texttt{true} \qquad \text{unit update}$$

An optional update gives the choice of performing an update or not performing an update. The optional update can be defined by a choice between an update and the unit update as follows.

$$\texttt{OPTIONAL}\ \mathit{Update} \triangleq \mathit{Update}\ \texttt{CHOOSE SKIP} \qquad \text{optional update}$$

Successive select queries are can be combined. The combined variables are listed in a single select quantifier, as follows.

$$\begin{array}{l}\texttt{SELECT}\ \mathit{?variable0}\ \mathit{?variable1}\\ \quad \mathit{Update}\end{array} \triangleq \begin{array}{l}\texttt{SELECT}\ \mathit{?variable0}\\ \quad \texttt{SELECT}\ \mathit{?variable1}\\ \quad\quad \mathit{Update}\end{array}$$

In this chapter queries will be encoded naïvely, using the unary keyword `ASK`. The effect of a query can be achieved by joined insert and delete, as follows.

$$\texttt{ASK}\,\mathit{Term} \triangleq \texttt{INSERT}\,\mathit{Term}\ \texttt{JOIN}\ \texttt{DELETE}\,\mathit{Term}$$

The joined delete and insert has the effect of a querying for a term, since the term deleted must exist for the delete to be applied but the insert immediately replaces the deleted term in the same atomic operation. Later in this work queries will be primitive, but the focus here is on updates.

Further abbreviations can capture concepts which appear in a standardised concrete syntax.

2.5 An Equivalence over RDF Terms

This section identifies equivalent syntax. A syntactic equivalence imposes less constraints on RDF than any requirement that collections of triples are sets. Instead, obviously equivalent syntax is considered to serve the same purpose, as defined by a structural congruence.

2.5.1 A Structural Congruence

A structural congruence, written = below, is a relation between RDF Terms. A congruence is an equivalence relation (reflexive, symmetric and transitive) which holds in all contexts. The structural congruence satisfies the following equations — unit, commutativity and associativity respectively.

Unit: *Term* , `NOTHING` = *Term* Commutativity: *Term0* , *Term1* = *Term1* , *Term0*

Associativity: *Term0* , {*Term1* , *Term2*} = {*Term0* , *Term1*} , *Term2*

Structural congruences can be applied at any point, when evaluating the operational semantics.

Example of Applying the Structural Congruence. The following RDF Data can be used interchangeably. If one appears in a rule in the next section then it can be replaced by the other.

{*eg:book1 ns:price* `5`} ,
`NOTHING` ,
{
  {*eg:book2 dc:title* `"Linked Data"`} ,
  {*eg:book1 dc:title* `"Web of Data"`}
}

=

{
  {*eg:book1 dc:title* `"Web of Data"`} ,
  {*eg:book1 ns:price* `5`}
} ,
{*eg:book2 dc:title* `"Linked Data"`}

Brackets are used similarly for Group Graph Patterns in SPARQL Query [115]. Associativity of par allows brackets to be omitted for readability.

2.6 Commitment Relations for SPARQL Updates

Commitment relations specify single atomic changes which can be made to an RDF store. Atomicity focuses on the local effect of an update. Only the exact RDF Terms which are required to perform an atomic update are accounted for. An advantage of this approach is that the data in a commitment reduction is exactly the data to be locked to ensure that an update occurs atomically.

A commitment relation consists of the data before an update, an update and the data after an update. The data before is the exact RDF Term which is consumed by the update. The update is exactly the update which is applied. The data after is exactly the RDF Term which is expected to replace the original RDF Term once the commitment has been performed. Commitment relations are axioms of the form.

Data before: *Term* Update: *Update* Data after: *Term*

Commitment relations can also be derived from rules. The premises of a rule are a number of commitment relations and the conclusion is a single commitment relation of the above form. The conclusion holds only if all the premises hold. The axioms and rules which specify operational semantics for SPARQL Update are defined throughout this section.

2.6.1 The Delete Axiom

The Delete Axiom removes an RDF Term from the store. The committed RDF Term, *Term*, and committed delete update, `DELETE` *Term*, interact. After the interaction both the term and the delete update are removed from the store. The data after the update is the empty RDF term.

Data before: *Term* Update: `DELETE` *Term* Data after: `NOTHING`

Example of the Delete Axiom. The following triple can be removed by the following update due to the following commitment relation. This commitment relation is an instance of the Delete Axiom.

Data before:

{*eg:book2 dc:title* `"The Semantic Web"`}

Update:

`DELETE` {*eg:book2 dc:title* `"The Semantic Web"`}

Data after:

`NOTHING`

2.6.2 The Insert Axiom

The Insert Axiom adds a designated RDF Term to the store. The designated RDF Term is indicated by the INSERT keyword. The data after the update is the inserted RDF Term.

Data before: `NOTHING` Update: `INSERT` *Term* Data after: *Term*

Example of the Insert Axiom. The two triples below can be inserted into anything (since nothing is required), due to the following commitment relation. This commitment relation is an instance of the Insert Axiom.

Data before:

```
NOTHING
```

Update:

```
INSERT{
  {eg:book1 dc:title "SPARQL Tutorial"} ,
  {eg:book1 eg:price 42}
}
```

Data after:

```
  {eg:book1 dc:title "SPARQL Tutorial"} ,
  {eg:book1 eg:price 42}
```

2.6.3 The Join Rule

The Delete Axiom and the Insert Axiom allow basic updates to take place where either the exact RDF Term to be deleted is known, or the exact RDF Term to be inserted is known, respectively. For more substantial updates, rules are required to build commitment relations. The first of these rules is the Join Rule.

The Join Rule ensures that two updates occur atomically, in the same commitment relation. If one update has one effect and another update has another effect, then the join of the updates is their combined effect. The rule ensures that both updates act on separate RDF Terms. Suppose that the following commitment relation holds.

Data before: *Term0* Update: *Update0* Data after: *Term2*

Also, suppose that the following commitment relation holds.

Data before: *Term1* Update: *Update1* Data after: *Term3*

The two commitment relations above can be combined to produce the following commitment relation.

Data before: *Term0* , *Term1* Update: *Update0* `JOIN` *Update1* Data after: *Term2* , *Term3*

Example of Joined Updates. The update below demonstrates three joined updates. The first two updates remove the two triples present. The third update inserts a triple. Thus the combined update removes both triples and adds a new triple atomically.

Data before:

{*eg:book1 dc:title* "SPARQL Tutorial"} ,
{*eg:book2 dc:title* "The Semantic Web"}

Update:

DELETE {*eg:book1 dc:title* "SPARQL Tutorial"}
JOIN
DELETE {*eg:book2 dc:title* "The Semantic Web"}
JOIN
INSERT {*eg:book2 dc:title* "The Web of Linked Data"}

Data after:

{*eg:book2 dc:title* "The Web of Linked Data"}

2.6.4 The Select Literal Rule and Select URI Rule

The Select Literal Rule is parametrised on a variable. The variable is bound to the update indicated (so cannot be referred to from outside the select). The Select Rule allows any literal which enables a commitment to be substituted for the variable. The data after the commitment with the variable substituted for a literal is used as the data after the commitment with the variable bound by a Select. Note that substitution is indicated by square brackets where the literal on the left replaces the variable on the right. Suppose that the following commitment relation holds.

Data before: *Term0* Update: *Update*['literal'/*?variable*] Data after: *Term1*

Given the commitment relation above the following commitment relation holds.

Data before: *Term0* Update: SELECT *?variable Update* Data after: *Term1*

The Select URI Rule has the same shape. In the case of URIs, a correct URI to input is substituted for the temporary URI which is bound in the Select expression. Thus two URIs replace both the variable and literal in the Select Literal Rule.

Example of the Select Literal Rule. The following example demonstrates how Select can be used to delete an RDF Term which involves a literal not known in advance. The update deletes a triple in which the variable *?title* appears. The variable can be instantiated with the literal

`"SPARQL Tutorial"`. Thus the delete matches the committed triple. Therefore the following commitment is valid.

Data before:

{*eg:book1 dc:title* `"SPARQL Tutorial"`}

Update:

`SELECT` *?title* {
`DELETE` {*eg:book1 dc:title ?title*}
}

Data after:

`NOTHING`

2.6.5 The Choose Left Rule and Choose Right Rule

The Choose Rules allow one of two updates to be committed. The choose rule has a left and right form, where respectively the left or right update is applied. The data after a choice is the same as the data after applying the chosen branch. Consider the Choose Left Rule and suppose that the following commitment relation holds.

Data before: *Term0* Update: *Update0* Data after: *Term1*

Given the above commitment relation, the following commitment relation holds.

Data before: *Term0* Update: *Update0* `CHOOSE` *Update1* Data after: *Term1*

The rule above chooses the left update. The Choose Right Rule is the symmetric rule which chooses the right branch instead.

Example of a Choice of Updates. The following demonstrates an update where either the first delete or second delete may be triggered. The two branches use different versions of the Dublin Core metadata vocabulary. In this case, the committed RDF Term matches the right branch. The effect is that the committed triple is deleted.

Data before:

{*eg:book dc11:title* `"SPARQL Protocol Tutorial"`}

Update:

```
SELECT ?title {
  DELETE {eg:book dc10:title ?title}
  CHOOSE
  DELETE {eg:book dc11:title ?title}
}
```

Data after:

```
NOTHING
```

2.6.6 The Filter Axiom

The Filter Axiom imposes a constraint on an update. The constraint is disposed only if the constraint evaluates to true. If the constraint does not evaluate to true then the update is blocked. The procedure for deciding whether a constraint holds is specified in the SPARQL Query Recommendation [115]. Given that the constraint evaluates to true the following commitment relation holds.

Data before: `NOTHING` Update: `FILTER` *Constraint* Data after: `NOTHING`

An Example of a Filtered Update. The following commitment relation holds. The update deletes the title of a book, where the title and the book are discovered using Select. The filter imposes the constraint that the title must also satisfy a regular expression. The literal in the committed triple does matche the regular expression. The triple is deleted.

Data before:

```
{eg:book1 dc:title "SPARQL Tutorial"}
```

Update:

```
SELECT :a ?title {
  DELETE {:a dc:title ?title}
  JOIN
  FILTER regex (?title, "^SPARQL")
}
```

Data after:

```
NOTHING
```

2.6.7 The Rules for Iterated Updates

All updates above are applied exactly once. Often the update should be applied wherever possible in an RDF store. This is achieved by iteration. The rules for iteration are similar to those for a Kleene star in a regular expression. Regular expressions are commonly used to update text files. This work is a generalisation of this common technique to RDF stores.[2]

Updates can be applied any number of times. Iteration is used when the number of times to apply an update is not known. The Weekening Axiom allows an interated update to be applied zero times, if there is no term which matches the update. The Weakening Axiom terminates an iterated update with no effect.

Data before: `NOTHING` Update: `DO` *Update* Data after: `NOTHING`

The Dereliction Rule allows an iterated update to be applied once. Assume that an update can be committed in the presence of some term resulting another term. Dereliction allows the same update but iterated to be committed in the presence of the same term with the same resulting term. Suppose that the following commitment relation holds.

Data before: *Term0* Update: *Update* Data after: *Term1*

Given the above commitment relation, the following commitment relation holds.

Data before: *Term0* Update: `DO` *Update* Data after: *Term1*

The Contraction Rule allows two copies of an iterated update to be simultaneously committed. Contraction can be applied repeatedly, along with the Join Rule and Dereliction Rule, to simultaneously commit any number of copies of an iterated update. Suppose that the following commitment relation holds.

Data before: *Term0* Update: `DO` *Update* `JOIN DO` *Update* Data after: *Term1*

Given the commitment relation above, the following commitment relation holds.

Data before: *Term0* Update: `DO` *Update* Data after: *Term1*

The combination of the Weakening, Dereliction and Contraction rules allow zero, one, two, or more copies of an iterated update to be atomically committed. The use of Join in the Contraction Rule ensures that disjoint RDF Terms are used for each copy.

[2]Generalisations of regular expression date back to the commutative regular algebras of J. H. Conway [41], and remain a prominent rejuvinated area of research. The majority of computer scientists and engineers use regular expressions or tools based on regular expressions.

An Example of an Iterated Update. The following demonstrates an iterated update. The update replaces occurrences of the predicate *dc11:title* with the predicate *dc:title*. The iteration of this update means that the update can be applied twice. The result is that two triples are committed and replaced by two new triples.

Data before:

{*eg:book1 dc11:title* "Query Tutorial"} ,
{*eg:book2 dc11:title* "Update Tutorial"}

Update:

DO SELECT *:a*, ?*x* {
  DELETE {*:a dc11:title* ?*x*}
  JOIN
  INSERT {*:a dc:title* ?*x*}
}

Data after:

{*eg:book1 dc:title* "Query Tutorial"} ,
{*eg:book2 dc:title* "Update Tutorial"}

2.6.8 An Example of a Nested Update.

This example, firstly, demonstrates most of the constructs combined to answer a larger update. Secondly, it demonstrates a common scenario which is enabled by nested selects and nested explicit iteration, which is impossible to express as an atomic update in initial proposals for SPARQL Update [126, 53]. Consider the following commitment, which removes all *foaf:knows* links to people younger than 21.

Data before:

{*eg:youth0 eg:dob* 01-01-2010} ,
{*eg:youth1 eg:dob* 01-02-2010} ,
{*eg:person foaf:knows eg:youth0*} ,
{*eg:person foaf:knows eg:youth1*} ,
{*eg:youth0 foaf:knows eg:youth1*} ,

Update:

```
DO SELECT :a ?dob {
  DELETE {:a eg:dob ?dob}
  JOIN
  INSERT {:a eg:dob ?dob}
  JOIN
  FILTER (current-year − year(?dob) < 21)
  JOIN
  DO SELECT :b {
   DELETE {:b foaf:knows :a}
  }
}
```

Data after:

```
{eg:youth0 eg:dob 01-01-2010} ,
{eg:youth1 eg:dob 01-02-2010}
```

Without the nested iteration and selects, the effect of the above update could only be achieved using two updates. This means that the update would not be atomic. The above update is correct and atomic. This example highlights a common problem which also appears in the first SPARQL Query recommendation [115]. This illustrates an improvement made by this work to the expressiveness of the language.

2.7 Reduction Relations for Concurrent RDF Stores

An RDF store deployed on servers requires many updates to occur concurrently. The behaviour of multiple updates can be specified by a reduction relation. The reduction relation provides a context for applying many commitment relations to an RDF store. The reduction relation also specified how blank nodes are treated by updates.

2.7.1 A Syntax for SPARQL Processes

A syntax for processes internalises RDF Terms and SPARQL Updates. Processes represent the state of part of an RDF store.

Process ::=	`NOTHING`	nothing
|	*Process* , *Process*	par
|	`LOCAL` *URI Process*	block
|	*Update*	query
|	*Term*	resource

A reduction relation consists of two processes: the process before the reduction; and the process after the reduction. Reduction relations are defined by the axioms and rules in this section.

2.7.2 The Idle Axiom for Unaffected Processes

The Idle Axiom allows a process to do nothing. This axiom indicates that a reduction has no effect on the part of the store described by the process.

Before: *Process* After: *Process*

Note that other structural axioms and rules may be added to specify further behaviour of a store. The Idle Axiom is the minimum required.

2.7.3 The Action Rule for a Commitment Acting on a Reduction

The Action Rule allows a local update to be applied to part of the store. The Action Rule indicates that a commitment relation (representing a local update) can be performed in the presence of some reduction relation (representing the behaviour of a separate part of the store). The commitment and the reduction do not interfere. Firstly, assume that the following commitment relation holds.

Data: *Term0* Update: *Update* Result: *Term1*

Secondly, assume that the following reduction relation holds.

Before: *Process0* After: *Process1*

Given the above commitment relation and the above reduction relation, the following reduction relation holds.

Before: *Term0* , *Update* , *Process0* After: *Term1* , *Process1*

The action rule can be applied repeatedly to capture any number of concurrent updates on an RDF store.

An Example of the Action Rule and the Idle Axiom. The following demonstrates an action on a small part of a store. The Idle Axiom allows one triple not to be touched. Another triple is updated by a commitment relation. The commitment relation is applied by the Action Rule. This simple pattern can be extended to an entire RDF store.

Before:

{
  DELETE{*eg:book2 dc:title* "The Semantic Web"}
  JOIN
  INSERT{*eg:book2 dc:title* "The Web of Linked Data"}
} ,
{*eg:book2 dc:title* "The Semantic Web"} ,
{*eg:book2 eg:price* 23}

After:

{*eg:book2 dc:title* "The Web of Linked Data"} ,
{*eg:book2 eg:price* 23}

2.7.4 The Local Rule for Handling Blank Nodes

The Local Rule is used for updates which involve blank nodes. The trick is to choose a temporary URI, which in the premise of the rule replaces the URI bound by Local. The temporary URI must not appear free in the conclusion of the rule. Suppose that the following reduction relation holds, where *:e* is a temporary URI.

Before: *Process0* , *Process1*[*:e*/*:a*] After: *Process2* , *Process3*[*:e*/*:b*]

Given that the above reduction relation holds and that *:e* does not appear free below, then the following reduction relation holds.

Before: *Process0* , LOCAL *:a Process1* After: *Process2* , LOCAL *:b Process3*

An Example of the Local Rule. The following example demonstrates a blank node updated. A temporary URI can represent *:a* in the premise of the Local Rule. This allows the update to be considered as if *:a* is not bound. One triple is deleted by a commitment relation, which discovers the temporary URI. However, the conclusion of the Local Rule still binds *:a*. This has the effect of discovering the blank node and using it in an update.

Before:

```
SELECT :b
 DELETE {:b foaf:mbox mailto:alice@example.org} ,
LOCAL :a {
  {:a foaf:name Alice} ,
  {:a foaf:mbox mailto:alice@example.org}
}
```

After:

```
LOCAL :a {:a foaf:name Alice}
```

2.7.5 A Substantial Example of Concurrent Updates Involving Blank Nodes

The following demonstrates two updates, which happen concurrently despite not being joined. One triple is idled. The Action Rule is applied twice to trigger the two updates. There is also a blank node which is selected and used in both updates. Notice that the scope of the blank node is maintained after the reduction.

Before:

```
SELECT :a
 DELETE {:a foaf:name Alice} ,
DO SELECT :b {
  DELETE {eg:Boss eg:employee :b}
  JOIN
  INSERT {:b eg:employer eg:Boss}
} ,
```

```
LOCAL :c {
  {eg:Boss eg:employee :c} ,
  {:c foaf:name Alice} ,
  {:c foaf:mbox mbox:alice@example.org}
} ,
{eg:Boss eg:employee eg:Bob}
```

After:

```
LOCAL :c {
  {:c eg:employer eg:Boss} ,
  {:c foaf:mbox mbox:alice@example.org}
} ,
{eg:Bob eg:employer eg:Boss}
```

All examples are entirely specified by the operational semantics in this paper.

2.8 Conclusions on the Specification

This chapter proposes an operational semantics for SPARQL Update. A fine grained update language is essential for using RDF stores in modern Web applications, where contributing

content is as important as consuming content. This proposal is a close counterpoint to the SPARQL Query recommendation, from which examples are adapted [115]. This proposal for an update language allows fine control of updates, as found in regular expressions. Updates are covered which cannot be expressed atomically in existing proposals, such as the example in Section 2.6.8. The finer control ensures that common updates can be performed atomically.

The rôle of this chapter in the context of this work is to demonstrate that an operational semantics can be presented without any meta-theory. The abstract syntax is purely ASCII so can be implemented directly. The operational semantics are described using only English and the abstract syntax itself, so no meta-symbols for the sole purpose of expressing the operational semantics are required. Furthermore, the operational semantics are defined so that the updates that hold are expressed in the same intuitive form as examples in the recommendation, which clearly indicate the data before the update, the update and the data after the update. Thus all correct examples are directly derived by the operational semantics. The intention is to find a presentation which can be understood by compiler engineers with diverse backgrounds.

Chapter 3

Reduction Systems for Read–Write Linked Data

This section defines an operational semantics of a high level programming language for Linked Data. The operational semantics are defined using a deductive system, where the atomic transitions permitted are provable using the deductive system. The deductive system allows key features of a programming language for Linked Data to be captured simultaneously, including queries, updates, reasoning and constraints. The concept of a region, modelling named graphs and feeds is also investigated.

The background material for this section is provided retrospectively, by comparing the model developed to existing models. The operational semantics presented was produced iteratively, by directly modelling the operational behaviour of real languages for Linked Data, as described in the previous chapter.

Existing models have also provided inspiration for design decision, including process calculi [20, 99] and Linear Logic [55]. A famous process calculus, the π-calculus, is compared to the syndication calculus. In particular, it is demonstrated that the π-calculus can be expressed using the same connectives as the syndication calculus; thus both calculi can be tightly integrated in one powerful framework for operational semantics. Another inspiration for the model is Linear Logic. The connectives of Linear Logic are compared to the analogous connectives of the syndication calculus.

Although neither the π-calculus nor Linear Logic are used directly to produce the syndication calculus, they provide inspiration for design decisions. Like the syndication calculus, both the π-calculus and Linear Logic provide deductive systems which can be applied to reason about interacting concurrent systems. Thus an intermediate calculus provides further insight into the connection between these two systems [3, 98, 18, 17].

The background material clarifies that classical logic is not being applied in this work. Most logical systems considered for the Web have been justified using classical model theory [74];

yet modern logics for computer science rarely fit into the classical box. Modern logics have been driven by common problems in computer science, including typing computable functions [42], accounting for resources [119], and understanding communication and concurrency [61]. Such problems are evident when considering the Web of Data. Thus the deductive system presented in this chapter, is a step towards a modern and appropriate logic for the Web of Data.

3.1 Motivating Examples for the Reduction System

This chapter systematically introduces a syntax and operational semantics for a high level language for Linked Data. The language tightly combines the key technologies for Linked Data, including queries, updates and reasoning. The language is concurrent, which is appropriate for servers on which systems which use Linked Data are deployed. Before embarking on a systematic definition and explanation, some examples are provided along with their intuition.

3.1.1 Simple sentences about Joe Armstrong the footballer

Firstly, consider some Linked Data presented in RDF. This particular data set is taken from DBpedia which lifts RDF data from Wikipedia [28]. The data presents triples where the subject is a retired footballer called Joe Armstrong. The URI for Armstrong provided by DBpedia is http://dbpedia.org/resource/Joe_Armstrong_(footballer), which is identified by *Armstrong* in examples. This allows this footballer to be disambiguated from, amongst others, Joe Armstrong the programmer, who is assigned a distinct URI.

The data obtained from Wikipedia is interpreted as RDF triples. To do so, a URI for a predicate has been created to distinguish each piece of information about Armstrong. Where possible, a predicate from a popular metadata vocabulary has been chosen. For instance, the FOAF vocabulary provides a predicate for the name of a person. Other predicates have been created specially to correspond with the data obtained from Wikipedia. For instance, predicates are provided for the position, birthPlace, caps, etc, in a name space owned by DBpedia. The name spaces used are the following.

foaf: `http://xmlns.com/foaf/0.1/`
res: `http://dbpedia.org/resource/`
p: `http://dbpedia.org/property/`
dbp: `http://dbpedia.org/ontology/`

Some predicates indicate literal values. For instance, the name predicate indicates a string, the caps predicate indicates a natural number and the birthDate predicate indicates a date. Other predicates indicate other resources identified by URIs. For instance, both the city Newcastle-upon-Tyne and the football club Gateshead F.C. have URIs. A special predicate – *rdf:type* – from the core RDF vocabulary is used. This predicate is used to indicate that Armstrong is a

footballer. The class of footballers has a URI which can be treated like any other URI. In the syntax of this chapter, the data set can be represented as follows.

(*Armstrong foaf:name* `'Joe Armstrong'`),
(*Armstrong rdf:type dbp:SoccerPlayer*),
(*Armstrong dbp:position res:Inside_forward*),
(*Armstrong dbp:birthPlace res:Newcastle_upon_Tyne*),
(*Armstrong dbp:birthDate* '29-01-1939'),
(*Armstrong p:clubs res:Gateshead_F.C.*),
(*Armstrong p:caps* 22),
(*Armstrong p:goals* 9)

Extra information is also provided which allows some further information to be inferred from the data above. Given that someone is a footballer, it can be inferred that the person is an athlete. It can also be inferred that the an athlete is indeed a person. Thus the two relationships below can be assumed. These relationships define a preorder over URIs which can be used when performing queries and updates.

$$\textit{dbp:SoccerPlayer} \sqsubseteq \textit{dbp:Athlete} \qquad \textit{dbp:Athlete} \sqsubseteq \textit{foaf:Person}$$

Similarly, a distinct URI may identify Newcastle-upon-Tyne. For instance, a search on the Web site sameAs.org [57] suggests that almost 100 different URIs can be used to identify Newcastle-upon-Tyne, including `http://data.ordnancesurvey.co.uk/id/7000000000009784`. Using the same preorder relationship the URIs for Newcastle-upon-Tyne can be related in both directions. This allows the URIs to be used interchangeably in queries and updates. The formal definition shall make precise how RDF triples are defined and how the preorder over URIs is used to reason over RDF triples.

3.1.2 Compound sentences enquiring about footballers

This section considers some queries over the example data provided. Processes are constructed using a number of operators. Each operator is simple, but the combined result is a highly expressive language. Some common constructs are employed to provide the intuitive examples in this section. The constructs will be considered in detail later in this chapter. In the meantime, we consider again the footballer Joe Armstrong.

One way to find Armstrong from a store, which includes the data given in the previous section, would be to pose a query. The example query below asks for a footballer associated with the Gateshead Football Club. The query consists of two triple patterns to match, which are joined together to indicate that they must be answered in the same atomic step. A quantifier indicates that the subject of both triples should be the same. The query is triggered by the two triples

provided. The state before the atomic step is indicated on the left of the triangle and the state after the atomic step is indicated on the right of the triangle.

$$\begin{array}{l}(\textit{Armstrong p:clubs res:Gateshead\_F.C.})\,,\\ (\textit{Armstrong rdf:type dbp:SoccerPlayer})\,,\\ \bigvee a.\left(\begin{array}{l}|(a\ \textit{p:clubs res:Gateshead\_F.C.})|\\ |(a\ \textit{rdf:type dbp:SoccerPlayer})|\,;\\ P\end{array}\right)\end{array} \quad \triangleright \quad \begin{array}{l}(\textit{Armstrong p:clubs res:Gateshead\_F.C.})\,,\\ (\textit{Armstrong rdf:type dbp:SoccerPlayer})\,,\\ P\left\{{}^{\textit{Armstrong}}/_{a}\right\}\end{array}$$

Notice that after this reduction, the continuation process has the URI for Armstrong. Thus the URI can be used in that process. The URI may be used to access further information from the example data above; or used to manipulate the data using an update; or, perhaps, used to perform some other operation such as passing the URI to another process on a channel. The URI is passed to the continuation process because the continuation process is bound by the quantifier which discovers the URI.

Using DBpedia, a description of the resource identified by *res:Inside_forward* can be obtained by using a query. This description provides the information that an inside forward was a position in football popular until the first half of the 20th century. Instead of an inside forward modern football teams now use an attacking midfielder. This provides a rationale for the example below. The following process is an update which turns all inside forwards born before 1950 into attacking midfielders. The state before the update is on the left of the triangle and the state after the update is on the right of the triangle.

$$\begin{array}{l}(\textit{Armstrong dbp:birthDate}\ \text{`29-01-1939'})\,,\\ (\textit{Armstrong dbp:position res:Inside\_forward})\,,\\ \qquad *\bigvee a.\left(\begin{array}{l}\bigvee x.\left(\begin{array}{l}|(a\ \textit{dbp:birthDate}\ x)|\\ (x \leq \text{`01-01-1950'})\end{array}\right)\\ (a\ \textit{dbp:position res:Inside\_forward})^{\perp}\\ (a\ \textit{dbp:position res:Attacking\_midfielder})\end{array}\right)\end{array} \quad \triangleright \quad \begin{array}{l}(\textit{Armstrong dbp:birthDate}\ \text{`29-01-1939'})\,,\\ (\textit{Armstrong dbp:position res:Attacking\_midfielder})\end{array}$$

The form of the update is similar to the query above. The update consists of triples composed together. However, for updates the triples are distinguished in three ways. The first triple above finds a birth date, so is a query; the second triple above removes the position inside forward, so is a delete; the third triple above includes the position attacking midfielder, so is an insert. There is also a constraint on the birth date composed with these triples. All of these components — the query, delete, insert and constraint — are performed in the same atomic step. This means that the insert can only take place if the query and constraints are satisfied and the delete is successful.

The operator for composition, the tensor product, is the main feature for composing compound atomic actions, which provides the necessary expressive power for the update language. The

tensor product is defined and discussed in this chapter. The star in front of the update above indicates that this update can be applied multiple times, meaning that more than one insider forward may be updated to an attacking midfielder in one atomic step.

Another larger example is provided. This example demonstrates how operators of the language can express complex queries. The query below asks for either artists or athletes. The query also asks for people with the surname Armstrong and a forename beginning with the character 'J'. The name of the person can be obtained in two ways: either from a single name predicate or by combining first name and last name predicates. For this query it is also assumed that *dbp:SoccerPlayer* $\sqsubseteq$ *dbp:Athlete*. This allows the more specific information that Armstrong is a footballer to be used to answer the part of the query which asks whether he is an artist or an athlete.

$$
\begin{array}{l}
(\textit{Armstrong foaf:name } \texttt{'Joe Armstrong'}),\\
(\textit{Armstrong rdf:type dbp:SoccerPlayer}),\\
\bigvee a.\left(\begin{array}{l}
\bigvee z.\left(\begin{array}{l}
\left(\begin{array}{l}
\bigvee x, y.\left(\begin{array}{l}
|(a\ \textit{foaf:givenName}\ x)|\\
|(a\ \textit{foaf:familyName}\ y)|\\
(z = x + \text{`\ '} + y)
\end{array}\right)\\
\oplus\\
|(a\ \textit{foaf:name}\ z)|
\end{array}\right)\\
(z \in \texttt{'J.* Armstrong'})
\end{array}\right)\\
\left(\begin{array}{l}
|(a\ \textit{rdf:type dbp:Athlete})|\\
\oplus\\
|(a\ \textit{rdf:type dbp:Artist})|
\end{array}\right);\\
P
\end{array}\right)\\
\qquad\qquad (\textit{Armstrong foaf:name } \texttt{'Joe Armstrong'}),\\
\qquad\qquad \rhd\ (\textit{Armstrong rdf:type dbp:SoccerPlayer}),\\
\qquad\qquad P\left\{{}^{\textit{Armstrong}}/_{a}\right\}
\end{array}
$$

A single language which includes all the features of queries and updates and continuation processes, with concurrency is proposed. The example atomic transitions of this section can all be derived using the deductive system introduced in this chapter. The final example above will also be used as a running example in subsequent chapters, to motivate results in those chapters.

3.1.3 Motivation for named graph features

This chapter also introduces the idea of a named graph, which is argued to be closely related to the idea of a feed. For now consider the idea that some applications need to know where information comes from i.e. the provenance of data.

For an intuitive motivation for named graphs, the football running example is continued. This example demonstrates an update involving a named graph. The data provided includes a triple where the provenance of the triple is indicated by an extra URI (the name of the 'graph' from

which the triple is obtained). The URI for the graph is described, using triples. The description indicates that the graph contains player statistics for October 1962.

Now, suppose that the player goal count for this month was not included in the goal tally of the players. The following update rectifies this missing information as follows. It finds a named graph which contains the player statistic for that month. It also updates the goal tally of the player, by adding the number of goals scored that month to the running total. The result is the the following atomic transition.

$$
\begin{array}{l}
(\mathit{Armstrong}\ \mathit{p{:}goals}\ 9)\,, \\
\mathcal{G}_{\mathit{eg{:}results\_Oct1961}}(\mathit{Armstrong}\ \mathit{p{:}goals}\ 2)\,, \\
(\mathit{eg{:}results\_Oct1961}\ \mathit{eg{:}month}\ 10)\,, \\
(\mathit{eg{:}results\_Oct1961}\ \mathit{eg{:}year}\ 1963)\,, \\
(\mathit{eg{:}results\_Oct1961}\ \mathit{rdfs{:}comment}\ \mathit{eg{:}player\_stats})\,, \\
\qquad \bigvee b, y. \left(\begin{array}{ll} & \bigvee a. \left(\begin{array}{l} |(a\ \mathit{eg{:}month}\ 10)| \\ |(a\ \mathit{eg{:}year}\ 1963)| \\ |(a\ \mathit{rdfs{:}comment}\ \mathit{eg{:}player\_stats})| \\ |\mathcal{G}_a(b\ \mathit{p{:}goals}\ y)| \end{array} \right) \\ & \bigvee x, z. \left(\begin{array}{l} (b\ \mathit{p{:}goals}\ x)^{\perp} \\ (b\ \mathit{p{:}goals}\ z) \\ (z = x + y) \end{array} \right) \end{array} \right) \\
\qquad\qquad \rhd\ (\mathit{Armstrong}\ \mathit{p{:}goals}\ 11)\,, \\
\qquad\qquad\quad \mathcal{G}_{\mathit{eg{:}results\_Oct1961}}(\mathit{Armstrong}\ \mathit{p{:}goals}\ 2)\,, \\
\qquad\qquad\quad (\mathit{eg{:}results\_Oct1961}\ \mathit{eg{:}month}\ 10)\,, \\
\qquad\qquad\quad (\mathit{eg{:}results\_Oct1961}\ \mathit{eg{:}year}\ 1963)\,, \\
\qquad\qquad\quad (\mathit{eg{:}results\_Oct1961}\ \mathit{rdfs{:}comment}\ \mathit{eg{:}player\_stats})
\end{array}
$$

Thus the language is easily extended to handle features for provenance. The provenance of data is particularly significant in modern applications [38]. Thus provenance is given special consideration in this chapter.

3.2 The Core Syntax and Semantics

A new calculus for Linked Data is considered. The focus of the calculus is the key standards for Linked Data, introduced in the previous chapter. Primarily, the core of the semi-structured data format RDF and the core of the SPARQL Update language are captured. The approach is that of structural operational semantics. An abstract syntax is defined, then the operational semantics are defined by a deductive system which derives relations over the abstract syntax. The operational semantics specifies the runtime behaviour of a high level programming language.

The rôle of the abstract syntax differs from the rôle of common concrete syntaxes for RDF [38]. A concrete syntax is intended for human readability or message exchange. In contrast, the abstract syntax is for the purpose of compiler engineering. It captures the essence of the languages, with minimal redundancies. The symbols for connectives are chosen to highlight connections with connectives in related systems. Brief examples of the concrete syntax are provided, then the abstract syntax is fully defined.

3.2.1 A Syntax for the Resource Description Framework

The Web is based on documents, represented by one URI, which link to other documents, represented by another URI. The link structure of the Web can therefore be represented by pairs of URIs. This link structure has been exploited by organisations such as Google [77]. The source and target URIs are the subject and the object of the link, respectively.

RDF extends the link structure of the Web to the power of simple sentences. In natural languages, a verb indicates how a subject is related to an object. In English for instance the structure of a simple sentence is subject–verb–object e.g. "Kleinberg wrote Authoritative Sources in a Hyperlinked Environment." RDF extends the link structure of the Web to include a predicate. The predicate serves the same rôle as a verb, by indicating the nature of the connection between a subject and an object.

Like the subject and the object, the predicate is also a URI. A URI is a standardised global identifier for any resource, so need not identify a document. Thus the URI of a predicate is just a global identifier from some vocabulary. Similarly, the URI of the subject and the object need not refer to a document. Instead the URI could provide a global reference to some resource which, in a traditional setting, would normally be a local identifier in a database. The following is an example of two triples.

```
soton:9724 foaf:knows soton:10511 .
eprint:21769 dc:creator soton:10511 .
```

Note that *soton: eprint: foaf:* and *dc:* represent URI prefixes `http://id.ecs.soton.ac.uk/person/`, `http://eprints.ecs.soton.ac.uk/id/eprint/`, `http://xmlns.com/foaf/0.1/` and `http://purl.org/dc/elements/1.1/` respectively. The first and second are namespaces used for people affiliated with Southampton University and publications on EPrints. The third and fourth are namespaces used for terminology in the Friend-of-a-Friend and Dublin Core metadata vocabularies.

Another notion generalised by RDF is the idea that a URI is associated with a document on the Web. RDF allows several pieces of traditional data to be associated with a URI. As with links between resources a predicate indicates how a URI is related to a piece of traditional data. Again this resembles simple sentences in natural language. The following is an example of two triples.

$$
\begin{array}{rcll}
o & ::= & v & \text{literal} \\
 & | & x & \text{variable} \\
 & | & a & \text{name}
\end{array}
\qquad\qquad
C ::= (a\ a\ o) \quad \text{triple}
$$

FIGURE 3.1: A syntax for objects and RDF content.

```
soton:doc1 dc:title "Tae a Link" .
soton:doc1 dc:description "Some poem." .
```

The examples above are written using the Turtle syntax for RDF. Turtle is one of several concrete languages for presenting RDF. Here an abstract syntax captures the essence of these formats, without redundancy.

3.2.1.1 An Abstract Syntax for RDF triples.

For the purpose of defining operational semantics, an abstract syntax for RDF triples is defined. The abstract syntax captures the idea that RDF consists of triples of the form subject–predicate–object, as presented in Fig. 3.1.

The atoms of the syntax are names, variables and literals. Names represent URIs. For simplicity of examples, names are presented in italics, e.g. a, b, *person:9724*, *foaf:knows*. Names bound by quantifiers represent place holders for URIs. Literals represent traditional data such as a string of characters or an integer, such as `'Authoritative Sources'` or `'7'`. Traditional data is well understood, so the technicalities of literals are left to the XML datatypes specification [25]. Variables are explicit place holders for literals.

A triple consists of a subject, a predicate and an object. The subject and predicate are names. The object is either a name, a literal or a variable. A URI as an object generalises the notion of a link between Web pages. Similarly, the use of a literal as an object generalises the notion of the document associated with a link. The following demonstrates two triples, the first with a URI object the second with a literal object.

(*doc1 creator Burns*) (*doc1 title* `'Links'`)

Note that the W3C recommendation describes how to obtain labelled directed graphs from the syntax of RDF [78]. This provides a denotational semantics, which is used by graph query languages [110]. In contrast, this work is purely syntactic. Denotational semantics for concurrency are notoriously difficult [4].

3.2.2 A Syntax and Semantics for Queries and Updates

When data is published openly it is rarely possible to predict how it might be used. It is therefore difficult to decide a suitable format in which to publish the data. Preferably, the application which consumes the data should decide. For this reason RDF is a simple semi-structured data format. Power is regained from this minimal structure by an expressive query language. The query language enables the consumption of emergent structures conveyed in RDF. In this way the power is shifted from the producer to the consumer and lowers barriers to publishing Linked Data.

The language SPARQL Query is the agreed standard for the purpose of querying RDF. The first SPARQL Query standard has been widely deployed [115, 28]. A second draft for SPARQL Query learns from the experiences of the first [66]. A SPARQL end point is used to observe an RDF store. The observer declares the link patterns of interest using SPARQL Query. The query language also determines the format in which results are presented.

For instance an application may be interested the question, "Obtain names for either products related to the show or products related to an exhibitor at the show." The example scenario can be specified as follows in the SPARQL Query concrete syntax, where `eg:show2011` identifies the show and `eg:exhibitor` and `eg:product` identify predicates from some vocabulary.

```
SELECT ?product WHERE {
 {
  eg:show2011 eg:exhibitor ?exhibitor .
  ?exhibitor eg:product ?product
 }
 UNION
 { eg:show2011 eg:product ?product }
}
```

An analogy is that queries support compound sentences although RDF only supports simple sentences. Several simple sentences may be required to verify the truth of a compound sentence. The returned result is witness to the veracity of the compound sentence in the given context. While the truth of RDF is subjective, the truth represented by a successful SPARQL Query is intersubjective [56]. For intersubjective truth there is an subjective agreement between multiple parties. The parties involved are the client that poses the query (compound sentence) and the providers of the triples (simple sentences).

Current SPARQL recommendations have no constructs for maintenance. However, there is a proposal by Hewlett-Packard Labs and a working draft for a language called SPARQL Update [126, 53]. The proposals allow RDF to be inserted and deleted at the level of triples. Update operations reuse the operations of SPARQL Query for powerful updates.

$$
\begin{array}{rlr}
\phi ::= & \mathrm{I} & \text{true} \\
| & 0 & \text{false} \\
| & \phi \wedge \phi & \text{and} \\
| & \phi \vee \phi & \text{or} \\
| & \neg\phi & \text{not} \\
| & \ldots & \text{etc.}
\end{array}
\qquad
\begin{array}{rlr}
U ::= & |C| & \text{ask} \\
| & C^{\perp} & \text{delete} \\
| & C & \text{insert} \\
| & \phi & \text{filter} \\
| & U\,;P & \text{then} \\
| & U \oplus U & \text{choice} \\
| & U \otimes U & \text{tensor} \\
| & *U & \text{iteration} \\
| & \bigvee a.U & \text{select name} \\
| & \bigvee x.U & \text{select literal}
\end{array}
\qquad
\begin{array}{rlr}
P ::= & \perp & \text{nothing} \\
| & P \parr P & \text{par} \\
| & \bigwedge a.P & \text{blank node} \\
| & U & \text{update}
\end{array}
$$

FIGURE 3.2: A syntax for constraints, updates and processes.

The example above can be extended to specify the update, "For either products related to the show or products related to an exhibitor at the show, insert a link from another show to that product." The extension is to add one clause to the query. The clause inserts a triple which relates the show to the discovered product.

A model for SPARQL Update is sufficient to model SPARQL Query. However, a model for SPARQL Update is more subtle than a model for SPARQL Query. Not only is the truth represented by a successful intersubjective interaction, it is also dependent on time.

3.2.2.1 An abstract syntax for updates.

An update is a declarative specification of the intention of the programmer. The meaning of an update is independent of a particular implementation. An application expects some behaviour and an implementation provides a behaviour within the bounds of expectation. The common interface between the application and the store is the syntax of the language. An abstract syntax for Updates is provided in Fig. 3.2.

Basic queries are formed by embedding the syntax of RDF. The embedding '*ask*' is used to demand that some RDF is matched. This models asking a query, which has no side effects. The following is an example of asking a query which is satisfied by the example RDF in Section 3.2.1.1.

$$|(\textit{doc1 creator Burns})| \otimes |(\textit{doc1 title } \texttt{'Links'})|$$

Basic updates are also formed by embedding the syntax of RDF in two ways. The embedding '*delete*' demands that some persistently stored RDF should be removed. The embedding '*insert*' stores some RDF persistently. Unlike queries, both delete and insert have side effects. For instance the above RDF could be inserted into a store then removed from the store.

Update can be formed using two binary operations, '*tensor*' and '*choose*'. The tensor product is the operation which combines two updates to ensure that they happen in the same atomic

commitment. For instance, a query and an insert can be combined using the tensor product to ensure that an insert occurs if and only if the query is satisfied. Choose presents an option where either the left update or the right update occurs. For instance, a choice can be presented between two possible query patterns. The iteration operation indicates that zero, one, two or more copies of an update are simultaneously applied.

In Chapter 4 it is proven that the constructs choose, tensor, iterate, true and false form a Kleene algebra [41]. Hence syntactic conventions for Kleene algebras are adopted for examples. The operator $\otimes$ binds stronger than $\oplus$ and the operator $\otimes$ can be omitted; hence $(U \otimes V) \oplus W$ is abbreviated $UV \oplus W$. The following example presents a syntax for the update described in the previous section. The update is a extension of the query given in concrete syntax. Here the query is translated into the abstract syntax and an insert is composed with the query using tensor.

$$*\bigvee a.\bigvee b.\left(\begin{array}{l}\left(\begin{array}{l}|(show2011\ exhibitor\ b)|\\ |(b\ product\ a)|\\ \oplus\\ |(show2011\ product\ a)|\end{array}\right)\\ (show2012\ product\ a)\end{array}\right)$$

Updates extend Kleene algebras with quantifiers. The select quantifier binds occurrences of a name not known in advance. For instance, in the example above the name of the product is not known. The name is bound in both the query and the insert, so the name discovered by the query is also the name inserted. Names and literals are disjoint, so a separate quantifier is provided for variables. Quantifiers highlight the logical content of updates.

The syntax of constraints is embedded in the syntax of updates. A constraint imposes a condition on the update taking place. Typically variables which occur as the object of a triple are constrained. For instance a variable may represent a string of characters which satisfies a regular expression, or a numeral within a range of values. Like literals, constraints are well understood, so technicalities are left to the SPARQL Query standards [115, 66]. It is sufficient to note that constraints form a Boolean algebra.

3.2.2.2 A structural congruence for processes.

For the purpose of defining operational semantics, a syntax for processes is introduced in Fig. 3.2. Processes allow updates to be composed in parallel using the '*par*' operator to establish their concurrent behaviour. The unit of par is '*nothing*', which represents the empty process. Processes with the par operator and nothing unit form a commutative monoid, as defined by the structural congruence in Fig. 3.3. The convention, common to sequent style deductive systems, is that the symbol $\parr$ is abbreviated with a comma in examples.

A blank node is used in place of a URI when a URI is not explicily assigned. A blank node is a local identifier which cannot be linked to directly. Blank nodes reduce the barrier between

$$\text{Unit: } P \parr \bot \equiv P \qquad \text{Commutativity: } P \parr Q \equiv Q \parr P$$

$$\text{Associativity: } P \parr (Q \parr R) \equiv (P \parr Q) \parr R$$

$$\text{Eliminate quantifier: } \bigwedge a.\bot \equiv \bot \qquad \text{Swap quantifiers: } \bigwedge a.\bigwedge b.C \equiv \bigwedge b.\bigwedge a.C$$

$$\text{Distribute quantifiers: } \bigwedge a.P \parr Q \equiv \bigwedge a.(P \parr Q) \quad a \notin \mathrm{fn}(Q)$$

FIGURE 3.3: The structural congruence for processes.

RDF and other data formats, by allowing common data structures to be encoded in RDF without introducing new URIs. Blank nodes are indicated by a quantifier for names, similarly to N3 logic [23]. The scope of the quantifier indicates the RDF content in which a blank node is bound. Bound names represent blank nodes, whereas unbound names represent URIs.

The structural congruence is extended to blank node quantifiers in Fig. 3.3. The first rule allows blank nodes to be eliminated if they bind nothing. The second rule allows the order of two quantifiers to be swapped. The third rule allows a blank node to be distributed across some RDF content where the name does not occur. The blank node rules preserve the free URIs in RDF content.

As standard, bound names can be α-converted. This avoids name clashes between blank nodes. Content where only blank nodes differ are equivalent, by α-conversion. This is a syntactic approach to the graph isomorphisms defined in the RDF standards. Like α-conversion, the graph isomorphisms preserve the structure and URIs but allow the blank nodes to change [78, 13].

3.2.2.3 An operational semantics for atomic updates.

The behaviour of an update at the level of the syntax is captured by operational semantics. A preliminary draft of an operational semantics for SPARQL Update was published in October 2010 [53]. The operational semantics presented elaborates the draft. A fine grained operational semantics for updates are specified using atomic actions.

Atomic actions are specified as a relation over processes called the commitment relation. The process on the left of the relation is exactly the processes used by the action. The process on the right of the relation is exactly the processes after the action. Thus a commitment relation describes only the local behaviour of an update.

A similar approach is given by commitment relations in the π-calculus [98], discussed further in Sec. 3.4. In the π-calculus there is one type of commitment – the passing of a name on a channel. Coordination of Web Services motivates extending the commitments to include the

$$\text{Delete axiom: } \frac{C \sqsubseteq D}{C \parr D^{\perp} \rhd \bot} \qquad \text{Insert axiom: } C \rhd C \qquad \text{Query axiom: } \frac{C \sqsubseteq D}{|D| \parr C \rhd C}$$

$$\text{Filter axiom: } \frac{\vDash \phi}{\phi \rhd \bot} \qquad \text{Continuation rule: } \frac{P \parr U \rhd R}{P \parr (U \,;\, Q) \rhd R \parr Q}$$

$$\text{Choose left rule: } \frac{P \parr U \rhd Q}{P \parr (U \oplus V) \rhd Q} \qquad \text{Choose right rule: } \frac{P \parr V \rhd Q}{P \parr (U \oplus V) \rhd Q}$$

$$\text{Select name rule: } \frac{P \parr U\left\{{}^{b}/_{a}\right\} \rhd Q}{P \parr \bigvee a.U \rhd Q} \qquad \text{Select variable rule: } \frac{P \parr U\{{}^{v}/_{x}\} \rhd Q}{P \parr \bigvee x.U \rhd Q}$$

$$\text{Tensor rule: } \frac{P \parr U \rhd P' \quad Q \parr V \rhd Q'}{P \parr Q \parr (U \otimes V) \rhd P' \parr Q'} \qquad \text{Weakening axiom: } *U \rhd \bot$$

$$\text{Dereliction axiom: } \frac{P \parr U \rhd Q}{P \parr *U \rhd Q} \qquad \text{Contraction axiom: } \frac{P \parr (*U \otimes *U) \rhd Q}{P \parr *U \rhd Q}$$

$$\text{Context rule: } \frac{P \rhd P'}{P \parr Q \rhd P' \parr Q} \qquad \text{Blank node rule: } \frac{P \parr Q \rhd P' \parr Q'}{\bigwedge a.P \parr Q \rhd \bigwedge a.P' \parr Q'} \quad \begin{array}{l} a \notin \mathrm{fn}(Q) \\ a \notin \mathrm{fn}(Q') \end{array}$$

FIGURE 3.4: The axioms and rules for atomic commitments.

tensor product of channels [31]. SPARQL provides a compelling reason to extend commitments to all updates. The commitment relation $\rhd$ is defined in Fig. 3.4.

The delete axioms. A simple interaction is when an update deletes a triple and the triple is available to delete. For instance, the delete and triple below are expected to interact. The result of the interaction is that the delete and the matching triple are consumed. The following is an instance of the delete axiom.

$$(\mathit{doc1\ creator\ Burns})^{\perp}, (\mathit{doc1\ creator\ Burns}) \rhd \bot$$

The axioms of Linear Logic and the atomic commitments of CCS are of a similar form [55, 96], as discussed further in Sec. 3.4 and 3.5. Note the syntactic convention of using a comma for $\parr$.

The insert axiom and query axiom. RDF to be stored after an update appears on the right of a commitment relation. There are two ways in which RDF can appear on the right. The first scenario is that a triple is inserted into a store. This is captured by the insert axiom. The insert axiom states that some RDF intended to be stored is stored by a successful update.

The second scenario is that some stored RDF can be used to answer a query. The stored RDF then returned to the store unaltered. For instance, the following example consists of a stored

triple and a query asking for that triple. The query is answered and the triple remains stored.

$$\begin{matrix} |(doc1\ creator\ Burns)|\,, \\ (doc1\ creator\ Burns) \end{matrix} \rhd (doc1\ creator\ Burns)$$

The syntax for an insert is the same as the syntax for some stored RDF. Therefore a trivial update which inserts some RDF is used to model stored RDF. Other SPARQL Query results may be modelled similarly to inserts, by indicating the results on the right of the commitment relation. Related calculi investigate updates and queries over inserted data as concurrent constraint satisfaction problems for Web Services [36, 124].

The tensor rule. The tensor rule forces two updates to occur in the same commitment. The use of tensors meets a requirement of SPARQL Update that a delete and insert can occur atomically. Atomic actions which combine deletes and inserts avoid the need to reverse an insert, when a delete fails.

Another requirement met by the tensor product is that updates can be dependent on queries. The following example demonstrates the tensor product of an insert and a query. The available triple is adequate for the query, so the insert takes place. Both the stored triple and the inserted triple persist, so are composed after the transaction.

$$\left(\begin{matrix} |(doc1\ title\ \texttt{`Links'})| \\ (doc1\ creator\ Burns) \end{matrix}\right), (doc1\ title\ \texttt{`Links'}) \rhd \begin{matrix} (doc1\ creator\ Burns)\,, \\ (doc1\ title\ \texttt{`Links'}) \end{matrix}$$

The tensor rule splits a query into two updates which can be resolved in separate locations. For instance, in the above query the two parts of the tensor can be resolved on different machines in a cluster of servers. Thus, the tensor product serves the same purpose as join in relational algebra [67, 43]. The tensor rule also appears as the rule for multiplicative conjunction (times) in Linear Logic (discussed further in Sec. 3.5.4), and as atomic commitments in process calculi for Web Services [31].

The choose rules. Choice allows the programmer to specify several possible updates. The example below asks for a triple where the predicate is one of two options. The branch with the query which matches the available data is chosen. This is an external choice dependent on the available data.

$$\left(\begin{matrix} (doc1\ creator\ Burns)^{\perp} \\ \oplus \\ (Burns\ is\_author\_of\ doc1)^{\perp} \end{matrix}\right), (Burns\ is\_author\_of\ doc1) \rhd \perp$$

If both branches of a choice can be enabled, one is chosen non-deterministically. The choose rules correspond to the rules for additive disjunction (plus) of Linear Logic and an external choice in process calculi for Web Services [37], see Sec. 3.4 and 3.5.5.

The select rules. Most constructs work at the level of triples. Quantifiers are required to access names within triples. The select name rule works by substituting a name for the quantified name. For instance, in the example below the bound name a is replaced by *person*. This particular substitution allows the query to be answered and determines the name in the triple inserted.

$$\bigvee a.\left(\begin{array}{l}|(doc1\ creator\ a)|\\(Hamish\ knows\ a)\end{array}\right),(doc1\ creator\ Burns)\ \triangleright\ \begin{array}{l}(Hamish\ knows\ Burns),\\(doc1\ creator\ Burns)\end{array}$$

The effect above is that the substituted name is passed from the triple to the update. This is also the effect of the atomic commitments of the π-calculus [98]. The commitments of the π-calculus are decomposed into: a select which inputs the name; tensor which composes a continuation; and ask which poses a guard. By replacing triples with channel-value pairs and inserts with processes, the π-calculus can be recovered, as investigated by Miller [72] and in Sec. 3.4.1.4.

The select literal rule substitutes literals for variables. As above, this captures the passing of literals from triples to updates. Value passing is achieved by atomic commitments in the applied π-calculus [1]. The select rules match the rule for first-order existential quantification (some) in Linear Logic (see Sec. 3.5.5 for discussion).

The continuation rule. The continuation rule is used to provide a high level model of query results. In official specifications, the SPARQL Query Results Format is used to return the results of a query as an XML message. The message can then be parsed and used in a process. In this model, the passing of query results as a message is abstracted away. Instead a continuation process is provided which receives the results of a query directly as a substitution.

The continuation rule makes the guarded process available after the atomic commitment. For instance, consider the query below, which is adapted from the concrete query in Section 3.2.2.

$$\begin{array}{l}\bigvee a.\left(\bigvee b.\left(\begin{array}{l}|(show2011\ exhibitor\ b)|\\|(b\ product\ a)|\end{array}\right);P\right),\\(show2011\ exhibitor\ Penguin),\\(Penguin\ product\ collection)\end{array}\quad\triangleright\quad\begin{array}{l}P\{^{collection}/_{a}\},\\(show2011\ exhibitor\ Penguin),\\(Penguin\ product\ collection)\end{array}$$

In the above example P represents some continuation process in which the name a appears free. The name a is bound by a select quantifier, which also binds a name in the query. The result is that the value which is used to answer the query is also passed to the continuation. The second select quantifier does not bind the continuation process, hence is used to answer the query, but not to provide results.

The constraint satisfaction relation. In general constraints form a Boolean algebra. True formulae are indicated by $\vDash$ the constraint satisfaction relation. The definition of the constraint satisfaction relation is left to the SPARQL Query standards [115, 66]. For instance, the constraint below is satisfied when x is at least 20 years before the current year. Select substitutes x for `'1987'`, enabling the following commitment.

$$\bigvee a.\bigvee x.\left(\begin{array}{l}|(a\ \mathit{year}\ x)|\\ (\texttt{year-now} - x > 20)\\ (a\ \mathit{copyright}\ \mathit{open})\end{array}\right), (\mathit{paper}\ \mathit{year}\ 1987) \quad \triangleright \quad \begin{array}{l}(\mathit{paper}\ \mathit{year}\ 1987),\\ (\mathit{paper}\ \mathit{copyright}\ \mathit{open})\end{array}$$

An equality comparison over names is another form of constraint. The tensor product of an equality comparison and an update captures 'match' found in common process calculi [98, 1]. The constraints true and false are the top and bottom elements of the Boolean algebra. True always holds, so true is embedded as the multiplicative unit in Linear Logic (see Sec. 3.5.4 for discussion). False never holds, so like the additive zero in Linear Logic, no rule can be applied. The embedding of Boolean algebras in Kleene algebras is elaborated by Kozen [84].

The rules for iteration. Without iteration updates are only applied once. This enables a protocol where the programmer requests an update. The user then observes the commitment relation. If the update was not as the user intended, the update can be revoked. Then the next update is observed until the user is satisfied. Caution is exercised when the exact update is difficult to express or the content to update is not certain. When a user is certain that the update is intended, the update can be applied iteratively. The replace tool in the reader's text editor probably has similar functionality.

To apply an iterated update zero times, the weakening axiom is used. To apply an iterated update once, the dereliction rule is used. To apply an iterated update twice, the contraction rule creates two copies of the update combined using the tensor product. The tensor product ensures that both copies occur in the same commitment. The example below demonstrates two nested iterations. The outermost applies twice, the innermost applies both once and twice.

$$*\bigvee a.\left(\begin{array}{l}|(a\ \mathit{status}\ \mathit{hidden})|,\\ *\bigvee b.(a\ \mathit{knows}\ b)^{\perp}\end{array}\right), \begin{array}{l}(\mathit{Alice}\ \mathit{status}\ \mathit{hidden}),\\ (\mathit{Bob}\ \mathit{status}\ \mathit{hidden}),\\ (\mathit{Alice}\ \mathit{knows}\ \mathit{Bob}),\\ (\mathit{Alice}\ \mathit{knows}\ \mathit{Chris}),\\ (\mathit{Bob}\ \mathit{knows}\ \mathit{Chris})\end{array} \quad \triangleright \quad \begin{array}{l}(\mathit{Alice}\ \mathit{status}\ \mathit{hidden}),\\ (\mathit{Bob}\ \mathit{status}\ \mathit{hidden})\end{array}$$

Iteration is the Kleene star in regular algebra. A classic result is that nested iteration can be represented by a single iteration [41, 84]. However, quantifiers ensure that the example above cannot be expressed without nested iteration. The SPARQL Query recommendation does not have nested iteration, so cannot express the corresponding query [115].

Iteration is not replication in process calculi. Iteration defines a single commitment of an unbounded size; whereas replication persists a process across an unbounded number of commitments [98]. The use of contraction, dereliction and weakening is similar to the exponentials in Linear Logic (see Sec. 3.5.6), but does not correspond to either. Iteration has been used by Hoare and Kozen to specify unbounded behaviour, such as while loops [71, 84].

The context rule and blank node rule. To query and update blank nodes, the blank node rule is introduced. The trick is to replace a quantified name with a temporary free name. This allows the quantifier to be removed and the rules of the calculus to be applied as if there were no quantifiers. This makes the traditional process calculus equivalences for quantifiers superfluous; but they are included anyway to make the calculus more familiar to readers with a background in process calculus.

The temporary name is chosen to be fresh in the context. By choosing a fresh name, the name can be tracked before and after the commitments. This ensures that the same name that was quantified before is quantified after, as expressed by the first blank node rule. The blank node rule is similar to universal quantification in Linear Logic and new name quantification in the π-calculus [55, 98].

The example below demonstrates a query which discovers a blank node. The 'blank node' rule uses a temporary name to represent the blank node. The result is that the scope of the blank node quantifier is extended to include the continuation, which receives the blank node.

$$\begin{array}{l}\bigvee c.(|(c\ \mathit{creator}\ b_2)|\ ;\ P)\,,\\ \bigwedge a.((a\ \mathit{creator}\ b_2)\,,(a\ \mathit{status}\ \mathit{open}))\end{array} \rhd \bigwedge a.(P\{{}^{a}/_{c}\},(a\ \mathit{creator}\ b_2)\,,(a\ \mathit{status}\ \mathit{open}))$$

The unused stored triple is idled using the context rule.

Alias assumptions in queries. Working with aliases for URIs is a key problem in Linked Data [9]. Aliases arise since different data sources use different URIs for similar purposes. For instance, in the context of a song, predicate *lyricist* may be more specific than predicate *creator* (see subPropertyOf in RDFS [33]). Similarly, $song_0$ and $song_1$ may be URIs for the same song (see sameAs in OWL [9]). Hence the aliases $\mathit{lyricist} \sqsubseteq \mathit{creator}$ and $song_0 \sqsubseteq song_1$ may be assumed. The application specific set of alias assumptions is referred to as β. The transitive reflexive closure of β gives rise to a preorder ($\sqsubseteq$) over URIs, which extends point-wise to triples, as defined in Fig. 3.5.

The following example demonstrates the interaction of a query with a stored triple, where the names are not exact matches. The conditions for a match are relaxed by the preorder over triples. The alias assumptions for this example are those introduced above.

$$\bigvee a.\Big(\big|(song_1\ \mathit{creator}\ a)\big|\ ;\ P\Big)\,,\ \bigwedge b_4.(song_0\ \mathit{lyricist}\ b_4) \rhd \bigwedge b_4.\Big(P\big\{{}^{b_4}/_{a}\big\},(song_0\ \mathit{lyricist}\ b_4)\Big)$$

$$a \sqsubseteq a \qquad \frac{a \sqsubseteq b \quad b \sqsubseteq c}{a \sqsubseteq c} \qquad \frac{a \sqsubseteq b \in \beta}{a \sqsubseteq b} \qquad \frac{a \sqsubseteq b \quad p \sqsubseteq q \quad c \sqsubseteq d}{(a\ p\ c) \sqsubseteq (b\ q\ d)}$$

FIGURE 3.5: The preorder over URIs extended to triples: reflexivity, transitivity, alias assumption, and refine triple.

Note that a side condition must be added to the blank node rule to avoid aliases being applied to bound names. In this example b_4 cannot appear in the alias assumptions. i.e. $b_4 \notin \mathrm{fn}(\beta)$. This could alternatively be achieved, by using distinct identifiers for bound names and URIs.

3.3 Features for Syndication

The core calculus focusses on fine grained updates, where updates act at the level of individual triples. This section considers a coarser level of granularity. A coarser granularity of data divides a store into regions, where each region contains triples. Each region can be considered separately from other regions. Regions impact the querying of data by allowing queries to be directed at particular regions. Regions also enable a coarser level of update, where entire regions become atomic units.

This section argues that two key Web technologies work at the granularity of regions — feeds and named graphs [125, 38]. At a suitable level of abstraction, feeds and named graphs can be queried and updated in one model. The model demonstrates that several key standards for feeds and named graphs enable a programming language for Linked Data. Furthermore, prominent examples of feeds and named graphs suggest several useful scenarios, including syndication and provenance. Both syndication and provenance have been found to be essential for a Web of Data, where separate authorities contribute separate data.

3.3.1 Extensions for named graphs

Named graphs are introduced as a minimal extension to RDF such that a large monolithic knowledge base, consisting of a single RDF store can be divided into smaller stores, each individually named [38]. The name of the graph is a URI, which can be liked to like any other URI. The RDF associated with the name of a graph, can describe the nature of the knowledge represented by the graph. Applications including provenance and access control have lead to the widespread acceptance of named graphs. Named graphs are primitive in SPARQL Query and SPARQL Update [66, 53].

The following is an example of two named graphs. The example is expressed in the TriG syntax, an extension of the Turtle syntax [38]. The first graph contains some RDF data. The second graph contains some RDF data about the first graph. This enables the user make decisions based

$$
\begin{array}{rcll}
G & ::= & \mathcal{G}_a P & \text{named graph} \\
& | & P & \text{default graph} \\
& | & G \parr G & \text{par} \\
& | & \bot & \text{nothing}
\end{array}
$$

FIGURE 3.6: An extended abstract syntax for named graphs.

on the source of the RDF. The user may trust RDF with provenance *eg:G1* and use that directly, ignoring data in *eg:G2*. Alternatively, the user may trust RDF with provenance *eg:G2* on the subject of whether to use data in *eg:G1*.

```
eg:G1 {
  _:Monica eg:name "Monica Murphy" .
  _:Monica eg:email <monica@murphy.org> .
}
eg:G2 {
  eg:G1 eg:author eg:Chris .
  eg:G1 eg:date "2003-09-03"^^xsd:date
  eg:G1 eg:disallowedUsage eg:Marketing.
}
```

The above example features URIs and blank nodes. Identifiers with prefix `eg:` are URIs in some example namespace. The identifier with prefix `_:` represents a blank node. A constraint placed on named RDF graphs is that blank nodes are local to each individual named graph. This preserves the integrity of data structures encoded in named graphs.

3.3.2 An abstract syntax for named graphs

The abstract syntax of RDF content (from Fig. 3.1) is extended with named graphs. The syntax of graphs indicates both named graphs and unnamed RDF content. Named graphs are represented by a prefix with a subscript indicating the name, called the naming operator. RDF content without a naming operator represent the default graph, which allows RDF to be published without the named graph mechanism. The example from the previous section is expressed below in the abstract syntax. The blank node quantifier binds the name *Monica* in the first graph. However, the name *G1* is not bound by the naming operator so can be linked to from the second

$$\mathcal{G}_a(P \parr Q) \equiv \mathcal{G}_a P \parr \mathcal{G}_a Q$$

FIGURE 3.7: The structural congruence extended for naming operators.

graph.

$$\mathcal{G}_{G1}\left(\begin{array}{l} \bigwedge Monica. \\ \left(\begin{array}{l} (Monica\ name\ \texttt{`Monica Murphy'}), \\ (Monica\ email\ monica@murphy.org) \end{array}\right) \end{array}\right),$$
$$\mathcal{G}_{G2}\left(\begin{array}{l} (G1\ author\ Chris), \\ (G1\ date\ \text{`2003-09-03'}), \\ (G1\ disallowedUsage\ Marketing) \end{array}\right)$$

The above example shows that par, abbreviated by comma, is used as before to compose triples and also graphs. The structural congruence over RDF content, in Fig. 3.3, ensures that par and nothing form a commutative monoid. The structural congruence extends, in Fig. 3.7 to naming operators. A named graph can be split into two pieces each with the same name. This is for the purpose of fine grained updates, as only the part of a graph required for an update need be considered. The naming operator and the quantifiers do not commute, so blank nodes remain within their designated graph.

Related work constrains named graphs so that the boundaries of a named graph are fixed [38]. For instance, knowing the boundaries of a named graph enable the named graph to be completely dropped from a store, as in the drop operation of SPARQL Update [53]. This structural constraint is a perpendicular concern to this work. Similar constraint on global structure are tackled, for instance, in dynamic epistemic logic as structure preserving maps [14]. Stronger preservation of structure is extensively researched in the context of the Web using ontologies [74]. This work focusses on Linked Data without such a global perspective on structure.

3.3.3 SPARQL Update over named graphs

Queries and updates also work in the named graph setting. The abstract syntax for updates, in Fig. 3.2, is extended to named graphs, by replacing RDF triples (from Fig. 3.1) with quads, which are triples prefixed by a named graph modality. The same rules for atomic commitments (in Fig. 3.4) work for named graphs.

The SPARQL Update submission describes an update, where the title of a book is replaced by a new title [126]. This example is captured by the commitment relation below. The delete axiom removes a triple from a graph, the insert axiom inserts the new triple in to the graph and the tensor rule ensures the delete and the insert happen synchronously. The presence of the naming

$$\frac{\mathcal{G}_b P \parr Q \rhd \mathcal{G}_b P' \parr Q'}{\mathcal{G}_b \bigwedge a.P \parr Q \rhd \mathcal{G}_b \bigwedge a.P' \parr Q'} \; a \notin \mathrm{fn}(Q, Q', b)$$

FIGURE 3.8: Commitments extended for blank nodes in named graphs.

prefix makes no difference to the rules.

$$\left(\begin{array}{l} \mathcal{G}_{store}(book_3 \; title \; \texttt{`Designs'})^{\perp} \\ \mathcal{G}_{store}(book_3 \; title \; \texttt{`Design'}) \end{array} \right), \; \rhd \mathcal{G}_{store}(book_3 \; title \; \texttt{`Design'})$$
$$\mathcal{G}_{store}(book_3 \; title \; \texttt{`Designs'})$$

The largest example given in initial SPARQL Update drafts can now be expressed [126]. The update combines a query which finds the date of a book, a filter which checks the date is in a certain range, and an iterated update on books in that range. The iterated update will only trigger if the query and filter are satisfied. The iterated update moves triples about a book across from one graph to another graph, by combining a delete and insert. The example can be expressed in the abstract syntax, as follows.

$$\begin{array}{l} *\bigvee d.\bigvee book. \\ \left(\begin{array}{l} (d \leq \text{`01-01-2000'}) \\ |\mathcal{G}_{store1}(book \; date \; d)| \\ *\bigvee a.\left(\begin{array}{l} \mathcal{G}_{store1}(book \; note \; a)^{\perp} \\ \mathcal{G}_{store2}(book \; note \; a) \end{array} \right) \end{array} \right), \\ \mathcal{G}_{store1}\left(\begin{array}{l} (Kidnapped \; date \; \texttt{`01-05-1886'}), \\ (Kidnapped \; note \; classic) \end{array} \right) \end{array} \quad \rhd \quad \begin{array}{l} \mathcal{G}_{store1}(Kidnapped \; date \; \texttt{`01-05-1886'}), \\ \mathcal{G}_{store2}(Kidnapped \; note \; classic) \end{array}$$

The above example is compact compared to the example in the draft concrete syntax. The presentation here is enabled by the constructs in the abstract syntax, whereas a single compound construct is used in the concrete syntax.

3.3.4 Updates for named graphs with blank nodes

The second blank node rule ensures that a blank node which originates in an named graph is returned to the same named graph. The extra mix rule allows triples in the scope of a blank node which are not used in a commitment to idle.

The following example demonstrates an update which involves a blank node in a named graph. The name of the blank node in the graph is replaced by a temporary name. The temporary name is discovered by the select quantifier as normal. A new triple with the temporary name is inserted into the same graph as normal. The blank node rule ensures that the temporary is bound after

the commitment.

$$\bigvee person.\bigvee x.\bigvee y.\left(\begin{array}{l}\left|\mathcal{G}_{graph1}(person\ nickname\ x)\right| \\ \mathcal{G}_{graph1}(person\ email\ y) \\ (y = \texttt{concat}(x, \texttt{`@soton.ac.uk'}))\end{array}\right), \mathcal{G}_{graph1} \bigwedge a.(a\ nickname\ \texttt{`Rabbie'})$$

$$\triangleright\ \mathcal{G}_{graph1}\ \bigwedge a.\left(\begin{array}{l}(a\ nickname\ \texttt{`Rabbie'}), \\ (a\ email\ \texttt{`Rabbie@soton.ac.uk'})\end{array}\right)$$

In the example the blank node does not leave the graph. Suppose that instead the update inserts the new triple into a different named graph. In this case the update cannot be applied since the blank node would appear free in another graph. The side condition would be violated.

3.3.5 Feeds as a ubiquitous syndication format

RDF is the format standardised by the W3C, however feeds are ubiquitous on the Web. Like RDF, feeds are a semi-structured data format which identifies resources using URIs. The two ubiquitous feed formats are RSS and Atom. RSS was originally created by Netscape and comes in several varieties. Atom has the same purpose as RSS but is standardised [125, 104]. Atom has been adopted by Google for its Google Data protocol, which shares data between applications.

Feeds are particularly suited to syndication. Syndication is the strategy of delivering data to the intended audience on demand. Feeds typically represent the view point of some authority. A BBC News feed on Africa contains data representing the viewpoint of the BBC on the topic of news in Africa. A user who is interested in that viewpoint can obtain that feed on demand. The user can answer questions such as, "According to BBC News on Africa, what are the headlines today?"

3.3.5.1 A history of feeds.

The history of feeds highlights the close connection between feeds and named graphs. Feeds were initially introduced for structuring the metadata about Web sites. The first significant version was introduced by Netscape in 1999. There have been two major branches of the RSS format. The first branch explicitly uses and RDF format where as the second branch uses a more direct XML format. A lot of fuss was made about the technical incompatibilities of the various formats. However, for the purpose of this work the idea behind all formats is the same. It is presumed that most technical difficulties can be resolved by the compiler in a high level language.

A coherent line of thinking behind feeds can be seen in the work of Guha [62]. His thesis draws from Cyc project which was an early attempt at computing using knowledge. His thesis acknowledges some problems associated with a project which attempts to assemble a single knowledge base of everything that is common sense. The problems highlighted, which are as old as philosophy, is that knowledge and the language used to describe knowledge, is subjective. There cannot be one general purpose knowledge base hence. Indeed the idea of a common sense is an objective myth. Guha's thesis builds a theory of contexts, where any representation of knowledge also indicates the context of the given knowledge.

According to Guha, a context structure indicates the context in which some knowledge holds. He gives the example sentence, "The king of France is bald." The sentence has little relevance in the context of modern politics, but is relevant in the context of a play about the French revolution. Guha then produces a logical theory of knowledge in contexts. The idea of the meaning of language being dependent on context is central to the Language Games of Wittgenstein which were later modelled by Hintikka [133, 69]. However the work of Guha is distinguished for its rôle in computer science. The necessity of contexts was driven by an application which is a precursor to the Web of Data. In his capacity at Netscape Guha applied the idea in an accessible form by inventing RSS feeds.

The later work of Guha re-evaluates contexts for their rôle in the Web of Data [63]. This re-evaluation is performed as part of the TAP project, under the slogan "Towards a Web of Data." At this point the language of feeds and the language of the Web of Data fold into one. Feeds are data on the Web. Feeds were inspired by contexts which were shown to be necessary by experience in early projects similar to the Web of Data project. The result is that a ubiquitous form of data on the Web can be found in feeds [60]. This work therefore argues that feeds are part of the Web of Data.

3.3.5.2 An example feed.

The following is an example of the Atom Syndication Format. Notice that the feed and the entry are identified by URIs, which are abbreviated here as *eg:feed_id* and *eg:entry_id*. The tags such as *title* and *updated* are also URIs indicated by the XML namespace.

```
<feed xmlns="http://www.w3.org/2005/Atom">
 <title>Example Feed</title>
 <link href="http://example.org/"/>
 <updated>2003-12-13</updated>
 <author>
  <name>John Doe</name>
 </author>
 <id>eg:feed_id</id>
 <entry>
```

```
  <title>Example Entry</title>
  <link href="http://example.org/03"/>
  <id>eg:entry_id</id>
  <updated>2003-12-13</updated>
  <summary>Some text.</summary>
 </entry>
</feed>
```

The above example can be represented using named graphs and blank nodes as follows. The entries are translated into triples and form the content of a named graph. The triples associated with the feed are part of the default graph. The XML style above does not indicate a URI for the author of the feed. Below the implicit author is represented by introducing a blank node.

$$
\begin{array}{l}
\bigwedge Doe.\left(\begin{array}{l}(Doe\ name\ \texttt{`John Doe'}),\\ (feed\_id\ author\ Doe)\end{array}\right),\\
(feed\_id\ title\ \texttt{`Example Feed'}),\\
(feed\_id\ updated\ \texttt{`2003-12-13'}),\\
(feed\_id\ link\ http://example.org/),\\
\mathcal{G}_{feed\_id}\left(\begin{array}{l}(entry\_id\ title\ \texttt{`Example Entry'}),\\ (entry\_id\ link\ http://example.org/03),\\ (entry\_id\ updated\ \texttt{`2003-12-13'}),\\ (entry\_id\ summary\ \texttt{`Some text.'})\end{array}\right)
\end{array}
$$

The above syntax demonstrates one representation of Atom in RDF, however there is no standard representation. Some varieties of RSS encode feeds using triples. However, named graphs are primitive in SPARQL, so are suggested here as a representation of the content of a feed.

3.3.6 The Atom Publishing Protocol

For RSS an application implements its own update mechanism. In contrast, the Atom Publishing Protocol extends Atom with a standard update mechanism [59]. The publishing protocol allows new resources to be published and existing resources to be edited. The protocol works at the low level of passing messages using an HTTP protocol. However, feeds can still be updated at the high level offered by SPARQL. This section demonstrates a high level update of a feed and outlines the corresponding low level operations which realise the high level update.

The Atom publishing protocol specification allows variations on the basic protocol. The example in this section features a main feed of articles and comment feed linked to each entry of an article. Firstly, a feed is declared such that initially it contains no resources. The data associated with

the feed indicates the author of the feed and a title for the feed.

$$\begin{array}{l}(\textit{feed author Hamish}),\\ (\textit{feed title } \texttt{`Caucasus reported'}),\\ \mathcal{G}_{\textit{feed}}\bot\end{array}$$

The first update, defined below, creates a new article in the feed and an empty comment feed to go with the article. The comment feed is linked to the new article. The triple associated with *entry* indicates a title and a modification date.

$$\begin{array}{l}\mathcal{G}_{\textit{feed}}\left(\begin{array}{l}(\textit{entry title } \texttt{`Invaded'})\\ (\textit{entry updated } \text{`01-02-2008'})\\ (\textit{entry comments discussion})\end{array}\right)\\ (\textit{discussion subject entry})\\ \mathcal{G}_{\textit{discussion}}\bot\end{array}$$

A second update, defined below, changes the title and the date the feed was updated. The update first discovers the old title and old date using select quantifiers. The update then deletes the old triples and inserts the new triples.

$$\begin{array}{l}\bigvee s,d.\mathcal{G}_{\textit{feed}}\left(\begin{array}{l}(\textit{entry title } s),\\ (\textit{entry updated } d)\end{array}\right)^{\bot}\\ \mathcal{G}_{\textit{feed}}\left(\begin{array}{l}(\textit{entry title } \texttt{`Ossetia invaded'}),\\ (\textit{entry updated } \text{`02-04-2008'})\end{array}\right)\end{array}$$

A third update creates a new comment in the comment feed associate with the entry. A query discovers the relevant comment feed and a new entry is inserted in that comment feed. The new comment is identified by a blank node rather than a URI.

$$\begin{array}{l}\bigvee \textit{discussion}.\left|\mathcal{G}_{\textit{feed}}(\textit{entry comments discussion})\right|\\ \qquad\bigwedge \textit{reaction}.\\ \mathcal{G}_{\textit{discussion}}\left(\begin{array}{l}(\textit{reaction content } \texttt{`Why?'}),\\ (\textit{reaction author Dmitri}),\\ (\textit{reaction updated } \text{`05-04-2008'})\end{array}\right)\end{array}$$

The updates can be applied to the initial configuration. By applying the operational semantics the results is the configuration bellow.

$$
\begin{array}{l}
(\textit{feed author Hamish}),\\
(\textit{feed title } \texttt{`Caucasus reported'}),\\
\mathcal{G}_{\textit{feed}}\left(\begin{array}{l}(\textit{entry title } \texttt{`Ossetia invaded'}),\\ (\textit{entry updated } \text{`02-04-2008'})\\ (\textit{entry comments discussion}),\end{array}\right),\\
\qquad \bigwedge \textit{reaction}.\\
\mathcal{G}_{\textit{discussion}}\left(\begin{array}{l}(\textit{reaction content } \texttt{`Why?'}),\\ (\textit{reaction author Dmitri}),\\ (\textit{reaction updated } \text{`05-04-2008'})\end{array}\right)
\end{array}
$$

This example demonstrates that the same language for updating named graphs can be used to update feeds. The underlying operations of a publishing protocol can realise these updates. Operations of the protocol are described by the verbs post, put and get as found in a REST protocol [48].

The underlying REST operations can be outlined as follows. The first update corresponds to posting an entry to the feed and posting a new feed to the store. The second update corresponds to getting the entry and putting it back in its updated form. The third entry corresponds to getting the entry, evaluating a query and posting a new entry in the comment feed.

This example demonstrates that other semi-structured data formats, such as Atom, are compatible with RDF. It also demonstrates that SPARQL Update can be realised by the operations of a lower level protocol. The details of the low level protocol are hidden from the programmer. Related work demonstrates high level operations encoded using low level operations in the setting of Web Services [31].

3.4 A Comparison to Established Process Calculi

Concurrency can be modelled in many ways. One of the most prominent methods is to use a process calculus. Process calculi have the advantage that they are defined syntactically using techniques accessible to software engineers. This contrast to denotational models of concurrency which tend to employ non-standard mathematics which is not universally understood.

The calculus in this work is not an extension of any existing calculus known to the author. It is however related to many existing process calculi. Here the π-calculus is introduced as an example of an established process calculus.

$U :=$	$\overline{a}a.P$	output action	$P :=$	U	guarded process
$\mid$	$aa.P$	input action	$\mid$	$P \parallel P$	parallel composition
$\mid$	$a(x).P$	bounded input action	$\mid$	0	the empty process
$\mid$	$\tau.P$	silent action	$\mid$	$va.P$	fresh name restriction
$\mid$	$U \oplus U$	choice			

FIGURE 3.9: The syntax of guarded processes (U) and processes (P).

The π-calculus also serves as a reference point to further justify the calculus by demonstrating that the new calculus introduced is more expressive than the π-calculus. The rules of the π-calculus decompose into the same basic rules as the syndication calculus, but in a restricted form. It is therefore necessary to introduce the syndication calculus as a new calculus, rather than encoding the syndication using existing calculi.

3.4.1 An established process calculus

> The π-calculus cannot claim to be definitive; in fact, since it was designed it has become common to express ideas about interaction and mobility in variants of the calculus. So it has become a kind of workshop of ideas.
>
> Robin Milner 2001 [123]

For the reader who is less familiar with concurrency theory the most famous calculus is presented. The π-calculus has received a huge amount of attention since it was introduced by Milner, Parrow and Walker [99]. The π-calculus was the first concise model of concurrency with communication which is Turing complete.

Many variants of the syntax have been investigated, including subtle changes to the syntax and extensions. More significantly, many variation on the semantics of calculus have been investigated. A reduction semantics provides the most concise definition, which is compared here to the reduction system of the syndication calculus. An alternative semantics for the π-calculus, a labelled transition system which enables novel proof techniques, will be presented in the next chapter.

3.4.1.1 A syntax for the π-calculus.

The syntax for the π-calculus is defined in terms of guarded processes and processes, as defined in Fig. 3.4.1.1. The basic atoms of the syntax are names, which represent both channels on which communication takes place and variables. Processes and guarded processes are mutually recursively defined. This allows processes to appear as continuations to guarded processes.

Three actions are used to guard processes in the syntax. The actions are either the output action, the input action or the silent action. A guarded process represents the intention to perform an action, then proceed with the continuation process.

Input and output Actions. The output action represents the intention to output a name on a channel. This is represented as a pair of names, where the first name is the channel and the second name is passed on the channel. For instance, the following process is ready to send the name b on channel a, and then the name c on channel b. The process P is the continuation after both actions.

$$\overline{a}b.\overline{b}c.P$$

Input actions represent the intention to receive a name on a channel. This is represented as a pair consisting of a name and a variable. The name represents the channel and the variable is a place holder for any name that is to be received. The variable binds occurrences of the variable in the continuation process.

For instance, the following process first receives a name on channel a, where the input replaces the variable x. The name received is then used as a channel to send a name b. The name b is then used to receive a further name. Both names received may appear in the continuation process.

$$a(x).\overline{x}b.b(y).P$$

Choice and silent actions. A choice can be offered between two guarded processes. Only one of the branches may proceed, where the chosen branch is determined by the action. For instance, the following process provides two options. The first branch outputs the name b on channel a and proceeds with process P. The second branch outputs the name c on channel a and proceeds with Q.

$$\overline{a}b.P \oplus \overline{a}c.Q$$

The silent action τ represents the ability to proceed autonomously. This action is silent because no input or output need be observed for the action to occur. For instance, the following process has the option of two silent actions. Because both silent actions are enabled, the process can non-deterministically proceed with one of the continuations presented. This example is often referred to as internal choice, since the reason for the choice of branch is unobservable.

$$\tau.P \oplus \tau.Q$$

Parallel composition. Processes are built from guarded processes and the empty process. The empty process indicates a terminated process. Parallel composition of processes allows two concurrent processes to be expressed. Parallel composed processes can behave independently or choose to interact when complementary actions are performed. For instance, the following

$$P \parallel Q \equiv Q \parallel P \qquad (P \parallel Q) \parallel R \equiv P \parallel (Q \parallel R) \qquad P \parallel 0 \equiv P$$

$$\nu a.(P \parallel Q) \equiv \nu a.P \parallel Q \quad a \notin \mathrm{fn}(Q) \qquad \nu a.\nu b.P \equiv \nu b.\nu a.P \qquad \nu a.0 \equiv 0$$

FIGURE 3.10: The structural congruence over processes.

process consists of two guarded processes composed in parallel. The two processes exhibit complementary actions so are capable of interacting with each other.

$$a(x).\overline{x}b.b(y).P \parallel \overline{a}c.c(z).\overline{z}d.0$$

Local names. The fresh name quantifier binds occurrences of a name in a process. The fresh name quantifier guarantees that the name only occurs in the scope of the quantifier. Any occurrences of the same name outside the scope of the quantifier are distinct to occurrences of the name within the quantifier. For instance, the two processes below each have a fresh name, which is not known to any other process. In contrast, the name a is global so can be used for interaction, as in the example above.

$$\nu b.\,(a(x).\overline{x}b.P) \parallel \nu c.(\overline{a}c.c(z).0)$$

In the example above, notice that the name c is restricted in the second process and also output on the channel a. This allows the name local to the first process to be communicated to the second process. To allow this communication the scope of the name c must be extended to the first process. To understand how this process behaves the operational semantics are required, which are defined in the next two sections.

3.4.1.2 A structural congruence for the π-calculus.

The reduction semantics for the π-calculus uses a structural congruence. A structural congruence is a relation over processes which indicates processes which are regarded as equivalent. The structural congruence assumed here is presented in Figure 3.10. A congruence is an equivalence relation which can be applied in any context.

Parallel composition forms a commutative monoid with the empty process as a unit. This allows processes to be reordered and the empty processes to be eliminated. This simplifies rules by allowing processes which interact to be positioned next to each other in the necessary order. For instance, in the process below the guarded processes which communicate on channel a are positioned next to each other.

$$\overline{a}b.0 \parallel b(y).Q \parallel a(x).P \equiv \overline{a}b.0 \parallel a(x).P \parallel b(y).Q$$

$$\tau.P \longrightarrow P \qquad \overline{a}b.P \parallel ab.Q \longrightarrow P \parallel Q \qquad \frac{P \parallel ab.Q\{^{b}/_{x}\} \longrightarrow R}{P \parallel a(x).Q \longrightarrow R}$$

$$\frac{P \parallel U \longrightarrow Q}{P \parallel (U \oplus V) \longrightarrow Q} \qquad \frac{P \parallel V \longrightarrow Q}{P \parallel (U \oplus V) \longrightarrow Q} \qquad \frac{P \longrightarrow Q}{P \parallel R \longrightarrow Q \parallel R} \qquad \frac{P \longrightarrow Q}{\nu a.P \longrightarrow \nu a.Q}$$

FIGURE 3.11: A reduction system for the π-calculus: The τ rule, the communication rule, structure rule, the par context rule and the fresh name context rule.

The standard rule of alpha conversion can always be applied to names bound by the fresh name quantifier. Alpha conversion allows name clashes to be avoided. The fresh name quantifier is also allowed to dynamically change scope. The scope of the quantifier can change as long as the free names in a process do not change. This is performed by three rules. The distributivity rule for select quantifiers allows the scope of a name to distribute over a process in which the name does not occur. The commutativity of quantifiers allows the scope of one quantifier to be extended beyond the scope of another quantifier. The unit rule allows names quantified to be eliminate when they quantify nothing.

For instance, in the following configuration the fresh name b is alpha converted so that it does not clash with the b which appears free in the second guarded process. The name is then extended over the second process. This means that the input and output on channel a are adjacent.

$$\nu b.\overline{a}b.0 \parallel a(x).b(y).Q \equiv \nu c.(\overline{a}c.0 \parallel a(x).b(y).Q) \quad c \notin fnQ$$

The rules of the structural congruence can always be applied in rules wherever the parallel composition and fresh name connectives appear.

3.4.1.3 Reduction semantics for the π-calculus.

The operational semantics of the π-calculus can be defined using a reduction system [73]. The reduction system is express using relations over processes, written $P \longrightarrow Q$. The process on the left indicates the process before the reduction; while the process on the right indicates the process after the reduction.

The interaction rule. This version of the reduction system uses the rules presented in Figure 3.11. Conventionally, the choose rules, the select rule and communication rule are expressed as a single rule. This single rule is derived in the following Lemma, using the rules in the figure. This decomposition of the conventional communication rule into four simpler rules highlights the close connection between the π-calculus and the new calculi introduced in this work. In this work queries and updates are similarly decomposed to reveal essentially the same simple rules.

Lemma 3.1. $(\overline{a}b.P \oplus U) \parallel (a(x).Q \oplus V) \longrightarrow P \parallel Q\{^{b}/_{x}\}$

Proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\overline{a}b.P \parallel ab.Q\left\{{}^{b}/_{x}\right\} \longrightarrow P \parallel Q\left\{{}^{b}/_{x}\right\}}}{\overline{a}b.P \parallel a(x).Q \longrightarrow P \parallel Q\left\{{}^{b}/_{x}\right\}}}{\overline{a}b.P \parallel (a(x).Q \oplus V) \longrightarrow P \parallel Q\left\{{}^{b}/_{x}\right\}}}{(\overline{a}b.P \oplus U) \parallel (a(x).Q \oplus V) \longrightarrow P \parallel Q\left\{{}^{b}/_{x}\right\}}$$

□

The rule derived above defines the interaction between an input action and an output action which share the same channel. In the interaction both the input action and the output action are consumed. The name which is output is substituted for the variable in the continuation in the input process. Thus the passing of a name from the output process to the input process is modelled.

The following example of a process exhibits three interactions. Notice that two of the names passed are used as communication channels.

$$\begin{aligned} a(x).\overline{x}b.b(y).P \parallel \overline{a}c.c(z).\overline{z}d.0 &\longrightarrow \overline{c}b.b(y).P\{{}^{c}/_{x}\} \parallel c(z).\overline{z}d.0 \\ &\longrightarrow b(y).P\{{}^{c}/_{x}\} \parallel \overline{b}d.0 \\ &\longrightarrow P\{{}^{c}/_{x}\}\left\{{}^{d}/_{y}\right\} \parallel 0 \end{aligned}$$

The silent action. The τ rule allows a silent action to be consumed, without requiring any process to interact with. For instance, in the following transition the first branch is chosen, without external communication. Both branches were however enabled so the choice is non-deterministic.

$$\tau.P \oplus \tau.Q \longrightarrow P$$

Context rules. The other rules could be described as meta-properties of processes calculi in general. The structure rule allows the structural congruence to be applied to processes at any point in a derivation. As discussed in the previous section, the structural congruence allows the full range of interactions to take place. The two context rules allow an interaction to take place in any process context. The context rules and structural rule allows a process which consists of many concurrent processes to interact. For instance, the following example demonstrates three process interaction in two different combinations.

$$\begin{aligned} \nu c.(\overline{a}c.0) \parallel a(x).\overline{b}x.0 \parallel b(y).Q &\longrightarrow \nu c.(\overline{b}c.0) \parallel b(y).Q \\ &\longrightarrow \nu c.(Q\left\{{}^{c}/_{y}\right\}) \end{aligned}$$

This completes a description of the key ingredients of the π-calculus. The literature on the π-calculus is vast, so further examples can be readily obtained. The literature includes many extensions of the calculus to tackle various applications. For instance, recursive processes and replications of process allows long term behaviour to be described [99]. The calculus can be

extended to pass data as well as names, then applied to formalise security protocols [1]. More elaborate extensions include calculi to model locations and XML trees, which has been used to model Web applications [90]. Further relevant applications of the π-calculus are referred to throughout this work.

3.4.1.4 Combining the expressive power of calculi.

The π-calculus can be tightly integrated with the syndication calculus introduced in this work. A calculus more primitive than the π-calculus, such as the syndication calculus, is a powerful tool for understanding process calculi.

The extension of the syndication calculus to include communication on channels is simple. Firstly, extend the syntax of the syndication calculus to include channel–value pairs as well as triples. So the atoms of the syntax for updates, U in Fig. 3.2, include the atoms aa and $\overline{aa}$, where a is a name. The following axiom is then included, extending the reduction semantics in Fig. 3.4.

$$\overline{ab} \parr ab \rhd \bot$$

Notice that the above axiom has exactly the same form as the delete axiom. The only difference is that the atoms are name–value pairs instead of RDF triples.

The π-calculus can now be translated into the extended calculus. Define the lifting, from the syntax of Fig. 3.4.1.1 to the extended syndication calculus, as follows.

$$[\![a(b).P]\!] = \bigvee b.(ab \,;\, [\![P]\!]) \qquad [\![\overline{a}b.P]\!] = \overline{ab} \,;\, [\![P]\!] \qquad [\![\tau.P]\!] = \mathrm{I} \,;\, [\![P]\!]$$

$$[\![P \parallel Q]\!] = [\![P]\!] \parr [\![Q]\!] \qquad [\![P \oplus Q]\!] = [\![P]\!] \oplus [\![Q]\!] \qquad [\![0]\!] = \bot \qquad [\![\nu a.P]\!] = \bigvee a.[\![P]\!]$$

The following theorem proves that the above encoding of the π-calculus in the extended syndication calculus is correct. Both the π-calculus and its encoding have the same operational power.

Theorem 3.2. *$P \longrightarrow Q$ iff $[\![P]\!] \rhd [\![Q]\!]$, for π-calculus processes P, Q.*

Proof. Lemma 3.1 proves that the compound interaction rule can be decomposed into the rules provided in Fig. 3.11. Thus it sufficient to consider the reduction semantics for the π-calculus provided in Fig. 3.11.

The structural congruence carries over immediately, for π-calculus processes. Thus $P \equiv Q$ iff $[\![P]\!] \equiv [\![Q]\!]$, where the first structural congruence is in the π-calculus and the second is in the syndication calculus.

Consider the τ axiom.

$$\frac{\mathrm{I} \rhd \bot}{\mathrm{I} \,;\, [\![P]\!] \rhd [\![P]\!]} \qquad \text{iff} \qquad \tau.P \longrightarrow P$$

Consider the interaction axiom.

$$\dfrac{\dfrac{}{ab \parr \overline{ab} \rhd \bot}}{\dfrac{ab \parr (\overline{ab} \,;\, [\![Q]\!]) \rhd [\![Q]\!]}{(ab \,;\, [\![P]\!]) \parr (\overline{ab} \,;\, [\![Q]\!]) \rhd [\![P]\!] \parr [\![Q]\!]}} \qquad \text{iff} \qquad ab.P \parallel \overline{a}b.Q \longrightarrow P \parallel Q$$

Consider the select rule.

$$\dfrac{[\![P]\!] \parr ab \,;\, [\![Q]\!]\left\{{}^{b}/_{x}\right\} \rhd [\![R]\!]}{[\![P]\!] \parr \bigvee x.(ax \,;\, [\![Q]\!]) \rhd [\![R]\!]} \qquad \text{iff} \qquad \dfrac{P \parallel ab.Q\left\{{}^{b}/_{x}\right\} \longrightarrow R}{P \parallel a(x).Q \longrightarrow R}$$

The left and right choice rules and the context rule translate directly across. Thus, the remaining case to consider is the fresh name context rule. Assume that $a \notin \mathrm{fn}(Q) \cup \mathrm{fn}(S)$ and note that $\nu a.(P \parallel Q) \equiv \nu a.P \parallel Q$ and $\nu a.(R \parallel S) \equiv \nu a.R \parallel S$ hold.

$$\dfrac{\bigwedge a.[\![P]\!] \parr [\![Q]\!] \rhd [\![R]\!] \parr [\![S]\!]}{\bigwedge a.[\![P]\!] \parr [\![Q]\!] \rhd [\![R]\!] \parr [\![S]\!]} \qquad \text{iff} \qquad \dfrac{P \parallel Q \longrightarrow R \parallel S}{\nu a.(P \parallel Q) \longrightarrow \nu a.(R \parallel S)}$$

Thus, by structural induction, the π-calculus and the embedding of the π-calculus in the syndication calculus have the same expressive power. □

Notice that the last case also formalises an observation about the blank node rule. The proof shows that, in the presence of the rule of the structural congruence which distributes quantifiers over par, the form of the blank node context rule and the new name quantifiers are equivalent. The more complex form of the blank node context rule is only required if that distribution rule is dropped from the structural congruence. If the blank node distribution rule is dropped from the syndication calculus, then it can still be derived as part of the algebra in the next chapter.

3.4.1.5 A foundation for Web Service Description Languages.

Tightly integrating the π-calculus with the syndication calculus also has a profound effect on the application domain. It is well known that calculi which pass information on channels model communication across a network. For instance, several Web Services may send each other messages on channels. This message passing can be modelled by channel based process calculi, such as the π-calculus [37]. Thus, by tightly integrating the syndication calculus with the π-calculus, a foundation for a high level language for both Linked Data and Web Services is provided. Here two examples of scenarios which can be captured in the extended calculus are presented.

Passing SPARQL results on channels. The first example below models the passing of results from queries on channel. The example is adapted from the introduction to this chapter, which passes a result to a continuation process. Here channels are employed to perform the passing of

results to a continuation process P. The result of the transition is the same as the example in the introduction, but more processes are employed to perform the operation.

$$
\begin{array}{l}
(\mathit{Armstrong\ p{:}clubs\ res{:}Gateshead\_F.C.})\,, \\
(\mathit{Armstrong\ rdf{:}type\ dbp{:}SoccerPlayer})\,, \\
\overline{\mathit{eg{:}endpoint\ eg{:}return\_channel}}, \\
\bigvee b. \\
\left(\begin{array}{l}
\mathit{eg{:}endpoint}\ b \\
\quad \bigvee a.\left(\begin{array}{l}
|(a\ \mathit{p{:}club\ res{:}Gateshead\_F.C.})| \\
|(a\ \mathit{rdf{:}type\ dbp{:}SoccerPlayer})| \\
\overline{b\ a}
\end{array}\right)
\end{array}\right), \\
\bigvee a.(\mathit{eg{:}return\_channel}\ a\ ;\ P)
\end{array}
\quad \rhd \quad
\begin{array}{l}
(\mathit{Armstrong\ p{:}clubs\ res{:}Gateshead\_F.C.})\,, \\
(\mathit{Armstrong\ rdf{:}type\ dbp{:}SoccerPlayer})\,, \\
P\left\{{}^{\mathit{Armstrong}}/_{a}\right\}
\end{array}
$$

The example above uses two channels *eg:endpoint* and *eg:return_channel*, which are both URIs. The first channel outputs the return channel. The query is composed with an input which receives a channel on the channel *eg:endpoint*. This channel received is then used to return the URI discovered by the query. A process P is guarded by a channel which inputs the URI passed on the channel *eg:return_channel*. The value passed on *eg:return_channel* is the result of the query *Armstrong*. Thus the effect is that the three processes and the stored triples are coordinated, such that they pass the URI *Armstrong* to the process P.

In the above example the tensor product is used to combine the input and output channels with the query. This ensures that all operations — the input, query and output — happen in the same atomic transition. The above example can be made more asynchronous by using ‘then’ instead of tensor. The result is the following configuration, which takes three steps to reduce to the same

configuration instead of one.

$$
\begin{array}{l}
(\mathit{Armstrong\ p{:}clubs\ res{:}Gateshead\_F.C.})\,, \\
(\mathit{Armstrong\ rdf{:}type\ dbp{:}SoccerPlayer})\,, \\
\overline{\mathit{eg{:}endpoint\ eg{:}return\_channel}}, \\
\bigvee b. \\
\left(\begin{array}{l} \mathit{eg{:}endpoint\ b}; \\ \bigvee a. \\ \left(\begin{array}{l} |(\mathit{a\ p{:}club\ res{:}Gateshead\_F.C.})| \\ |(\mathit{a\ rdf{:}type\ dbp{:}SoccerPlayer})|\,; \\ \overline{b\ a} \end{array} \right) \end{array} \right), \\
\bigvee a.(\mathit{eg{:}return\_channel\ a}\ ;\ P) \\
\qquad \rhd\ (\mathit{Armstrong\ p{:}clubs\ res{:}Gateshead\_F.C.})\,, \\
\qquad\ (\mathit{Armstrong\ rdf{:}type\ dbp{:}SoccerPlayer})\,, \\
\qquad\ \bigvee a. \left(\begin{array}{l} |(\mathit{a\ p{:}club\ res{:}Gateshead\_F.C.})| \\ |(\mathit{a\ rdf{:}type\ dbp{:}SoccerPlayer})|\,; \\ \overline{\mathit{eg{:}return\_channel\ a}} \end{array} \right), \\
\qquad\ \bigvee a.(\mathit{eg{:}return\_channel\ a}\ ;\ P) \\
\qquad\qquad \rhd\ (\mathit{Armstrong\ p{:}clubs\ res{:}Gateshead\_F.C.})\,, \\
\qquad\qquad\ (\mathit{Armstrong\ rdf{:}type\ dbp{:}SoccerPlayer})\,, \\
\qquad\qquad\ \overline{\mathit{eg{:}return\_channel\ Armstrong}}, \\
\qquad\qquad\ \bigvee a.(\mathit{eg{:}return\_channel\ a}\ ;\ P) \\
\qquad\qquad\qquad \rhd\ (\mathit{Armstrong\ p{:}clubs\ res{:}Gateshead\_F.C.})\,, \\
\qquad\qquad\qquad\ (\mathit{Armstrong\ rdf{:}type\ dbp{:}SoccerPlayer})\,, \\
\qquad\qquad\qquad\ P\left\{{}^{\mathit{Armstrong}}/_{a}\right\}
\end{array}
$$

Any transition that the first example above can make, the second example above can make that same transition in zero or more steps. In this sense, the first synchronous process is stronger (more deterministic) than the second asynchronous process. Note that this relationship is an example of weak simulation, which is not investigated further in this work.

Using SPARQL to discover Web Services. The previous example demonstrates using Web Services to pass information discovered using queries. Now consider the related scenario of using queries to discover information about Web Services. The Web Service Description Language (WSDL) is a W3C standard for publishing data about Web Services [32]. The information in a WSDL document can easily be lifted to RDF [81]. This means that SPARQL can be used to query WSDL to discover information about Web Services. The following example demonstrates a process which uses some WSDL (lifted to RDF) to discover services then coordinate an operation.

The following is an example of using WSDL to discover two services to used in a transaction. One endpoint is ready to output a token; the other is ready to input a token. A process discovers

that the endpoints implement complementary interfaces and causes them to interact. The result is that the token is passed from one endpoint to another endpoint. The prefix *wsdl:* abbreviates `http://www.w3.org/ns/wsdl-rdf#` from the WSDL to RDF mapping [81].

$$
\begin{array}{l}
(\mathit{eg{:}serviceA}\ \mathit{wsdl{:}endpoint}\ \mathit{eg{:}endpointA})\,, \\
(\mathit{eg{:}serviceA}\ \mathit{wsdl{:}implements}\ \mathit{eg{:}offer\_token})\,, \\
(\mathit{eg{:}serviceB}\ \mathit{wsdl{:}endpoint}\ \mathit{eg{:}endpointB})\,, \\
(\mathit{eg{:}serviceB}\ \mathit{wsdl{:}implements}\ \mathit{eg{:}receive\_token})\,, \\
\overline{\mathit{eg{:}endpointA}\ \texttt{`The\_Token'}}\,, \\
\bigvee y.(\mathit{eg{:}endpointB}\ y\ ;\ P)\,, \\
\qquad\qquad \bigvee a,b,c,d. \left(\begin{array}{l}
|(c\ \mathit{wsdl{:}endpoint}\ a)| \\
|(c\ \mathit{wsdl{:}interface}\ \mathit{eg{:}offer\_token})| \\
|(d\ \mathit{wsdl{:}endpoint}\ b)| \\
|(d\ \mathit{wsdl{:}interface}\ \mathit{eg{:}receive\_token})| \\
\bigvee x.\left(\ a\ x \otimes \overline{b\ x}\ \right)
\end{array} \right) \\
\qquad\qquad\qquad\qquad \rhd\ (\mathit{eg{:}serviceA}\ \mathit{wsdl{:}endpoint}\ \mathit{eg{:}endpointA})\,, \\
\qquad\qquad\qquad\qquad\quad\ (\mathit{eg{:}serviceA}\ \mathit{wsdl{:}implements}\ \mathit{eg{:}offer\_token})\,, \\
\qquad\qquad\qquad\qquad\quad\ (\mathit{eg{:}serviceB}\ \mathit{wsdl{:}endpoint}\ \mathit{eg{:}endpointB})\,, \\
\qquad\qquad\qquad\qquad\quad\ (\mathit{eg{:}serviceB}\ \mathit{wsdl{:}implements}\ \mathit{eg{:}receive\_token})\,, \\
\qquad\qquad\qquad\qquad\quad\ P\left\{ {}^{\texttt{`The\_Token'}}/_{y} \right\}
\end{array}
$$

The above example is obviously a very simple example. Many Web Services will engage in more complex exchanges of messages. Formal models for message exchange patterns have been investigated as session types [37]. In future work, it would be possible to apply session types to this extended calculus. Thereby the models of Web Services and Linked Data can be tightly integrated.

3.5 A Comparison to Established Logics

> Instead of teaching logic to nature, it is more reasonable to learn from her.
>
> Jean-Yves Girard [56]

It is widely acknowledged that Linear Logic is one of the most exciting modern developments in logic [50]. Indeed it challenges the definition of logic itself. Basic assumptions made by the school of Tarski about the relationship between the syntax and semantics of logic, no longer apply in Linear Logic [56].

Linear Logic was introduced by Girard [55], due to insight gained from his earlier work on cut elimination for System F [54]. System F is a second order intuitionistic logic, which extends intuitionistic logic with polymorphism. Cut elimination is a process of normalising proofs. For

$$
\begin{array}{lll}
b := & x & \text{variable} \\
\mid & a & \text{name} \\
\\
A := & (b\ b\ b) & \text{triples}
\end{array}
\qquad
\begin{array}{lll}
P := & A & \text{atoms} \\
\mid & P^{\perp} & \text{linear negation (nil)} \\
\mid & P \otimes P & \text{multiplicative conjunction (tensor)} \\
\mid & P \parr P & \text{multiplicative disjunction (par)} \\
\mid & \mathrm{I} & \text{multiplicative true (one)} \\
\mid & \perp & \text{multiplicative false (nothing)} \\
\mid & P \,\&\, P & \text{additive conjunction (with)} \\
\mid & P \oplus P & \text{additive disjunction (plus)} \\
\mid & \top & \text{additive true (top)} \\
\mid & 0 & \text{additive false (zero)} \\
\mid & \bigvee x.P & \text{additive existential quantification (some)} \\
\mid & \bigwedge x.P & \text{additive universal quantification (any)}
\end{array}
$$

FIGURE 3.12: A syntax for Linear Logic with triples as atoms.

intuitionistic logic, such as System F, normalisation corresponds to the evaluation of functions. Normalisation of function is the essence of the operational behaviour of functional programming languages.

Intuitionistic logics, such as System F, controls the use of the structural rules, which determine the number of times a premise can be used. Linear Logic goes beyond intuitionistic logic by demonstrating that common connectives can be decomposed into more fundamental operations. The decomposition allows fine control over the use of structural rules, exposing new connectives.

The insight offered by Linear Logic has proven to be useful for developing logics for real systems where the resources consumed are important, such as programming languages which manipulate memory [106, 119]. Hence variants of Linear Logic are an ideal setting for investigating the subtleties of update languages.

3.5.1 A syntax for Linear Logic.

The syntax of Linear Logic is built from atomic formulae. The choice of the formulae are dependent on the application. For instance the atoms could be triples of names or variables, where variables are place holders for names. In examples, triples are used as atoms to emphasise that Linear Logic applies directly to RDF. This allows a more concrete intuition.

The syntax of Linear Logic with triples as atoms is presented in Fig. 3.12. Linear negation can be applied to any formula. The meaning of linear negation is very different to negation in classical and intuitionistic logics, as explained in Section 3.5.2.

The remaining constructs are classified into sub-logics. Firstly, there are the multiplicatives, which form Multiplicative Linear Logic. Multiplicative Linear Logic is the restriction to the logic $(P, (\_)^{\perp}, \otimes, \parr, \mathrm{I}, \perp)$, introduced in Section 3.5.4. Multiplicative Additive Linear Logic extends Multiplicative Linear Logic with the additive constructs $(\&, \oplus, \top, 0, \bigvee, \bigwedge)$, introduced in

Section 3.5.5. Full Linear Logic is obtained by including the exponentials Why Not and Of Course (?, !). However, these exponentials are not required for this work, so full Linear Logic is not presented here.

3.5.2 Linear negation v.s. classical and intuitionistic negation

In Linear Logic any formula can be negated. The notion of negation in Linear Logic is very different from negation in logics such as classical and intuitionistic logics. Here the differences are informally discussed.

In classical logic negation is interpreted via Boolean algebras (most generally as the Boolean algebra of sets that are both closed and open in a suitable topological space [130]). A Boolean algebra has universe which dominates all other elements. If a formula is interpreted as a set in a Boolean algebra then its negation is all elements in the universe which are not in the original set.

For instance, take the universe to be all people affiliated to a research group. Now put all people that label themselves as 'practical' in a set, then go around calling everyone in the complement of the set 'not practical.' This leads to problems, since just because the someone was not explicitly tagged as 'practical' in the data available it does not mean they are 'not practical.' Thus classical negation is only suited to closed systems with complete information.

A problem in logic is that the law of the excluded middle does not necessarily hold. The law of the excluded middle states that either a formula holds or does not hold. The failure of this law was confounded by Gödel's famous incompleteness theorem, which demonstrates that facts are not necessarily provably true or false. The subtleties of the law had been anticipated by mathematical philosophers, such as Poincaré and Brouwer [114, 34]. Intuitionistic negation models negation using a bottom element. The bottom element, false, represents a contradiction. To say that a formula is false is to say that it is refutable by counter example.

For instance, suppose that it is assumed that being labelled as both 'practical' and 'theoretical' is intuitionistically negated. Thus, if there is someone labelled both as 'practical' and as 'theoretical,' then a contradiction arises. But clearly there are scenarios where such a statement should be allowed to hold. So in Linked Data, where a such constraints on data are subjective, it is possible for constraints to be imposed which are not suited to all parties involved. Thus intuitionistic negation is not suited to global systems.

Linear negation is more tolerant since no boundaries or constraints are imposed. Linear negation identifies complementary pairs of formulae. The negation of a formula is the largest formula which interacts perfectly with the formulae. If a formula demands that 'you are practical' then the linear negation of the formula can simply consumes that information.

$$(P \parr Q) \parr R \equiv P \parr (Q \parr R) \qquad P \parr \bot \equiv P \qquad P \parr Q \equiv Q \parr P$$

FIGURE 3.13: The structural rules of Linear Logic: associativity, unit and exchange.

In the syndication calculus linear negation only applies to atoms. A future aim is to extend linear negation to the entire calculus. The concept of linear negation has considerable depth as demonstrated by its use for modelling particles and anti-particles in physics, which are complementary rather than contradictory [134, 29, 12]. Linear negation is also used in models of hardware, where linear negation is used to switch the direction of components [129].

3.5.3 Structural rules of Linear Logic.

The subtlety of Linear Logic arises from limiting the structural rules of the logic. The standard structural rules used in logic are weakening, contraction and exchange. Weakening is dropped, forbidding unused premises to appear. Also contraction is dropped, forbidding premises to be used more than once.

Control of structural rules allows control of the number of times a formula is used in a proof. The control of resources is useful for shared memory concurrency, where only locked resources are considered. Hence control of structural rules is relevant to the concurrent updates of this work. Control of resources is also useful for interaction where two endpoint interact, rather than broadcast.

Having dropped weakening and contraction, the structural rule that remains is exchange. Exchange allows formulae which are composed by par to be swapped. Associativity is also permitted for par. The unit of par is $\bot$. Thus $(P, \parr, \bot)$ a commutative monoid. Exchange and associativity and the unit are captured by a structural congruence over formulae, presented in Fig. 3.13. The structural congruence can be applied at any point in a logical deduction. This style of presentation highlights connections with the other calculi introduced this work, which use an identical structural congruence.

More subtle versions of Linear Logic restrict the use of these structural rules further, such as non-commutative Linear Logics which can be used to model observations sensitive to time [134, 61]. Furthermore, Lambek discovered applications in linguistics, where restriction of the use of associativity is required [85].

3.5.4 Multiplicative Linear Logic.

Firstly, consider the subsystem which consist of only the multiplicative connectives. The multiplicative connectives are analogues to the classical connectives ‘and’, ‘or’, ‘true’ and ‘false’ — $\otimes$, $\parr$, I and $\bot$ respectively. Par and negation define linear implication $P \multimap Q := P^{\bot} \parr Q$. This

$$\vdash P^{\perp} \parr P \qquad \frac{\vdash P \parr R \quad \vdash Q \parr S}{\vdash (P \otimes Q) \parr R \parr S} \qquad \frac{\vdash Q \parr P \quad \vdash P^{\perp} \parr R}{\vdash Q \parr R}$$

FIGURE 3.14: A deductive system for multiplicative Linear Logic

definition of linear implication is similar to the definition of classical implication in classical logic ($P \rightarrow Q \coloneqq \neg P \vee Q$). The rules of the deductive system for Multiplicative Linear Logic are presented in Fig. 3.14.

The De Morgan properties of the multiplicatives. The De Morgan properties in classical logic reveals a duality between 'and' and 'or' with respect to 'not'. This duality is lost in intuitionistic logic; but is recovered again in Linear Logic. The De Morgan properties of Multiplicative Linear Logic are defined in Fig. 3.15. Each connective is De Morgan dual to another connective.

The axioms. Given a formula, its linear negation is the weakest formula which interacts perfectly with the original formula. The basic axioms state that a formula and its negation interact perfectly. These basic axioms are just the formulae of the form $P \multimap P$, which are the pure axioms of many constructive logics, such as Gentzen's system LJ [51]. For instance, (*Jim is practical*) and its negation interact perfectly. Thus their composition using par holds, as follows.

$$\vdash (\textit{Jim is practical}) \parr (\textit{Jim is practical})^{\perp}$$

The axioms of Linear Logic appears in two forms in this work in communication and in storage. For communication, a formula 'You are practical' represents an output; while its negation represents the complementary input. The input and output interact consuming each other. For storage, if the formula 'You are practical' represents a stored triple, then its negation is the formula that deletes that fact. So the interaction of the stored triple and the delete command removes the triple'. For instance, a delete looks as follows in the syndication calculus.

$$(\textit{Jim is practical}) \parr (\textit{Jim is practical})^{\perp} \rhd \perp$$

It is clarified, using algebra in the the next chapter, that the above reduction axiom is equivalent to the axioms of Linear Logic.

$$\left(A^{\perp}\right)^{\perp} \equiv A \qquad (P \parr Q)^{\perp} \equiv P^{\perp} \otimes Q^{\perp} \qquad \mathrm{I}^{\perp} \equiv \perp$$

FIGURE 3.15: De Morgan properties for the multiplicatives.

The tensor product. The axioms show that par enables interaction. In contrast, tensor forbids interaction. In the example bellow, the first two atoms are joined using tensor, thus the single observer of those atoms expects both to be answered using disjoint resources. The third and forth atoms are negated, hence offer the complementary observations. The par connective allows the necessary interactions to validate this formula.

$$\vdash \left((\textit{Jim is theoretical})^{\perp} \otimes (\textit{Jim is practical})^{\perp}\right) \parr (\textit{Jim is practical}) \parr (\textit{Jim is theoretical})$$

The separation imposed by tensor is useful for concurrency, since each of the premises can be evaluated independently in parallel. The tensor product is used to synchronously compose operations in the calculus in this work. For instance, the following reduction results in two separate triples being synchronously deleted.

$$\left((\textit{Jim is theoretical})^{\perp} \otimes (\textit{Jim is practical})^{\perp}\right) \parr (\textit{Jim is practical}) \parr (\textit{Jim is theoretical}) \rhd \perp$$

The algebra in the next chapter verifies that the tensor in the calculus is equivalent to the tensor in Linear Logic.

The units. The units I and $\perp$ are the units of $\otimes$ and $\parr$ respectively. The unit $\perp$ can always be introduced in a formula using the unit rule of the structural congruence. The unit I holds as an axiom since I $\parr \perp$ is an axiom and I $\parr \perp \equiv$ I.

In subtle variations of Linear Logic the units are equivalent, due to the mix rule [61]. However in the calculus introduced in this work the units are distinguished. In the calculus, I is the unit transition and equivalent to the top element of an embedded Boolean algebra. However, $\perp$ is the empty context which makes no transitions. The unit transition interacts perfectly with the empty context, due to the following axiom instance.

$$\vdash \mathrm{I} \parr \perp$$

The above axiom corresponds to the following reduction. The reduction holds due to the filter axiom, since true is always satisfied, and since $\perp$ is the unit of par.

$$\mathrm{I} \parr \perp \rhd \perp$$

The cut rule. The counterpoint to the basic axioms is the cut rule. The cut rule allows two premises to be composed by cancelling out a formula in one premise and its negation the other premise. The famous cut elimination theorem for Linear Logic states that the resulting conclusion can be obtained without using cut [55]. The proof pushes the cut rule up through the proof tree, interacting wherever possible, until it can be replaced by basic axioms.

Theorem 3.3 (Cut elimination)**.** *In Linear Logic, if a formula holds, then the same formula holds without using the cut rule.*

$$\vdash \top \parr P \qquad \frac{\vdash P \parr R}{\vdash (P \oplus Q) \parr R} \qquad \frac{\vdash P \parr R \quad \vdash Q \parr R}{\vdash (P \,\&\, Q) \parr R} \qquad \frac{\vdash P\{^{a}/_{x}\} \parr Q}{\vdash \bigvee x.P \parr Q} \qquad \frac{\vdash P\{^{y}/_{x}\} \parr Q}{\vdash \bigwedge x.P \parr Q}$$

FIGURE 3.16: A deductive system for the additives: Top, Plus, With, Some and Any. y must not appear free in the conclusion of the rule for Any.

The big question is where the cut rule appear in the syndication calculus. The answer is subtle. In the current form linear negation does not extend to the entire calculus. Linear negation does however extend to the fragment of the calculus which corresponds to Multiplicative Linear Logic.

This suggests the following cut rule for the syndication calculus, defined below. $P, Q, \ldots$ are arbitrary processes, while A is a formula in Multiplicative Linear Logic with triples as atoms.

$$\frac{P \parr A \rhd P' \quad Q \parr A^{\perp} \rhd Q'}{P \parr Q \rhd P' \parr Q'}$$

This rule will be discussed in the next chapter, Sec. 4.6.4. It is argued that a soundness result in the next chapter is a weak cut elimination result. The prospect of a full cut elimination result in an extended calculus is also discussed. A full cut elimination result would demonstrate that the model is more than just a calculus... it is also a logic.

3.5.5 Multiplicative Additive Linear Logic.

Multiplicative Linear Logic can be extended with additives. The rules for the additives have a different intuition to the multiplicatives. The multiplicatives capture a control of usage of resources intuition; whereas the intuition for the additives is a control of possible worlds.

Like the multiplicatives, the additives include analogues to classical conjunction, disjunction, true and false — with, plus, top and zero, respectively. There are also the additive quantifiers some and any which correspond to existential and universal quantification. The deductive system for the additives are defined in Fig. 3.16.

De Morgan properties of the additives. The De Morgan properties of the additives are presented in Fig. 3.17. These rules state that additive conjunction is De Morgan dual to additive disjunction, similarly to multiplicative conjunction and disjunction. The De Morgan properties also apply to the units, where top and zero are De Morgan dual and some and any are De Morgan dual.

Additive disjunction and conjunction. Plus $\oplus$ presents two branches, where only one branch may be selected. Both branches are considered non-deterministically with respect to the same context, but only one branch is chosen. Plus is choose in the syndication calculus.

With & presents two worlds that are simultaneously exists, but cannot interact. Both worlds must hold in the same context. With can be seen as considering two possible computation paths simultaneously. This can be useful for modelling a suspended choice in conventional computing, and superposition in quantum computing.

Consider the example below of an additive conjunction and additive disjunction which interact. With presents two possible worlds, one in which 'Jim is theoretical' and the other in which 'Jim is practical.' Both worlds share the same context. The context is a choice of two branches. One branch deletes 'Jim is practical' and the other branch deletes 'Jim is theoretical.' In both worlds there is a suitable branch that can be chosen, so the formula holds.

$$\vdash ((\textit{Jim is theoretical}) \,\&\, (\textit{Jim is practical})) \parr \Big((\textit{Jim is practical})^{\perp} \oplus (\textit{Jim is theoretical})^{\perp}\Big)$$

It is possible to add the connective With to the syndication calculus. However, the interaction of With and the continuation operators of the calculus is non-trivial. Hence, for clarity, this work refrains from including additive conjunction in the syndication calculus. Additive conjunction is discussed here for purely objective reasons, to show that a more complete calculus can be obtained.

The additive units. The top element is defined by an axiom. The axiom states that in the presence of $\top$ any formula holds. This axiom can be added to the syndication calculus. It represents the process which does anything.

The program which does anything would be useful when the calculus is used for logical specifications. For instance, to ensure that a program deletes 'Jim is theoretical' but is also allow to simultaneously to anything else, it can be checked the program refines the specification $\top \otimes (\textit{Jim is theoretical})^{\perp}$.

There is no axiom for zero, which means that zero corresponds to the formula which never holds. This corresponds to the false constraint in the syndication calculus. The false constraint forces a deadlock.

The additive first-order quantifiers. The quantifiers Some and Any are the only way of accessing names and variables in atoms. The quantifiers are De Morgan dual, as with For All and Exists in classical logic.

Some $\bigvee$ represents an infinite choice where the formulae can be satisfied under any substitution of a variable for a name. This is particularly useful for defining programs which receive a name

$$(P \,\&\, Q)^{\perp} \equiv P^{\perp} \oplus Q^{\perp} \qquad \top^{\perp} \equiv 0 \qquad \Big(\bigvee x.P\Big)^{\perp} \equiv \bigwedge x.P^{\perp}$$

FIGURE 3.17: De Morgan properties for the additives.

from some context. Some is used in the π-calculus for inputting a name, and in the syndication calculus for discovering a URI in a query.

Any $\bigwedge$ represents a formula which holds regardless of the choice variable. A temporary variable is substituted for the quantified variable to represent the quantified name. The side condition ensures that the temporary variable chosen does not introduce any new interactions; thus any other variable or name could equally have been chosen. Any corresponds to the blank node quantifier in the syndication calculus.

The following formula is an example of an interaction between Some and Any. The formula on the left has two variables bound by the Any quantifier. In the formula on the right the corresponding variables are bound by the Some quantifier. The two formulae interact. Variables quantified using Any can only be addressed using Some in the following fashion.

$$\vdash \bigwedge x.\bigwedge y.(x \mathit{\ knows\ } y) \parr \bigvee a.\bigvee b.(a \mathit{\ knows\ } b)^{\perp}$$

The above formula corresponds to the following reduction in the syndication calculus. The process demonstrates an update which discovers a triple where the subject and object are blank nodes. The update then deletes that triple.

$$\bigwedge x.\bigwedge y.(x \mathit{\ knows\ } y) \parr \bigvee a.\bigvee b.(a \mathit{\ knows\ } b)^{\perp} \rhd \perp$$

The rule for blank nodes in the syndication calculus is slightly more involved. This is due to the interaction of the blank node quantifier and the continuation operator.

3.5.6 The exponentials of Linear Logic.

Full Linear Logic extends Multiplicative Additive Linear Logic with modalities called exponentials. The exponentials recover the structural rules of weakening and contraction. The exponentials ensures that Linear Logic is sufficiently strong to embed intuitionistic logic.

In the calculi introduced in this work, a different style of exponential is used. Girard himself acknowledges problems with exponentials, confounded by models of Linear Logic based on Banach spaces, where exponentials are analytic (limits of power series) [55, 56]. He suggests that a solution is to modify the rules of the exponential itself.

This work defines exponentials such that coproducts convert to tensor (Why Not converts coproducts to par). This results in exponentials which resemble the Kleene star of Kleene algebra [41]. The Kleene star appears in the calculus as iteration. An alternative would be to use least and greatest fixed point operators, which would increase the expressive power of the calculus [11].

3.6 Conclusion on the Deductive System

The main inspirations for the reduction system are process calculi and Linear Logic. The innovation is to extend the sequent calculus of Linear Logic with an extra place holder which accumulates the continuation process. The rules borrowed from Linear Logic provide a spatial dimension, which allows large synchronous actions and interactions to be expressed. Synchronous actions are required for complex updates. The innovative continuation provides a temporal dimension to the calculus. Several connections between operational semantics and Linear Logic have been attempted, but this presentation is new [18, 79, 72].

The match with Linear Logic is not exact, but several rules are almost identical. Half of the rules borrowed from Linear Logic appear in an identical form on the left. The rules also accumulate a monoid of continuations on the right of the sequent. These rules are the following: additive disjunction, which models choice; the tensor product, which models the synchronous join of updates and additive existential quantification, which models the input of a variable.

A rule similar to additive universal quantification is used to model local variables, called blank nodes. The rule acts like universal quantification on both sides of the sequent. The additive zero of Linear Logic appears as a false in the embedded Boolean algebra, while the multiplicative true of Linear Logic appears as true in the embedded Boolean algebra.

The structural rules of the calculus, which ensure par is a commutative monoid, match the structural rules of Linear Logic. Also, the basic axiom of Linear Logic appears as the delete axiom, which permits the interaction of a triple and its complement.

However, par presents the most prominent departure from its Linear Logic analogue. The context rule for par, taken from the π-calculus, means that a resource need not be used in the current operational step. Instead the resource may be ignored and perhaps used in a subsequent operational step. This means that there is not an exact duality between tensor and parallel composition, which is core to Linear Logic.

One way to resolve this is to decompose par into several operators, as in ACP [20]. Parallel composition can then choose to commit to one of the four operations: a proper par operation indicating interaction, a left merge operation, a right merge operation or a tensor product indicating true concurrency without interaction. The left and right merge operations are defined using the continuation operator ‘then.’

The main rule which appear in Linear Logic, but not in the syndication calculus is additive conjunction. There are several ways to include additive conjunction in the calculus. Additive conjunction provides interesting models of computation, where the intuition is multiple computation paths which cannot interact. However such a feature is not obviously required for modelling Linked Data, so is left as a note in this work.

It is worth noting that the section on Feeds is a relic of the original hypothesis of this work. The original hypothesis was based on the observation that RDF had no update mechanisms, but had

suitable query languages and inference mechanisms. The Atom Publishing Protocol however does have a standardised update mechanism, and also the expressive power to deliver RDF. The idea was to demonstrate that RDF and Atom integrate seamlessly in a high level programming language. The programming language would take the form of a controller that coordinates Atom updates and SPARQL Queries, for the back end of a Web application.

This goal is achieved to a large extent. Unfortunately, there is no related work in the community which pursues this line of work. Instead the trend, endorsed recently by Tim Berners-Lee is to pursue updates by extending SPARQL Update [22]. This shift was relatively easy to make, which explains the emphasis in this work on SPARQL Update rather than the Atom Publishing Protocol. The SPARQL Update approach and the Atom Publishing Protocol approach provide different levels of granularity of update, so can both be employed in different scenarios.

Chapter 4

Algebra for Read–Write Linked Data

This chapter investigates the algebraic properties of the syndication calculus. For background material, the chapter beings with an introduction to the π-calculus, which follows on from the reduction system for the π-calculus explained in the previous chapter. This introduces the concepts of labelled transition systems and bisimulation equivalences.

These concepts are then applied to the syndication calculus. Due to a wider spectrum of possible interactions, the syndication calculus demands a more expressive framework than the π-calculus. The resulting algebra is also richer, uncovering some canonical algebraic structures which frequently appear in computer science applications.

4.1 Motivating Examples for the Algebra

There are several reasons why an algebra for processes is desirable. Firstly, there is equivalence checking. Equivalence checking is useful to programmers, who need confirmation that writing a process in different ways has the same meaning. If two programs are not the same, then an algebra may be used to show that one process is more deterministic than another process. The more deterministic process is a refinement of the original process; hence can be used in a more specific situation.

An algebra is important for the development of efficient compilers for languages based on the calculus. Two programs may have the same operational behaviour, but may differ in efficiency when deployed on a specific computer architecture. Query planners make use of an algebra, referred to as relational algebra in relational databases. The algebra is used to rewrite a query so that it can be executed as efficiently as possible.

The algebra derived in this chapter applies to queries, updates and processes; hence the techniques used for query planning can be applied to more general processes. Furthermore, the

algebra employed is proven to be correct using the modern proof technique of coinduction. This is the first work to employ such techniques to extend relational algebra to a much broader setting.

Further to enhancing programming techniques and implementations, a good algebra provides objective justification for the calculus. If an operator satisfies well understood algebraic properties, then it is more likely to be correct than if its algebraic properties appear to be arbitrary. For instance idempotent semirings are very common structures in a wide range of applications in computer science. The fact that idempotent semirings appear is justification that the operators involved have been correctly defined. Furthermore, since semirings are well understood, the properties of semirings may be exploited.

4.1.1 Normal forms for processes

The previous chapter introduces data for Joe Armstrong the footballer. Some queries and updates which used the data were considered. This section returns to these examples to consider the effect of the algebra on the processes. The examples demonstrate the use of algebra for rewriting processes to normal forms. Rewriting to normal forms is useful for programming and optimisation.

Firstly, consider the simple update example. This update can be rewritten to a form such that all quantifiers are pulled to the outside, all deletes are grouped together, then all inserts, then all queries, then all constraints. This results in the following equivalence.

$$*\bigvee a.\left(\begin{array}{l}\bigvee x.\left(\begin{array}{l}|(a\ \mathit{dbp{:}birthDate}\ x)|\\(x \leq \text{`01-01-1950'})\end{array}\right)\\(a\ \mathit{dbp{:}position}\ \mathit{res{:}Inside\_forward})^{\perp}\\(a\ \mathit{dbp{:}position}\ \mathit{res{:}Attacking\_midfielder})\end{array}\right)$$
$$\sim\ *\bigvee a, x.\left(\begin{array}{l}(a\ \mathit{dbp{:}position}\ \mathit{res{:}Inside\_forward})^{\perp}\\(a\ \mathit{dbp{:}position}\ \mathit{res{:}Attacking\_midfielder})\\|(a\ \mathit{dbp{:}birthDate}\ x)|\\(x \leq \text{`01-01-1950'})\end{array}\right)$$

This simple rewrite make use of several rules. Firstly, the scope of the select quantifier which binds x can be extended over tensor since x does not appear free in the insert or the delete. Secondly, the tensor operator, which combines the delete, insert, query and constraint, is commutative. This allows the operators to be reordered. This new form clearly corresponds to the more constrained syntax of updates which was defined in the first proposal for SPARQL Update from Hewlett-Packard Laboratories [126]. In that proposal the above process would be written as follows.

```
DELETE {
  ?a dbp:position res:Inside_forward
```

```
}
INSERT {
  ?a dbp:position res:Attacking_midfielder
}
WHERE {
  ?a dbp:birthDate ?x
  FILTER (?x <= '01-01-1950')
}
```

This normal form demonstrates that the update language could be quickly implemented. Updates in the calculus can be normalised, then rewritten to updates in the original language of HP-Labs. The HP-Labs language has been implemented; thus the updates can be executed over real RDF stores. This allows the calculus to be used as a high level language for updating RDF; however it does not allow the results of this work to be used to verify and optimise updates.

4.1.2 A disjunctive and a conjunctive normal form

Several normal forms are envisioned. The choice of which form to use will depend on the underlying architecture. Consider the more substantial example from the previous chapter. The original form of the query is presented below.

$$\bigvee a.\left(\begin{array}{l}\bigvee z.\left(\begin{array}{l}\left(\begin{array}{l}\bigvee x,y.\left(\begin{array}{l}|(a\ \textit{foaf:givenName}\ x)|\\ |(a\ \textit{foaf:familyName}\ y)|\\ (z = x + \text{`\ '} + y)\end{array}\right)\\ \oplus\\ |(a\ \textit{foaf:name}\ z)|\end{array}\right)\\ (z \in \texttt{`J.* Armstrong'})\end{array}\right)\\ \left(\begin{array}{l}|(a\ \textit{rdf:type dbp:Athlete})|\\ \oplus\\ |(a\ \textit{rdf:type dbp:Artist})|\end{array}\right);\\ P\end{array}\right)$$

The query above can be rewritten to the query below. This rewrite shows that the original query can be written as the disjunction of four queries in a normal form.

$$\bigvee a.\left(\begin{array}{l} \bigvee given, family.\left(\begin{array}{l} |(a\ \textit{rdf:type}\ \textit{dbp:Artist})| \\ |(a\ \textit{foaf:givenName}\ given)| \\ |(a\ \textit{foaf:familyName}\ family)| \\ (given + \text{`` ''} + family) \in \texttt{`J.* Armstrong'} \end{array}\right); P \end{array}\right)$$
$$\oplus$$
$$\bigvee a.\left(\bigvee full.\left(\begin{array}{l} |(a\ \textit{rdf:type}\ \textit{dbp:Artist})| \\ |(a\ \textit{foaf:name}\ full)| \\ a \in \texttt{`J.* Armstrong'} \end{array}\right); P\right)$$
$$\oplus$$
$$\bigvee a.\left(\bigvee full.\left(\begin{array}{l} |(a\ \textit{rdf:type}\ \textit{dbp:Athlete})| \\ |(a\ \textit{foaf:name}\ full)| \\ full \in \texttt{`J.* Armstrong'} \end{array}\right); P\right)$$
$$\oplus$$
$$\bigvee a.\left(\begin{array}{l} \bigvee given, family.\left(\begin{array}{l} |(a\ \textit{rdf:type}\ \textit{dbp:Athlete})| \\ |(a\ \textit{foaf:givenName}\ given)| \\ |(a\ \textit{foaf:familyName}\ family)| \\ (given + \text{`` ''} + family) \in \texttt{`J.* Armstrong'} \end{array}\right); P \end{array}\right)$$

The above rewrite uses many of the algebraic properties established in this chapter. It uses the facts that select quantifiers and choice operators are colimits, that colimits distribute over tensor, that tensor forms a commutative monoid, that the semiring structure of Boolean algebras is a subalgebra of the semiring structure of Kleene algebras and algebraic properties of continuations. All of these algebraic properties are proven and discussed in this chapter.

By translation to HP Labs implementation of SPARQL Update, each of the four patterns above can be implemented. Given also an extra mechanism for externally choosing between several updates, the above full process can be implemented. The disjunctive normal form of updates and queries described in this section is only one potential normal form. The disjunctive normal reveals the degree of branching in processes.

A conjunctive normal form may be more useful for concurrency. A conjunctive normal form would rewrite a process so that it consists of the tensor product of several processes. Each part of the process decomposed using the tensor product is considered separately using disjoint resources.

Consider the process above, extended with iteration as a prefix. Suppose that two stores are queried. Suppose that one uses the name predicate, while the other expresses uses the givenName and familyName predicates. Thus the query could be rewritten as as the tensor of two

simpler queries. Each of the parts can then be executed over the store which uses the appropriate predicates.

$$*\bigvee a.\left(\begin{array}{l} \bigvee given, family.\left(\begin{array}{l}\left(\begin{array}{l}|(a\ \mathit{rdf{:}type}\ \mathit{dbp{:}Artist})|\\ \oplus\\ |(a\ \mathit{rdf{:}type}\ \mathit{dbp{:}Athlete})|\end{array}\right)\\ |(a\ \mathit{foaf{:}givenName}\ given)|\\ |(a\ \mathit{foaf{:}familyName}\ family)|\\ (given + \texttt{`}\ \texttt{'} + family) \in \texttt{`J.* Armstrong'}\end{array}\right)\\ \otimes\\ *\bigvee a.\left(\bigvee full.\left(\begin{array}{l}\left(\begin{array}{l}|(a\ \mathit{rdf{:}type}\ \mathit{dbp{:}Athlete})|\\ \oplus\\ |(a\ \mathit{rdf{:}type}\ \mathit{dbp{:}Artist})|\end{array}\right)\\ |(a\ \mathit{foaf{:}name}\ full)|\\ full \in \texttt{`J.* Armstrong'}\end{array}\right)\right);P\end{array}\right);P$$

The above process is equivalent to the previous processes prefixed with iteration. Further to the algebraic properties used already mentioned, the above example uses properties of iteration. Iteration is defined as a least fixed point, from which many algebraic properties can be derived. The property used above is that iteration converts disjunction to tensor. The algebraic properties of iteration are proven and discussed in this chapter.

4.2 A Labelled Transition System for the π-calculus.

Before considering the new calculus introduced in this work, an established calculus is discussed. This allows the techniques employed to be introduced. In Section 3.4.1.3, a reduction semantics for the π-calculus was presented. The π-calculus has an alternative operational semantics to the reduction semantics. The semantics can be expressed as a labelled transition system.

A labelled transition system is a relation consisting of two process and a label. The first process is the process before the transition; the label indicates the effect of the transition on the context; and the second process indicates the process after the transition. A version of the (early) labelled transition system for the π-calculus is presented in Figure 4.1.

A labelled transition system provided the original semantics of the π-calculus [99]. The labelled semantics is slightly more complicated than the reduction semantics, but has the advantage of enabling powerful proof techniques. The proof technique can be used to establish an algebra over π-calculus processes. Furthermore, for the π-calculus the reduction system and the labelled transition system are equivalent in expressive power.

$$a(x).P \xrightarrow{ab} P\{^{b}/_{x}\} \qquad a(x).P \xrightarrow{a(y)} P\{^{y}/_{x}\} \qquad \overline{a}b.P \xrightarrow{\overline{a}b} P \qquad \tau.P \xrightarrow{\tau} P$$

$$\frac{U \xrightarrow{l} P}{U \oplus V \xrightarrow{l} P} \qquad \frac{V \xrightarrow{l} Q}{U \oplus V \xrightarrow{l} Q} \qquad \frac{P \xrightarrow{l} P'}{\nu x.P \xrightarrow{l} \nu x.P'}\ x \notin \mathrm{n}(l) \qquad \frac{P \xrightarrow{ax} P'}{\nu x.P \xrightarrow{a(x)} P'}\ x \neq a$$

$$\frac{P \xrightarrow{l} P'}{P \parallel Q \xrightarrow{l} P' \parallel Q}\ \mathrm{bn}(l) \cap \mathrm{fn}(Q) = \emptyset \qquad \frac{P \xrightarrow{ab} P' \quad Q \xrightarrow{\overline{a}b} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'} \qquad \frac{P \xrightarrow{a(x)} P' \quad Q \xrightarrow{\overline{a}(x)} Q'}{P \parallel Q \xrightarrow{\tau} \nu x.(P' \parallel Q')}$$

FIGURE 4.1: The axioms and rules of the π-calculus labelled transition system: input axiom, bound input axiom, output axiom, τ axiom, choose left axiom, choose right axiom, fresh name context rule, open rule, par context rule, The symmetric versions of all the par rules are also included.

The input labels can take five forms. Firstly, consider the τ label, input label and output label. The τ label represents a transition without side effects. The input label ab represents the input of a name b on a channel a. The complementary output label $\overline{a}b$ represents the output of a name b on channel a. A complementary input and output label represent two perspectives of an interaction. The example below demonstrates how a basic communication can be established using input and output labels.

$$\frac{\overline{a}b.P \xrightarrow{\overline{a}b} P \quad a(x).Q \xrightarrow{ab} Q\{^{b}/_{x}\}}{\overline{a}b.P \parallel a(x).Q \xrightarrow{\tau} P \parallel Q\{^{b}/_{x}\}}$$

The input and output labels are both positioned on the labels using axioms. The axiom for output positions the output on the label and eliminates the output from the head of the process. The axiom for the input proceeds similarly, by positioning the complementary input on the label. The input label also instantiates the input variable with the name which is input. This instantiation is performed both on the label and the continuation process. The input label and the output label then communicate using the communication rule. Since the input and output labels match the result is a τ label. The τ label indicates that no further communications are required for the transition to take place.

There is also the complementary pair of input and output labels, represented by $a(x)$ and $\overline{a}(x)$ respectively. These represent ability to communicate on channel a where a bound name x is communicated. A bound name represents a fresh name. The example bellow demonstrates how a communication which involves a fresh name can be established.

$$\frac{\dfrac{\overline{a}b.P \xrightarrow{\overline{a}b} P}{\nu b.\overline{a}b.P \xrightarrow{\overline{a}(b)} P} \quad a(x).Q \xrightarrow{a(b)} Q\{^{b}/_{x}\}}{\nu b.(\overline{a}b.P) \parallel a(x).Q \xrightarrow{\tau} \nu b.(P \parallel Q\{^{b}/_{x}\})}$$

The bound input axiom is used to position a bound input on the label. The bound name is chosen

to match the bound name to be received, which results in a substitution in the continuation process. The output is positioned on the label as in the previous example. Since the name passed is bound by a fresh name quantifier, the open rule is then applied. Open explicitly indicates that the name is fresh, by indicating that the name is bound on the label. The bound input and bound output therefore match so the close rule is applied. The close rule works like the communication rule above except that the bound name encompasses both processes in the continuation. The result is that the bound name is passed to the process guarded by the input.

The choice rules work by selecting the left or right branch of a choice. The context rules work as in the reduction system, by allowing transitions to take place in an process context. The following example demonstrates the use of choice and a context rule.

$$\dfrac{\dfrac{\overline{a}c.0 \xrightarrow{\overline{a}c} 0}{\overline{a}c.0 \parallel \overline{b}d.0 \xrightarrow{\overline{a}c} 0 \parallel \overline{b}d.0} \quad \dfrac{a(x).P \xrightarrow{ac} P\{^{c}/_{x}\}}{a(x).P \oplus b(y).Q \xrightarrow{ac} P\{^{c}/_{x}\}}}{\overline{a}c.0 \parallel \overline{b}d.0 \parallel (a(x).P \oplus b(y).Q) \xrightarrow{\tau} 0 \parallel \overline{b}d.0 \parallel P\{^{a}/_{x}\}}$$

In the example above the par context rule allow one of the outputs to occur. The unused output remains untouched in the continuation. The choice rule is used to choose the branch which is capable of the complementary input. The input and output labels then interact as normal.

This completes a brief overview of the operational semantics of the π-calculus. For this version of the π-calculus, and many variants, the labelled transition system and the reduction system have equivalent expressive power. This means that the natural notion of equivalence in each system coincides. The equivalence of the two approaches is used to give further justification that the labelled transition system and the reduction system have been correctly formulated. A version of this result is proven for the syndication calculus in this chapter.

4.2.1 Bisimulations for the π-calculus.

The labelled transition system is particularly useful as a foundation for novel proof techniques. A proof technique, known as bisimulation, can be used to demonstrate that two processes have identical behaviour. Bisimulation can be used to prove that algebraic properties of the π-calculus hold. Bisimulation is dual to the technique of induction, which is used to prove that algebraic properties of data-structures hold.

Bisimulation allows processes to be treated equivalently if they exhibit the same observable behaviour. The observable behaviour of a process is established by considering the labels of the labelled transition system. Such observations allow two processes to be compared. If a particular label can be observed for one process, then the same label can be observed for another process, and vice versa. Furthermore, the continuations of both processes must maintain the same observational behaviour. Bisimulation can be formalised as follows.

Definition 4.1. A bisimulation is an equivalence relation over processes $\sim_0$ such that the following holds. If $P \sim_0 Q$ and $P \xrightarrow{l} P'$ then there exists Q' such that $Q \xrightarrow{l} Q'$ and $P' \sim_0 Q'$. Bisimilarity, written $\sim$ is the greatest bisimulation.

In categorical models bisimulation has been demonstrated to be a canonical concept. In particular, it is the dual notion to the technique of induction used by every mathematician [120]. The definition of bisimulation above is defined directly over the labelled transition system of the calculus, so a categorical foundation is not required here. However, a categorical foundation for bisimulation in the π-calculus can be achieved by combining co-algebraic methods with permutation groups [102].

Induction dismantles data-structures (such as natural numbers or trees) down to some base element (zero for the natural number for instance), to prove that some property is shared by the structures. On the other hand co-induction tracks the possibly infinite behaviours of dynamic systems to demonstrate that some property will always be maintained by regardless of the choice of computation path. This explains why there is no base case in the definition of bisimulation above.

Induction is concerned with the least structure satisfying the given properties (the initial algebra). Dually co-induction is concerned with the largest number of possible behaviours which satisfy a property (the final co-algebra). This explains why the greatest bisimulation is considered in the example above.

Many notions of bisimulation have been studied. The above definition is for strong bisimulation, where the two processes are compared step by step. Other notions include weak bisimulation, which allows multiple steps to be taken by each process that is being compared using weak bisimulation. Strong bisimulation is useful when the atomicity of actions in the calculus must be accounted for. Atomicity is prominent in this application, therefore strong bisimulation is the appropriate notion of bisimulation.

A complete algebra for the π-calculus is established using bisimulation [99]. Here one example of an algebraic identity is provided. The following processes are bisimilar. This can be established by case analysis.

$$a(x).P \parallel \overline{a}b.0 \sim a(x).(P \parallel \overline{a}b.0) \oplus \overline{a}b.a(x).P \oplus \tau.P$$

Both processes can make a transition with labels $\overline{a}b$, ac, $a(c)$ and τ, for any c, to identical processes. Identical processes are trivially bisimilar, thus the bisimulation is established.

A famous non-example is provided. The first process below can make a τ transition to $\overline{a}c.0 \oplus \overline{b}d.0$. The second process can also make a τ transition. However, for either τ transition of the second process neither continuation is bisimilar to $\overline{a}c.0 \oplus \overline{b}d.0$, since $\overline{a}c.0 \oplus \overline{b}d.0$ can still observe either choice but for both $\overline{a}c.0$ and $\overline{b}d.0$ the choice is already determined.

$$\tau.\overline{a}c.0 \oplus \tau.\overline{b}d.0 \nsim \tau.(\overline{a}c.0 \oplus \overline{b}d.0)$$

The notion of bisimulation can be weakened, so that it is not necessarily symmetric. This weaker notion is a preorder called simulation, where the process on the left may have a more deterministic behaviour. In the example above the first process simulates the second process, since after the first observation both $\overline{a}c.0$ and $\overline{b}d.0$ do simulate $\overline{a}c.0 + \overline{b}d.0$.

Indeed, any trace (sequence of observations) of a process simulates a process and the sum of the traces simulates the process. However the above example demonstrates the sum of the traces is not in general bisimilar to the original process. This demonstrates that bisimilarity is a stronger notion than trace equivalence, as used for CSP for instance [70].

The established techniques for the π-calculus can be applied to the syndication calculus. A labelled transition system will be formulated and an algebras derived using bisimulation. Many algebraic properties of the syndication calculus have an analogous algebra in the π-calculus. However, several more insightful algebraic properties will be established, which are not revealed by the π-calculus.

4.3 A Labelled Transition System for a Sub-Calculus

The operational semantics of the syndication calculus, introduced in the previous chapter, can be expressed as a labelled transition system. This provides an alternative operational semantics to the reduction system. This alternative semantics allows the behaviour of queries and data to be evaluated separately and then composed. Theorem 4.8 verifies that the labelled transition system is sound with respect to the reduction system.

4.3.1 The sub-calculus considered

A restricted version of the calculus is considered in this chapter. The focus is placed on queries and inserted data. This restricted version of the calculus makes the labels easy to understand. The restriction on the calculus is only allow processes with queries as actions; rather than both queries and updates as actions.

The restriction is not for any technical reason; a labelled transition system can be provided for the whole language. An almost identical semantics can be provided where queries are replaced by updates, or feature along side updates. The only change required is to include more information in the labels. Readers familiar with labelled transition semantics will be more familiar with only one type of communication being handled by the labels. The same algebraic properties established in this section would hold if instead deletes had been chosen rather than queries.

The restricted syntax for this chapter is presented in Fig. 4.2. The syntax excludes delete updates. This leaves a query language with continuations. Stored triples are kept separate from queries, instead of being modelled as a trivial insert operation. The syntax of triples and constraints are unchanged. Also, the extensions for named graphs are excluded.

$$
\begin{array}{rll}
U ::= & |C| & \text{asked triple} \\
| & \phi & \text{filter} \\
| & U \oplus U & \text{choice} \\
| & U \otimes U & \text{tensor} \\
| & \bigvee a.U & \text{select name} \\
| & \bigvee x.U & \text{select literal} \\
| & *U & \text{iteration} \\
| & U \,;\, P & \text{then}
\end{array}
\qquad
\begin{array}{rll}
P ::= & \bot & \text{nothing} \\
| & P \parr P & \text{par} \\
| & \bigwedge a.P & \text{blank node} \\
| & U & \text{query} \\
| & C & \text{stored triple}
\end{array}
$$

FIGURE 4.2: The restricted syntax of queries (U) and processes (P), over triples (C) and constraints (ϕ).

$$
\begin{array}{rll}
E ::= & \mathrm{I} & \text{unit} \\
| & C & \text{triple} \\
| & E \otimes E & \text{combination}
\end{array}
\qquad
(E \otimes F) \otimes G \equiv (E \otimes F) \otimes G \qquad E \otimes \mathrm{I} \equiv E
$$

$$
E \otimes F \equiv F \otimes E
$$

FIGURE 4.3: The syntax and congruence for monoids which appear on labels.

The reduction semantics are the same as the previous chapter, in Fig. 3.4. Only the delete axiom and the insert axiom are no longer required. The reduction semantics are considered to be the definitive semantics for the calculus. Thus, the alternative labelled transition semantics and algebraic semantics must be verified to be at least sound with respect to the reduction semantics. The main results of this section establish that soundness does indeed hold.

4.3.2 The purpose of labels

A labelled transition consists of two processes and a label. The first process is the process before the transition. The label is a constraint on the context in which a transition can take place. The second process is the resulting process after the transition.

The labels are formed from a commutative monoid over triples $(E, \otimes, \mathrm{I})$, as defined in Fig. 4.3. A label indicates the inputs and outputs of a process. An input indicates that a process can proceed if it can receive the triples on the label from its context. An output indicates that a process outputs the triple on the label to its context. For instance, the query below inputs a triple; while the stored triple below outputs a triple.

$$
|(b_4 \; \mathit{knows} \; b_3)| \,;\, P \xrightarrow{(b_4 \; \mathit{knows} \; b_3)} P \qquad (b_4 \; \mathit{knows} \; b_3) \xrightarrow{\overline{(b_4 \; \mathit{knows} \; b_3)}} (b_4 \; \mathit{knows} \; b_3)
$$

A relevant interpretation is that the first transition above is an action from the perspective of a client which resolves a query; whereas the second is an action from the perspective of a server that provides a triple. Two processes composed in parallel with matching inputs and outputs may interact. For instance, the above processes can be composed, resulting in the following

$$\frac{C \sqsubseteq D}{|D| \xrightarrow{C} \bot} \qquad \frac{U \xrightarrow{E} Q}{U\,;P \xrightarrow{E} Q \parr P} \qquad \frac{U \xrightarrow{E} P \quad V \xrightarrow{F} Q}{U \otimes V \xrightarrow{E \otimes F} P \parr Q} \qquad \frac{\vDash \phi}{\phi \xrightarrow{\mathrm{I}} \bot}$$

$$\frac{S \xrightarrow{E} P}{S \oplus T \xrightarrow{E} P} \qquad \frac{T \xrightarrow{E} Q}{S \oplus T \xrightarrow{E} Q} \qquad \frac{S\left\{{}^{b}/_{a}\right\} \xrightarrow{E} Q}{\bigvee a.S \xrightarrow{E} Q} \qquad \frac{S\{{}^{v}/_{x}\} \xrightarrow{E} Q}{\bigvee x.S \xrightarrow{E} Q}$$

$$*S \xrightarrow{\mathrm{I}} \bot \qquad \frac{S \xrightarrow{E} P}{*S \xrightarrow{E} P} \qquad \frac{*S \otimes *S \xrightarrow{E} P}{*S \xrightarrow{E} P}$$

FIGURE 4.4: Labelled transitions for queries: input triple, trigger guard, tensor, filter, choose left, choose right, select name, select literal, weakening, dereliction and contraction.

transition. The unit label indicates an operational step without side effects.

$$(|(b_4\ knows\ b_3)|\,;P)\,,(b_4\ knows\ b_3) \xrightarrow{\mathrm{I}} P,(b_4\ knows\ b_3)$$

Output labels can also indicate extruded names. For instance, the example below extrudes the name a. The extruded names represent blank nodes where the scope of the blank node quantifier may be extended. This is similar to extrusion of new names in the π-calculus [99].

$$\bigwedge a.(a\ has\ paper)\,,(b_2\ has\ stone) \xrightarrow{a|\overline{(a\ has\ paper)}} (a\ has\ paper)\,,(b_2\ has\ stone)$$

The commutative monoid rules can always be applied to reorder labels.

4.3.3 Labelled transitions for queries

The input transitions allow the behaviour of a query to be modelled independently. The rules for queries are presented in Fig. 4.4. The rules accumulate RDF triples on an input label, which represents contexts in which a query may be answered.

The 'input triple' rule poses the triple as an input on the label. The triple on the label may be strengthened by the preorder over triples. The 'trigger guard' rule allows a continuation process to be triggered exposing the continuation. The following example demonstrates a query consisting of a single triple and a continuation process, where the preorder $colleague \sqsubseteq knows$ is assumed.

$$|(b_4\ knows\ b_3)|\,;P \xrightarrow{(b_4\ colleague\ b_3)} P$$

Select quantifiers are resolved by anticipating the name or literal to input. For instance, the following labelled transition indicates that the query can be answered in a context where a name

is chosen. The same name is passed to the continuation process.

$$\bigvee a.(|(b_4\ knows\ a)|\ ;\ P) \xrightarrow{(b_4\ knows\ b_3)} P\left\{{}^{b_3}/_{a}\right\}$$

Choices are resolved by anticipating the left or right branch. For instance, the following transition indicates the label and continuation which results from choosing the left branch.

$$(|(b_4\ knows\ b_2)|\ ;\ P) \oplus (|(b_4\ knows\ b_3)|\ ;\ Q) \xrightarrow{(b_4\ knows\ b_2)} P$$

Tensor synchronises two queries, by composing their respective labels and continuations. For instance, the following query simultaneously inputs two triples. The continuations of both queries are triggered in parallel, with the appropriate substitutions.

$$\bigvee a.((|(b_4\ knows\ a)|\ ;\ P) \otimes (\bigvee x.|(a\ name\ x)|\ ;\ Q)) \xrightarrow{(b_4\ knows\ b_2)\otimes(b_2\ name\ \texttt{`John'})} P\left\{{}^{b_2}/_{a}\right\}, Q\left\{{}^{b_2,\texttt{`John'}}/_{a,x}\right\}$$

A constraint is disposed when it is satisfied. For instance, in the following query the length of a selected literal is constrained, but satisfied by the substitution.

$$\bigvee x.(|(b_2\ name\ x)| \otimes (|x| \leq 4)\ ;\ P) \xrightarrow{(b_2\ name\ \texttt{`John'})} P\left\{{}^{\texttt{`John'}}/_{x}\right\}$$

Iteration anticipates the number of copies of a query to pose using weakening, dereliction and contraction. For instance, two copies of following query are posed using contraction and dereliction. The label indicates the two separate triples which are to be answered simultaneously. Both continuations are composed in parallel.

$$*\bigvee a.(|(b_4\ knows\ a)|\ ;\ P) \xrightarrow{(b_4\ knows\ b_2)\otimes(b_4\ knows\ b_3)} P\left\{{}^{b_2}/_{a}\right\}, P\left\{{}^{b_3}/_{a}\right\}$$

The rules of the labelled transition system are sufficient to model queries.

4.3.4 Labelled transitions for an RDF store

The behaviour of stored RDF triples can be modelled using output labels. The rules of output labels are presented in Fig. 4.5. The names extruded on the label are indicated by α, where + indicates disjoint union of names. The abbreviation $\bigwedge\alpha.P$ is used to indicate the quantification of all names in α.

Stored triples can output the triple on the label. The same triple appears in the continuation unchanged. The preorder over names may be used to weaken the output triple. Names are extruded on the label using the 'open scope' rule. For instance, the following triple outputs a

$$\frac{C \sqsubseteq D}{C \xrightarrow{\overline{D}} C} \qquad \frac{P \xrightarrow{\alpha|\overline{E}} Q}{\bigwedge a.P \xrightarrow{\alpha+a|\overline{E}} Q}\ a \notin \mathrm{fn}(\beta) \qquad \frac{P \xrightarrow{\alpha|\overline{E}} Q}{\bigwedge a.P \xrightarrow{\alpha|\overline{E}} \bigwedge a.Q}\ a \notin \alpha \cup \mathrm{fn}(E)$$

$$\frac{P \xrightarrow{\alpha|\overline{E}} P'}{P \parr Q \xrightarrow{\alpha|\overline{E}} P' \parr Q}\ \alpha \cap \mathrm{fn}(Q) = \emptyset \qquad \frac{P \xrightarrow{\alpha_0|\overline{E}} P' \quad Q \xrightarrow{\alpha_1|\overline{F}} Q'}{P \parr Q \xrightarrow{\alpha_0+\alpha_1|\overline{E \otimes F}} P' \parr Q'}\ \begin{array}{l}\alpha_0 \cap \mathrm{fn}(Q) = \emptyset \\ \alpha_1 \cap \mathrm{fn}(P) = \emptyset\end{array}$$

$$\frac{P \xrightarrow{E \otimes F} P' \quad Q \xrightarrow{\alpha|\overline{F}} Q'}{P \parr Q \xrightarrow{E} \bigwedge \alpha.(P' \parr Q')}\ \alpha \cap (\mathrm{fn}(P) \cup \mathrm{fn}(E)) = \emptyset$$

FIGURE 4.5: Process rules: output triple, open, blank node context, par context, parallel outputs and close. The symmetric versions of the par context and close rule are also assumed. Note $\mathrm{fn}(\beta)$ is the set of names for which alias assumptions are defined.

triple and extrudes the blank node, using the assumption *colleague* $\sqsubseteq$ *knows*.

$$\bigwedge b_4.(b_4\ \mathit{colleague}\ b_3) \xrightarrow{b_4|\overline{(b_4\ \mathit{knows}\ b_3)}} (b_4\ \mathit{colleague}\ b_3)$$

Output labels composed in parallel can be combined. Extruded names on both labels must be disjoint to preserve the scope of blank nodes. For instance, the following transition simultaneously outputs two triples and extrudes three names.

$$\bigwedge b_4.(\bigwedge b_2.(b_4\ \mathit{knows}\ b_2), \bigwedge b_3.(b_4\ \mathit{knows}\ b_3)) \xrightarrow{b_2,b_3,b_4|\overline{(b_4\ \mathit{knows}\ b_2) \otimes (b_4\ \mathit{knows}\ b_3)}} (b_4\ \mathit{knows}\ b_2), (b_4\ \mathit{knows}\ b_3)$$

Two parallel processes may interact using the close rule. Close allows complementary inputs and outputs to be matched. Names extruded on the output label are introduced as quantifiers in the continuation. Any inputs not answered remain on the resulting label, to be answered later. For instance, the following iterated query is answered twice. One copy is answered by the available process and the other copy must be answered by the context for the transition to occur. In the continuation, the scope of the blank node is extended.

$$*\bigvee a.((b_4\ \mathit{knows}\ a)\ ;P), \bigwedge b_3.(b_4\ \mathit{knows}\ b_3) \xrightarrow{(b_4\ \mathit{knows}\ b_2)} \bigwedge b_3.\left((b_4\ \mathit{knows}\ b_3), P\left\{{}^{b_3}/_{a}\right\}\right)$$

The context rule for parallel composition allows processes which do not contribute to an interaction to idle. Similarly, the context rule for blank node quantifiers allows a blank node to be ignored in a transition if it does not appear on the label.

4.3.5 The operational power of the labelled transition system

To justify the labelled transition system, the operational power of the labelled transition system is shown to match the operational power of the reduction system is verified. To show that the operational powers match, it is demonstrated that if a unit labelled transition can be derived then the corresponding reduction can also be derived. The significance is that, given the independent perspectives of the query and the store in terms of labelled transitions, their combination satisfies the global perspective specified by the reduction system.

Scope extrusion presents technical difficulties. The following technical lemma reduces these difficulties, by eliminating scope extrusion. The proof demonstrates that combinations of opening names and closing names can be eliminated from an proof tree which uses an extruded name using a structural congruence.

Lemma 4.2 (Elimination of extrusion)**.** *Suppose that a labelled transition proof uses name extrusion, but not in the conclusion. The same labelled transition, up to structural congruence, holds without any name extrusion.*

Proof. Consider the interaction of an input label and an output label, with an extruded name using the close rule.

Consider the structure of the output label. Firstly, demonstrate that if $Q \xrightarrow{\alpha|\overline{E}} Q'$, then there exists an R such that $Q \equiv \bigwedge \alpha.R$ and $R \xrightarrow{\overline{E}} Q'$. There are two cases to consider.

Consider the case of composition of output labels as follows.

$$\frac{P \xrightarrow{\alpha_0|\overline{D}} P' \quad Q \xrightarrow{\alpha_1|\overline{E}} Q'}{P \parr Q \xrightarrow{\alpha_0+\alpha_1|\overline{D\otimes E}} P' \parr Q'}$$

By induction, there exists a process P_0 such that $P \equiv \bigwedge \alpha_0.P_0$ and $P_0 \xrightarrow{\overline{D}} P'$. Similarly, there exists a process Q_0 such that $Q \equiv \bigwedge \alpha_1.Q_0$ and $Q_0 \xrightarrow{\overline{E}} Q'$. Therefore $P \parr Q \equiv \bigwedge(\alpha_0 + \alpha_1).(P_0 \parr Q_0)$ and the following proof tree holds.

$$\frac{Q_0 \xrightarrow{\overline{E}} Q' \quad P_0 \xrightarrow{\overline{D}} P'}{P_0 \parr Q_0 \xrightarrow{\overline{D\otimes E}} P' \parr Q'}$$

Consider the case of blank node restriction, where α_0 and α_1 are disjoint.

$$\frac{P \xrightarrow{\alpha_0|\overline{E}} P'}{\bigwedge \alpha_1.P \xrightarrow{\alpha_0|\overline{E}} P'}$$

By induction, there exists a process P_0 such that $P \equiv \bigwedge \alpha_0.P_0$ and $P_0 \xrightarrow{\overline{E}} P'$. Therefore $\bigwedge \alpha_1.P \equiv \bigwedge \alpha_0.\bigwedge \alpha_1.P_0$ and the following proof tree holds.

$$\frac{P_0 \xrightarrow{\overline{E}} P'}{\bigwedge \alpha_1.P_0 \xrightarrow{\overline{E}} P'}$$

Now consider the composition of an input transition and an output transition with an extruded name.

$$\frac{P \xrightarrow{C \otimes D} P' \quad Q \xrightarrow{\alpha | \overline{D}} Q'}{P \parr Q \xrightarrow{C} \bigwedge \alpha.(P' \parr Q')}$$

By the above lemma, there exists a process Q_0 such that $Q \equiv \bigwedge \alpha.Q_0$ and $Q_0 \xrightarrow{\overline{D}} Q'$. So $P \parr Q \equiv \bigwedge \alpha.(P \parr Q_0)$ and the following holds.

$$\frac{\dfrac{P \xrightarrow{C \otimes D} P' \quad Q_0 \xrightarrow{\overline{D}} Q'}{P \parr Q_0 \xrightarrow{C} P' \parr Q'}}{\bigwedge \alpha.(P \parr Q_0) \xrightarrow{C} \bigwedge \alpha.(P' \parr Q')}$$

By applying the above inductively a proof tree scope extrusion is eliminated. □

Every completed labelled transition can also be expressed as a reduction, Lemma 4.3. The proof works by transforming proof trees so that labels used in interactions are eliminated.

Lemma 4.3 (Elimination of labels)**.** *If* $P \xrightarrow{\mathrm{I}} Q$ *then* $P \rhd Q$.

Proof. Firstly, apply Lemma 4.2 to eliminate extrusion of names. Therefore any unit transition follows from some context rules and a close rule, where no names are extruded.

For a parallel composition where one process is idled, the idled process is pushed down the tree. Given the first proof tree bellow, the second holds.

$$\frac{\dfrac{P \xrightarrow{C} P'}{P \parr Q \xrightarrow{C} P' \parr Q} \quad R \xrightarrow{\overline{C}} R'}{(P \parr Q) \parr R \xrightarrow{\mathrm{I}} (P' \parr Q) \parr R'} \qquad \text{yields} \qquad \frac{P \xrightarrow{C} P' \quad R \xrightarrow{\overline{C}} R'}{P \parr R \xrightarrow{\mathrm{I}} P' \parr R'}$$

By induction, $P \parr R \rhd P' \parr R'$ holds. Hence $P \parr Q \parr R \rhd P' \parr Q \parr R'$ holds, by the idle rule.

A blank node which is not used is eliminated as follows. Given the first proof tree bellow, where $a \notin \mathrm{fn}(C)$ and using alpha conversion $a \notin \mathrm{fn}(P)$, the second holds.

$$\frac{P \xrightarrow{C} P' \quad \dfrac{Q \xrightarrow{\overline{C}} Q'}{\bigwedge a.Q \xrightarrow{\overline{C}} \bigwedge a.Q'}}{P \parr \bigwedge a.Q \xrightarrow{\mathrm{I}} P' \parr \bigwedge a.Q'} \qquad \text{yields} \qquad \frac{P \xrightarrow{C} P' \quad Q \xrightarrow{\overline{C}} Q'}{P \parr Q \xrightarrow{\mathrm{I}} P' \parr Q'}$$

By induction, $P \parr Q \rhd P' \parr Q'$ holds. Hence $P \parr \bigwedge a.Q \rhd P' \parr \bigwedge a.Q'$ holds, by the blank node rule.

Consider the case of axioms. The first proof tree bellow can be replaced by the second below. Transitivity over content on the labels can be eliminated trivially by applying transitivity pointwise to the names in a triple.

$$\dfrac{\dfrac{D \sqsubseteq C}{|C| \xrightarrow{D} \bot} \quad \dfrac{E \sqsubseteq D}{E \xrightarrow{\overline{D}} E}}{|C| \parr E \xrightarrow{\mathrm{I}} \bot \parr E} \qquad \text{maps to} \qquad \dfrac{E \sqsubseteq C}{|C| \parr E \rhd E}$$

For choice the cut is pushed through the brackets. So assuming the first proof tree holds, the second proof tree holds.

$$\dfrac{\dfrac{U \xrightarrow{C} Q}{U \oplus V \xrightarrow{C} Q} \quad P \xrightarrow{\overline{C}} R}{(U \oplus V) \parr P \xrightarrow{\mathrm{I}} Q \parr R} \qquad \text{yields} \qquad \dfrac{U \xrightarrow{C} Q \quad P \xrightarrow{\overline{C}} R}{U \parr P \xrightarrow{\mathrm{I}} Q \parr R}$$

Hence by induction $U \parr P \rhd Q \parr R$ holds. Therefore $(U \oplus V) \parr P \rhd Q \parr R$ holds, the choose left rule. A symmetric proof works for choose right.

For select the cut is pushed through the brackets. So assuming the first proof tree holds, the second proof tree holds.

$$\dfrac{\dfrac{U\left\{{}^{b}/_{a}\right\} \xrightarrow{C} Q}{\bigvee a.U \xrightarrow{C} Q} \quad P \xrightarrow{\overline{C}} R}{\bigvee a.U \parr P \xrightarrow{\mathrm{I}} Q \parr R} \qquad \text{yields} \qquad \dfrac{U\left\{{}^{b}/_{a}\right\} \xrightarrow{C} Q \quad P \xrightarrow{\overline{C}} R}{U\left\{{}^{b}/_{a}\right\} \parr P \xrightarrow{\mathrm{I}} Q \parr R}$$

Hence by induction $U\left\{{}^{b}/_{a}\right\} \parr P \rhd Q \parr R$ holds. Therefore $\bigvee a.U \parr P \rhd Q \parr R$ holds, by the select rule. A similar proof works for selecting literals.

For weakening the translation of proof trees is direct. The labelled transition $*U \xrightarrow{\mathrm{I}} \bot$ becomes $*U \rhd \bot$. Similary for filters, the labelled transition $\phi \xrightarrow{\mathrm{I}} \bot$ becomes $\phi \rhd \bot$, assuming that $\vDash \phi$ holds.

For dereliction the cut is pushed through the brackets. So assuming the first proof tree holds, the second proof tree holds.

$$\dfrac{\dfrac{U \xrightarrow{C} Q}{*U \xrightarrow{C} Q} \quad P \xrightarrow{\overline{C}} R}{*U \parr P \xrightarrow{\mathrm{I}} Q \parr R} \qquad \text{yields} \qquad \dfrac{U \xrightarrow{C} Q \quad P \xrightarrow{\overline{C}} R}{U \parr P \xrightarrow{\mathrm{I}} Q \parr R}$$

Hence by induction $U \parr P \rhd Q \parr R$ holds. Therefore $*U \parr P \rhd Q \parr R$ holds, by the select rule. A similar proof works for selecting literals.

For contraction the cut is pushed through the brackets. So assuming the first proof tree holds, the second proof tree holds.

$$\dfrac{\dfrac{*U \otimes *U \xrightarrow{C} Q}{*U \xrightarrow{C} Q} \quad P \xrightarrow{\overline{C}} R}{*U \parr P \xrightarrow{\mathrm{I}} Q \parr R} \qquad \text{yields} \qquad \dfrac{*U \otimes *U \xrightarrow{C} Q \quad P \xrightarrow{\overline{C}} R}{(*U \otimes *U) \parr P \xrightarrow{\mathrm{I}} Q \parr R}$$

Hence by induction $(*U \otimes *U) \parr P \rhd Q \parr R$ holds. Therefore $*U \parr P \rhd Q \parr R$ holds, by the contraction rule.

For continuations, assuming the first proof tree holds below, then the second proof tree holds.

$$\dfrac{\dfrac{U \xrightarrow{C} \bot}{U \mathbin{;} P \xrightarrow{C} P} \quad Q \xrightarrow{\overline{C}} R}{(U \mathbin{;} P) \parr Q \xrightarrow{\mathrm{I}} Q \parr R} \qquad \text{yields} \qquad \dfrac{U \xrightarrow{C} \bot \quad Q \xrightarrow{\overline{C}} R}{U \parr Q \xrightarrow{\mathrm{I}} R}$$

Hence by induction $U \parr Q \rhd R$ holds. Therefore $(U \mathbin{;} P) \parr Q \rhd P \parr R$ holds, by the continuation rule.

Tensor has two cases. The first reorders the application of outputs. Suppose the outputs are composed It is possible to first compose one part of the rule then another part.

$$\dfrac{\dfrac{(U \otimes V) \xrightarrow{C \otimes D} P' \quad Q \xrightarrow{\overline{D}} Q'}{(U \otimes V) \parr Q \xrightarrow{C} P \parr Q'} \quad R \xrightarrow{\overline{C}} R'}{((U \otimes V) \parr Q) \parr R \xrightarrow{\mathrm{I}} (P \parr Q') \parr R'}$$

Using associativity, the following transition is equivalent up to structural congruence. Repeated application normalises the proof tree.

$$\dfrac{(U \otimes V) \xrightarrow{C} P \quad \dfrac{Q \xrightarrow{\overline{D}} Q' \quad R \xrightarrow{\overline{C}} R'}{Q \parr R \xrightarrow{\overline{C \otimes D}} Q' \parr R'}}{(U \otimes V) \parr (Q \parr R) \xrightarrow{\mathrm{I}} P \parr (Q' \parr R')}$$

Now consider the tensor rule as for previous rules. Given the following proof tree.

$$\dfrac{\dfrac{U \xrightarrow{C} R \quad V \xrightarrow{D} S}{U \otimes V \xrightarrow{C \otimes D} R \parr S} \quad \dfrac{P \xrightarrow{\overline{C}} P' \quad Q \xrightarrow{\overline{D}} Q'}{P \parr Q \xrightarrow{\overline{C \otimes D}} P' \parr Q'}}{(U \otimes V) \parr P \parr Q \xrightarrow{\mathrm{I}} R \parr S \parr P' \parr Q'}$$

the following proof trees hold.

$$\dfrac{U \xrightarrow{C} R \quad P \xrightarrow{\overline{C}} P'}{U \parr P \xrightarrow{\mathrm{I}} R \parr P'} \qquad \text{and} \qquad \dfrac{V \xrightarrow{D} S \quad Q \xrightarrow{\overline{D}} Q'}{V \parr Q \xrightarrow{\mathrm{I}} S \parr Q'}$$

By induction, $V \parr Q \rhd S \parr Q'$ and $U \parr P \rhd R \parr P'$ hold. Hence $(U \otimes V) \parr P \parr Q \rhd P' \parr Q'$ holds, by the tensor rule.

All cases are covered hence, by induction on the structure of a labelled proof, labels can be eliminated. □

Suppose that the rule of the interaction of labels is the cut rule. Note note that the reduction semantics provide cut free semantics. Then the above proof is a cut elimination result for the calculus. Each deduction in the labelled transition system, which uses cut, can be transformed into a deduction in the reduction system, which does not use cut.

A cut elimination proof has several types of cases to handle, as explained clearly in [131]. There is the 'identity case' which absorbs axioms. There are the 'commutative cases' which push the cut up the proof tree past rules which are not involved in the cut formula. The most involved rules are the 'key cases', which break break down pairs of complementary rules involved in a cut.

This correspondence between cut elimination and soundness of the labelled transition system is reflected in the form of the proof of Lemma 4.3. The 'identity case' is visible in the form of the 'axiom case.' Almost all other cases are 'commutative cases,' since they simply push the cut up the proof tree.

Due to the asymmetry of the calculus there is only one 'key case.' The key case to consider is when the tensor product and negated tensor product interact through the cut rule. The negated tensor product is just par from Linear Logic. Hence the shape of the tensor case is almost identical to the shape of the proof for the key case of the interaction of tensor and par in the cut elimination proof Linear Logic. Other key cases would only arise in an extended calculus, with additive conjunction for instance.

This argument presents a novel perspective on the rôle of cut elimination in operational semantics. The reader, may still not be convinced, since cut elimination is normally conducted within a single system. The argument that soundness of a labelled transition system with respect to a reduction system is a cut elimination results will be continued in Sec. 4.6.4. This section uses the algebraic semantics common to both systems, to translate between the two systems. More complete cut elimination results are also highlighted.

The converse of Lemma 4.3 holds. This converse lemma states that any commitment in the reduction system is a unit transition in the labelled transition system. The formulation in Lemma 4.4 makes explicit that this result is considered up to structural congruence [122]. This is because if a reduction holds then the redux of the corresponding labelled transition may not have exactly the same form unless structural congruence is applied. This emphasises that structural congruence is not required in the definition of the labelled transition system. The proof of the converse lemma follows by the same argument as for the original lemma above, so is not repeated.

Lemma 4.4 (Reductions are unit transitions)**.** *If $P \rhd Q$ then $P \xrightarrow{\mathrm{I}} Q'$, for some Q' such that $Q \equiv Q'$.*

Thus the local perspectives provided by the labelled transition system combine to provide the same operational power as the reduction system.

4.4 Equivalences for the Syndication Calculus

In this section the notion of bisimulation is introduced. Bisimulation is a useful proof technique for verifying algebraic properties over queries and processes. Bisimulation is demonstrated to be sound with respect to the natural equivalence in the reduction system.

4.4.1 Bisimulation and its congruence property

Processes which are capable of the same observable behaviour can be regarded as equivalent. The observable behaviour of a process is given by the labels of the labelled transition system. Observational equivalence of processes is established using the technique of (strong) bisimulation, as follows.

Definition 4.5 (Bisimulation)**.** Bisimulation, written $\sim$, is the greatest symmetric relation such that the following holds, for any label l. If $P \sim Q$ and $P \xrightarrow{l} P'$ then there exists some Q' such that $Q \xrightarrow{l} Q'$ and $P' \sim Q'$, where $\mathrm{bn}(l) \cap \mathrm{fn}(P, Q) = \emptyset$. Note $\mathrm{bn}(l)$ refers to the set of names extruded on the label l.

The following verifies that bisimulation is a congruence. A congruence relation holds in any context, which is necessary to use bisimulation as an algebra. A context is a process with a place holder for some syntax.

Lemma 4.6 (Bisimulation is a congruence)**.** *For processes P and Q, if $P \sim Q$ and C is a context for a process, then $CP \sim CQ$. For queries U and V, if $U \sim V$ and $\mathcal{D}$ is a context for a query, then $\mathcal{D}U \sim \mathcal{D}V$. Explicitly, the contexts for processes to check are as follows, assuming that $P \sim Q$.*

$$P \parr R \sim Q \parr R \qquad \bigwedge a.P \sim \bigwedge a.Q \qquad U\,;P \sim U\,;Q$$

Explicitly, the contexts for queries to check are as follows, assuming that $U \sim V$.

$$U \otimes W \sim V \otimes W \qquad U \oplus W \sim V \oplus W \qquad \bigvee a.U \sim \bigvee a.V \qquad *U \sim *V \qquad U\,;P \sim V\,;P$$

Proof. All cases are proven by assuming the existence of a bisimulation, then demonstrating that a bisimulation which contains the relation in question can be constructed.

Consider the case of a blank node context. Assume that $\sim_0$ is a bisimulation. Let $\sim_1$ be the least relation such that if $P \sim_0 Q$ then $P \sim_1 Q$ and $\bigwedge a.P \sim_1 \bigwedge a.Q$. Assume that $P \sim_0 Q$ and consider the cases of the open and blank node context rules.

For the open rule, assume that the first transition below holds. Since $P \sim_0 Q$ and $P \xrightarrow{\alpha_0|\overline{E}} P'$ there exists a process Q' such that $Q \xrightarrow{\alpha_0|\overline{E}} Q'$ and $P' \sim_0 Q'$. Hence the second transition below holds and $P' \sim_1 Q'$.

$$\frac{P \xrightarrow{\alpha_0|\overline{E}} P'}{\bigwedge \alpha_1.P \xrightarrow{\alpha_0+\alpha_1|\overline{E}} P'} \quad \text{yields} \quad \frac{Q \xrightarrow{\alpha_0|\overline{E}} Q'}{\bigwedge \alpha_1.Q \xrightarrow{\alpha_0+\alpha_1|\overline{E}} Q'}$$

For the blank node context rule, assume that the first transition holds, where α_0, α_1 are disjoint. Since $P \sim_0 Q$ and $P \xrightarrow{\alpha_0|\overline{E}} P'$ there exists a process Q' such that $Q \xrightarrow{\alpha_0|\overline{E}} Q'$ and $P' \sim_0 Q'$. So the second transition below holds and $\bigwedge \alpha_1.P' \sim_1 \bigwedge \alpha_1.Q'$.

$$\frac{P \xrightarrow{\alpha_0|\overline{E}} P'}{\bigwedge \alpha_1.P \xrightarrow{\alpha_0|\overline{E}} \bigwedge \alpha_1.Q'} \quad \text{yields} \quad \frac{Q \xrightarrow{\alpha_0|\overline{E}} Q'}{\bigwedge \alpha_1.Q \xrightarrow{\alpha_0+\alpha_1|\overline{E}} Q'}$$

The same argument works for input labels. Hence $\sim_1$ is a bisimulation, as required.

Consider the cases of par. Assume that $\sim_0$ is a bisimulation. Let $\sim_1$ be the least relation such that if $P \sim_0 Q$ then $P \parr R \sim_1 P \parr R$ and, recursively, if $P \sim_1 Q$ then $\bigwedge a.P \sim_1 \bigwedge a.Q$. Assume that $P \sim_0 Q$ and consider the following three cases.

For the case of the par context rule. Assume that the first transition below holds. Since $P \sim_0 Q$ and $P \xrightarrow{E} P'$, there exists Q' such that $Q \xrightarrow{E} Q'$ and $Q \sim_0 Q'$. Hence the second transition below holds and $P' \parr R \sim_1 Q' \parr R$.

$$\frac{P \xrightarrow{E} P'}{P \parr R \xrightarrow{E} P' \parr R} \quad \text{yields} \quad \frac{Q \xrightarrow{E} Q'}{Q \parr R \xrightarrow{E} Q' \parr R}$$

For the case of the parallel outputs, assume that the first transition below holds. Since $P \sim_0 Q$ and $P \xrightarrow{\alpha_1|\overline{E}} P'$, there exists Q' such that $Q \xrightarrow{\alpha_1|\overline{E}} Q'$ and $Q \sim_0 Q'$. Hence the second transition below holds and $R' \parr P' \sim_1 R' \parr Q'$.

$$\frac{R \xrightarrow{\alpha_0|\overline{D}} R' \quad P \xrightarrow{\alpha_1|\overline{E}} P'}{R \parr P \xrightarrow{\alpha_0+\alpha_1|\overline{D \otimes E}} R' \parr P'} \quad \text{yields} \quad \frac{R \xrightarrow{\alpha_0|\overline{D}} R' \quad Q \xrightarrow{\alpha_1|\overline{E}} Q'}{R \parr Q \xrightarrow{\alpha_0+\alpha_1|\overline{D \otimes E}} R' \parr Q'}$$

For the case of the close rule. Suppose that the first transition below holds. Since $P \sim_0 Q$, given that $P \xrightarrow{\alpha|\overline{D}} P'$ there exists a process Q' such that $Q \xrightarrow{\alpha|\overline{D}} Q'$ and $P' \sim_0 Q'$. Hence the second transition below holds and $\bigwedge \alpha.(R' \parr P') \sim_1 \bigwedge \alpha.(R' \parr Q')$.

$$\frac{R \xrightarrow{C \otimes D} R' \quad P \xrightarrow{\alpha|\overline{D}} P'}{R \parr P \xrightarrow{C} \bigwedge \alpha.(R' \parr P')} \quad \text{yields} \quad \frac{R \xrightarrow{C \otimes D} R' \quad Q \xrightarrow{\alpha|\overline{D}} Q'}{R \parr Q \xrightarrow{C} \bigwedge \alpha.(R' \parr Q')}$$

Other cases follow by symmetric arguments and the argument for closure of scope works as before. Therefore $\sim_1$ is a bisimulation.

Consider the case of choice. Assume that there exists a bisimulation $\sim_0$. Let $\sim_1$ be the least equivalence relation such that if $U \sim_0 V$ then $U \oplus W \sim_1 V \oplus W$ and if $P \sim_0 Q$ then $P \sim_1 Q$. To demonstrate that $\sim_1$ is a bisimulation, assume that $U \sim_0 V$ and consider $U \oplus W \sim_1 V \oplus W$. There is one non-trivial cases as follows.

Suppose that that, the first transition below holds, by choose left. Since $U \sim_0 V$, if $U \xrightarrow{C} P$ then there exists some Q such that $V \xrightarrow{C} Q$, such that $P \sim_0 Q$. So the second transition below holds and $P \sim_1 Q$, as required.

$$\frac{U \xrightarrow{C} P}{U \oplus W \xrightarrow{C} P} \qquad \text{yields} \qquad \frac{V \xrightarrow{C} Q}{V \oplus W \xrightarrow{C} Q}$$

Hence $\sim_1$ is a bisimulation. A symmetric argument works for the symmetric context.

Consider the case of tensor. Assume that there exists a bisimulation $\sim_0$. Let $\sim_1$ be the least equivalence relation such that if $U \sim_0 V$ then $U \otimes W \sim_1 V \otimes W$ and if $P \sim_0 Q$ then $P \parr R \sim_1 Q \parr R$ and $\bigwedge a.P \sim_1 \bigwedge a.Q$. Assume that $U \sim_0 V$ and consider $U \otimes W \sim_1 V \otimes W$.

Now, suppose the first transition below exists. Then, since $U \sim_0 V$ and $U \xrightarrow{C} P$ there exists a process Q such that $V \xrightarrow{D} Q$ and $P \sim_0 Q$. Hence the second transition below holds and $P \parr R \sim_1 Q \parr R$, as required.

$$\frac{U \xrightarrow{C} P \quad W \xrightarrow{D} R}{U \otimes W \xrightarrow{C \otimes D} P \parr R} \qquad \text{yields} \qquad \frac{V \xrightarrow{C} Q \quad W \xrightarrow{D} R}{V \otimes W \xrightarrow{C \otimes D} Q \parr R}$$

The argument for par works as before. Hence $\sim_1$ is a bisimulation.

Consider the case of select. Assume that there exists a bisimulation $\sim_0$. Let $\sim_1$ be the least equivalence relation such that if $U \sim_0 V$ then $\bigvee a.U \sim_1 \bigvee b.V$ and if $P \sim_0 Q$ then $P \sim_1 Q$. To demonstrate that $\sim_1$ is a bisimulation, assume that $U \sim_0 V$ and consider $\bigvee a.U \sim_1 \bigvee a.V$.

Firstly, two lemmas are established, both are established by simple structural induction. The first lemma states that, if $U\{b/a\} \xrightarrow{C} P$ then $U \xrightarrow{D} Q$ such that $C \leq D\{b/a\}$ and $P = Q\{b/a\}$. Now, since $U \sim V$ and $U \xrightarrow{D} Q$, by bisimulation, there exists Q' such that $V \xrightarrow{D} Q'$ and $Q \sim Q'$. The second lemma states that, if $V \xrightarrow{D} Q'$ then $V\{b/a\} \xrightarrow{C} P'$ such that $C \sqsubseteq D\{b/a\}$ and $P' = Q'\{b/a\}$. This establishes that $U\{b/a\} \sim V\{b/a\}$.

Suppose that that, the first transition below holds, by select. By the above lemmas, $U\{b/a\} \sim_0 V\{b/a\}$ holds; hence if $U\{b/a\} \xrightarrow{C} P$ then there exists some Q such that $V\{b/a\} \xrightarrow{C} Q$, such that $P \sim_0 Q$. So the second transition below holds and $P \sim_1 Q$, as required.

$$\frac{U\{b/a\} \xrightarrow{C} P}{\bigvee a.U \xrightarrow{C} P} \qquad \text{yields} \qquad \frac{V\{b/a\} \xrightarrow{C} Q}{\bigvee a.V \xrightarrow{C} Q}$$

Hence $\sim_1$ is a bisimulation. A similar argument works for literals.

Consider the case of guards. Assume that $\sim_0$ is a bisimulation. Let $\sim_1$ be the least equivalence relation such that if $U \sim_0 V$ then $U\,;P \sim_1 V\,;P$, if $P \sim_0 Q$ then $P \parr R \sim_1 Q \parr R$, and recursively if $P \sim_1 Q$ then $\bigwedge a.P \sim_1 \bigwedge a.Q$. Assume that $U \sim_0 V$ holds and consider $U\,;P \sim_1 V\,;P$.

Suppose that that, the first transition below holds, by select. Since $U \sim_0 V$, if $U \xrightarrow{C} P$ then there exists some process Q such that $V \xrightarrow{C} Q$ and $P \sim_0 Q$. So the second transition below holds and $P \parr R \sim_1 Q \parr R$.

$$\frac{U \xrightarrow{C} P}{U\,;R \xrightarrow{C} P \parr R} \qquad \text{yields} \qquad \frac{V \xrightarrow{C} Q}{V\,;R \xrightarrow{C} Q \parr R}$$

Hence $\sim_1$ is a bisimulation.

Consider the case of continuations. Assume that $\sim_0$ is a bisimulation. Let $\sim_1$ be the least equivalence relation such that if $P \sim_0 Q$ then $U\,;P \sim_1 U\,;Q$ and $R \parr P \sim_1 R \parr Q$, and recursively if $P \sim_1 Q$ then $\bigwedge a.P \sim_1 \bigwedge a.Q$. Assume that $P \sim_0 Q$ holds and consider $U\,;P \sim_1 U\,;Q$. Assume that $U \xrightarrow{C} R$ hence the following proof trees hold.

$$\frac{U \xrightarrow{C} R}{U\,;P \xrightarrow{C} R \parr P} \qquad \text{and} \qquad \frac{U \xrightarrow{C} R}{U\,;Q \xrightarrow{C} R \parr Q}$$

Furthermore $R \parr P \sim_1 R \parr Q$, hence $\sim_1$ is a bisimulation.

Suppose that, the first transition below holds, by select. Since $U \sim_0 V$, if $U \xrightarrow{C} P$ then there exists some process Q such that $V \xrightarrow{C} Q$ and $P \sim_0 Q$. So the second transition below holds and $P \parr R \sim_1 Q \parr R$.

Consider the case of the exponential. Assume that $\sim_0$ is a bisimulation. Let $\sim_1$ be the least equivalence such that if $U \sim_0 V$ then $*U \sim_1 *V$, if $P \sim_0 Q$ then $P \sim_1 Q$ and, recursively, if both $P_0 \sim_1 Q_0$ and $P_1 \sim_1 Q_1$ then $P_0 \parr P_1 \sim_1 Q_0 \parr Q_1$ and $\bigwedge a.P_0 \sim_1 \bigwedge a.Q_0$. Assume $U \sim_0 V$ and consider the relation $*U \sim_1 *V$. Transitions are due to the weakening, dereliction and contraction rules.

The case of the weakening rule is trivial. If $*U \xrightarrow{\mathrm{I}} \bot$ then $*V \xrightarrow{\mathrm{I}} \bot$ and $\bot \sim_1 \bot$. For the dereliction rule, suppose the first transition below holds. Since $U \sim_0 V$ and $U \xrightarrow{C} P$, there exists a Q such that $V \xrightarrow{C} Q$ and $P \sim_0 Q$. Hence the second transition below holds and $P \sim_1 Q$.

$$\frac{U \xrightarrow{C} P}{*U \xrightarrow{C} P} \qquad \text{yields} \qquad \frac{V \xrightarrow{C} Q}{*V \xrightarrow{C} Q}$$

For contraction, proceed by induction on the derivation of a transition. Suppose that the first transition below holds. By induction, since $*U \xrightarrow{C} P_0$, there exist processes P_0 such $*V \xrightarrow{C} Q_0$ and $P_0 \sim_1 Q_0$. Similarly, since $*U \xrightarrow{D} P_1$, there exist processes Q_1 such $*V \xrightarrow{C} Q_1$ and

$P_1 \sim_1 Q_1$. Hence the second transition below holds and $P_0 \parr P_1 \sim_1 Q_0 \parr Q_1$.

$$\dfrac{*S \xrightarrow{C_0} P_0 \quad *S \xrightarrow{C_1} P_1}{*S \otimes *S \xrightarrow{C_0 \otimes C_1} P_0 \parr P_1}{\scriptstyle *S \xrightarrow{C_0 \otimes C_1} P_0 \parr P_1} \quad \text{yields} \quad \dfrac{*T \xrightarrow{C_0} Q_0 \quad *T \xrightarrow{C_1} Q_1}{*T \otimes *T \xrightarrow{C_0 \otimes C_1} Q_0 \parr Q_1}{\scriptstyle *T \xrightarrow{C_0 \otimes C_1} Q_0 \parr Q_1}$$

Hence by induction over the derivation of an iterated transition, $\sim_1$ is a bisimulation. Note: this case uses both induction for understanding the iterative operation, and coinduction in the form of bisimulation, for understanding the observable operational behaviour.

This covers all cases, hence bisimulation is closed under all contexts. □

4.4.2 Contextual Equivalence and soundness

An alternative notion of equivalence is defined using the reduction system. Contextual equivalence is used in related work to justify notions of bisimulation on the π-calculus and ambient calculus [76, 95].

Definition 4.7 (Contextual equivalence)**.** Contextual equivalence, written $\simeq$, is the greatest symmetric, reduction closed, context closed relation. A relation $\mathcal{R}$ is reduction closed iff $P \mathcal{R} Q$ and $P \rhd P'$ then there exists some Q' such that $Q \rhd Q'$ and $P' \mathcal{R} Q'$. A relation $\mathcal{R}$ is context closed iff $P \mathcal{R} Q$ yields that $CP \mathcal{R} CQ$, for all contexts C.

Bisimulation is sound with respect to contextual equivalence. Soundness is essential to justify the chosen notion of bisimulation.

Theorem 4.8 (Bisimulation is a contextual equivalence)**.** *If $P \sim Q$ then $P \simeq Q$.*

Proof. Reduction closure follows from Lemma 4.3 and context closure follows from Lemma 4.6. □

Soundness of bisimulation ensures that algebraic properties proven using bisimulation also hold for contextual equivalence. Bisimulation simplifies proofs in the following section. Note that completeness (contextual equivalence is a bisimulation) is not required for this work.

Full completeness would be demonstrated by showing that for every label there is a context which makes a transition to a specific state if and only if a particular label holds. For instance for a transition $P \xrightarrow{l} R$ there should be a context C_l that $C_l(P) \rhd P' \parr \checkmark$, where $\checkmark$ is a special process which flags a particular transition. The existence of such a context for each label allows the bisimulation game to be played using the reduction system. Thus a completeness proof must show that: if $P \simeq Q$, then if $C_l(P) \rhd \checkmark \parr P'$ then $C_l(Q) \rhd \checkmark \parr Q'$ such that $P' \simeq Q'$ for all labels l. Thus contextual equivalence is shown to be a bisimulation (the converse of 4.8). This technique has been employed to prove full completeness for the π-calculus and the ambient calculus [95, 100].

Full completeness can only be achieved in an extended calculus. The problem with the current calculus is that queries can only be observed by stored triples, which are static so would not decay to detect a flag (✓ above). Therefore the calculus should be extended with perishable outputs to detect the perishable inputs. A calculus which is symmetric in its inputs and outputs would be a stronger setting for investigating the framework for operational semantics introduced. However, in this work the focus is on the application rather than the framework.

A weak completeness result is Lemma 4.4. Combined with Lemma 4.3, it is demonstrated that the reduction system and the labelled transition system have the same operational power. However, this is not sufficient to demonstrates that two systems give rise to the same algebraic equations.

4.5 A Sound Algebra for Queries

Using bisimulation as an equivalence, properties of queries are established. Firstly a standard bisimulation result for processes is verified. The algebra for queries is then investigated. Queries form a monoid in a sup-lattice. The monoid is the tensor operation, while other features, such as choice, select and iteration are various suprema.

This section amounts to a soundness result. Each result introduces an algebraic property of queries. The bisimulation proof then demonstrates that the algebraic property is sound with respect to the notion of bisimulation. Since bisimulation is sound with respect to structural equivalence, the algebra is sound with respect to structural equivalence.

4.5.1 The structural congruence for processes

For the labelled transition system, structural congruence is not assumed, hence verified here. The proof for the distributivity of blank node quantifiers over par requires extensive case analysis. The case of associativity of par follows from distributivity of blank node quantifiers. Proofs are similar to the analogous bisimulations in the π-calculus [99] (Theorem. 8).

Proposition 4.9 (Structural congruence is a bisimulation)**.** *So,* $(\parr, \bot)$ *forms a commutative monoid, as follows.*

$$(P \parr Q) \parr R \sim P \parr (Q \parr R) \qquad P \parr \bot \sim P \qquad P \parr Q \sim Q \parr P$$

Blank node quantifiers annihilate with $\bot$*, commute, and distribute over* $\parr$*, as follows.*

$$\bigwedge a.\bot \sim \bot \qquad \bigwedge a.\bigwedge b.P \sim \bigwedge b.\bigwedge a.P \qquad \bigwedge a.(P \parr Q) \sim \bigwedge a.P \parr Q \; a \notin \mathrm{fn}(Q)$$

Proof. Consider the case of the empty process in the presence of a blank node quantifier. Both $\bot$ and $\bigwedge a.\bot$ have no transitions so the least equivalence relation $\sim_0$ such that $\bot \sim_0 \bigwedge a.\bot$ is vacuously a bisimulation.

Consider the case of the empty process in the presence of par. Assume that $P \xrightarrow{C} P'$, where C is an input or an output label, hence by the par context rule $P \parr \bot \xrightarrow{C} P' \parr \bot$. Hence the least equivalence relation $\sim_0$ such that $P \parr \bot \sim_0 P$ is a bisimulation.

Consider the case of commuting quantifiers. If $a = b$ then quantifiers immediately commute, so suppose that $a \neq b$. Assume that $P \xrightarrow{\alpha|\overline{C}} P'$ and that $a \notin \alpha$ and $b \notin \alpha$. There are three cases to consider. In the first case the open rule is applied twice, hence the following proof trees are interchangeable.

$$\dfrac{\dfrac{P \xrightarrow{\alpha|\overline{C}} P'}{\bigwedge a.P \xrightarrow{\alpha+a|\overline{C}} P'}}{\bigwedge b.\bigwedge a.P \xrightarrow{\alpha+a+b|\overline{C}} P'} \quad \text{iff} \quad \dfrac{\dfrac{P \xrightarrow{\alpha|\overline{C}} P'}{\bigwedge b.P \xrightarrow{\alpha+b|\overline{C}} P'}}{\bigwedge a.\bigwedge b.P \xrightarrow{\alpha+a+b|\overline{C}} P'}$$

In the second case the open rule and context rule is applied once in either order, hence the following proof trees are interchangeable.

$$\dfrac{\dfrac{P \xrightarrow{\alpha|\overline{C}} P'}{\bigwedge a.P \xrightarrow{\alpha+a|\overline{C}} P'}}{\bigwedge b.\bigwedge a.P \xrightarrow{\alpha+a|\overline{C}} \bigwedge b.P'} \quad \text{iff} \quad \dfrac{\dfrac{P \xrightarrow{\alpha|\overline{C}} P'}{\bigwedge b.P \xrightarrow{\alpha|\overline{C}} \bigwedge b.P'}}{\bigwedge a.\bigwedge b.P \xrightarrow{\alpha+a|\overline{C}} P'}$$

In the third case the context rule is applied twice, hence the following proof trees are interchangeable.

$$\dfrac{\dfrac{P \xrightarrow{\alpha|\overline{C}} P'}{\bigwedge b.P \xrightarrow{\alpha|\overline{C}} \bigwedge b.P'}}{\bigwedge a.\bigwedge b.P \xrightarrow{\alpha|\overline{C}} \bigwedge a.\bigwedge b.P'} \quad \text{iff} \quad \dfrac{\dfrac{P \xrightarrow{\alpha|\overline{C}} P'}{\bigwedge b.P \xrightarrow{\alpha|\overline{C}} \bigwedge b.P'}}{\bigwedge a.\bigwedge b.P \xrightarrow{\alpha|\overline{C}} \bigwedge a.\bigwedge b.P'}$$

Hence the least equivalence relation $\sim_0$ such that $\bigwedge a.\bigwedge b.P \sim_0 \bigwedge b.\bigwedge a.P$ is a bisimulation.

Consider the case of commutativity of par. There are three cases to consider. Firstly, suppose that α_0 and α_1 are disjoint and that $P \xrightarrow{\alpha_0|\overline{C}} P'$ and $Q \xrightarrow{\alpha_1|\overline{D}} Q'$ hold. Therefore the following proof trees are interchangeable.

$$\dfrac{P \xrightarrow{\alpha_0|\overline{C}} P' \quad Q \xrightarrow{\alpha_1|\overline{D}} Q'}{P \parr Q \xrightarrow{\alpha_0+\alpha_1|\overline{C \otimes D}} P' \parr Q'} \quad \text{iff} \quad \dfrac{Q \xrightarrow{\alpha_1|\overline{D}} Q' \quad P \xrightarrow{\alpha_0|\overline{C}} P'}{Q \parr P \xrightarrow{\alpha_0+\alpha_1|\overline{C \otimes D}} Q' \parr P'}$$

Secondly, suppose that $P \xrightarrow{\alpha_0|\overline{C}} P'$ and $Q \xrightarrow{\alpha_1|\overline{D}} Q'$ hold and that α_0 and α_1 are disjoint. Therefore the following proof trees are interchangeable, by the symmetry of the close rule.

$$\frac{P \xrightarrow{C\otimes D} P' \quad Q \xrightarrow{\alpha|\overline{C}} Q'}{P \parr Q \xrightarrow{D} \bigwedge\alpha.(P' \parr Q')} \quad \text{iff} \quad \frac{Q \xrightarrow{\alpha|\overline{C}} Q' \quad P \xrightarrow{C\otimes D} P'}{Q \parr P \xrightarrow{D} \bigwedge\alpha.(Q' \parr P')}$$

Thirdly, suppose that $P \xrightarrow{C} P'$ holds, where C is any input or output label, then the following trees are interchangeable, by the symmetry of the par context rule.

$$\frac{P \xrightarrow{C} P'}{P \parr Q \xrightarrow{C} P' \parr Q} \quad \text{iff} \quad \frac{P \xrightarrow{C} P'}{Q \parr P \xrightarrow{C} Q \parr P'}$$

Hence the least equivalence relation $\sim_0$ such that $P \parr Q \sim_0 Q \parr P$ and, recursively by Lemma 4.6, if $P \sim_0 Q$ then $\bigwedge a.P \sim_0 \bigwedge a.Q$, is a bisimulation.

Consider the case of the change of scope of a blank node quantifier, where $a \notin \mathrm{fn}(Q)$. There are eight cases to consider covering combinations of the combined output rule, the close rule and context rules.

Firstly, consider the two cases for combining output labels. Assume that $P \xrightarrow{\alpha_0|C} P'$ and $Q \xrightarrow{\alpha_1|D} Q'$, where α_0, α_1 and a are pairwise disjoint, $\alpha_0 \cap \mathrm{fn}(Q) = \emptyset$ and $\alpha_1 \cap \mathrm{fn}(P) = \emptyset$. Hence furthermore $(\alpha_0 + a) \cap \mathrm{fn}(Q) = \emptyset$, so the following trees are interchangeable.

$$\frac{\dfrac{P \xrightarrow{\alpha_0|\overline{C}} P' \quad Q \xrightarrow{\alpha_1|\overline{D}} Q'}{P \parr Q \xrightarrow{\alpha_0+\alpha_1|\overline{C\otimes D}} P' \parr Q'}}{\bigwedge a.(P \parr Q) \xrightarrow{\alpha_0+\alpha_1+a|\overline{C\otimes D}} P' \parr Q'} \quad \text{iff} \quad \frac{\dfrac{P \xrightarrow{\alpha_0|\overline{C}} P'}{\bigwedge a.P \xrightarrow{\alpha_0+a|\overline{C}} P' \quad Q \xrightarrow{\alpha_1|\overline{D}} Q'}}{\bigwedge a.P \parr Q \xrightarrow{\alpha_0+\alpha_1+a|\overline{C\otimes D}} P' \parr Q'}$$

Assume also that $a \notin \mathrm{fn}(C)$. By the lemma, if $a \notin \mathrm{fn}(Q) \cup \alpha_1$ and $Q \xrightarrow{\alpha_1|\overline{D}} Q'$ then $a \notin \mathrm{fn}(D)$, it follows that $a \notin \mathrm{fn}(C \otimes D)$. So the following trees are interchangeable.

$$\frac{\dfrac{P \xrightarrow{\alpha_0|\overline{C}} P' \quad Q \xrightarrow{\alpha_1|\overline{D}} Q'}{P \parr Q \xrightarrow{\alpha_0+\alpha_1|\overline{C\otimes D}} P' \parr Q'}}{\bigwedge a.(P \parr Q) \xrightarrow{\alpha_0+\alpha_1|\overline{C\otimes D}} \bigwedge a.(P' \parr Q')} \quad \text{iff} \quad \frac{\dfrac{P \xrightarrow{\alpha_0|\overline{C}} P'}{\bigwedge a.P \xrightarrow{\alpha_0|\overline{C}} \bigwedge a.P' \quad Q \xrightarrow{\alpha_1|\overline{D}} Q'}}{\bigwedge a.P \parr Q \xrightarrow{\alpha_0+\alpha_1|\overline{C\otimes D}} \bigwedge a.P' \parr Q'}$$

Secondly, consider the three cases which use the close rule. Assume that $P\{b/a\} \xrightarrow{\alpha|\overline{C}} P'$ and $Q \xrightarrow{C\otimes D} Q'$ such that $\alpha \cap \mathrm{fn}(Q) = \emptyset$ and $\alpha \cap \mathrm{fn}(D) = \emptyset$. Consider each direction separately, presented below. The forward direction holds since $a \notin \mathrm{fn}(Q)$ so $(\alpha + a) \cap \mathrm{fn}(Q) = \emptyset$. The converse direction holds, by choosing such that $b \notin \mathrm{fn}(\bigwedge a.(P \parr Q), Q')$. Since $a \notin \mathrm{fn}(Q)$ gives $P\{b/a\} \parr Q = (P \parr Q)\{b/a\}$ alpha conversion can be applied. This avoids the worst case that $a \in \mathrm{fn}(Q')$ and $a \in \mathrm{fn}(D)$, where a is offered as an input but trapped as a blank node in the

continuation.

$$\dfrac{\dfrac{P\left\{{}^{b}/_{a}\right\} \xrightarrow{\alpha|\overline{C}} P' \quad Q \xrightarrow{C\otimes D} Q'}{P\left\{{}^{b}/_{a}\right\} \parr Q \xrightarrow{D} \bigwedge\alpha.(P' \parr Q')}}{\bigwedge a.(P \parr Q) \xrightarrow{D} \bigwedge b.\bigwedge\alpha.(P' \parr Q')} \quad \text{iff} \quad \dfrac{\dfrac{P\left\{{}^{b}/_{a}\right\} \xrightarrow{\alpha|\overline{C}} P'}{\bigwedge a.P \xrightarrow{\alpha+b|\overline{C}} P' \quad Q \xrightarrow{C\otimes D} Q'}}{\bigwedge a.P \parr Q \xrightarrow{D} \bigwedge b.\bigwedge\alpha.(P' \parr Q')}$$

A third case, with a different form of continuation to the above, need only be checked in one direction. The argument is similar to above.

$$\dfrac{\dfrac{P\left\{{}^{b}/_{a}\right\} \xrightarrow{\alpha|\overline{C}} P'}{\bigwedge a.P \xrightarrow{\alpha|\overline{C}} \bigwedge b.P' \quad Q \xrightarrow{C\otimes D} Q'}}{\bigwedge a.P \parr Q \xrightarrow{D} \bigwedge\alpha.(\bigwedge b.P' \parr Q')} \quad \text{yields} \quad \dfrac{\dfrac{P\left\{{}^{b}/_{a}\right\} \xrightarrow{\alpha|\overline{C}} P' \quad Q \xrightarrow{C\otimes D} Q'}{P\left\{{}^{b}/_{a}\right\} \parr Q \xrightarrow{D} \bigwedge\alpha.(P' \parr Q')}}{\bigwedge a.(P \parr Q) \xrightarrow{D} \bigwedge b.\bigwedge\alpha.(P' \parr Q')}$$

Now, assume that $P \xrightarrow{C\otimes D} P'$ and $Q \xrightarrow{\alpha|C} Q'$ such that $\alpha \cap \mathrm{fn}(P, D) = \emptyset$. In both cases $a \notin \mathrm{fn}(D)$, so the following proof trees are interchangeable.

$$\dfrac{\dfrac{P \xrightarrow{C\otimes D} P' \quad Q \xrightarrow{\alpha|\overline{C}} Q'}{P \parr Q \xrightarrow{D} \bigwedge\alpha.(P' \parr Q')}}{\bigwedge a.(P \parr Q) \xrightarrow{D} \bigwedge a.\bigwedge\alpha.(P' \parr Q')} \quad \text{iff} \quad \dfrac{\dfrac{P \xrightarrow{C\otimes D} P'}{\bigwedge a.P \xrightarrow{C\otimes D} \bigwedge a.P' \quad Q \xrightarrow{\alpha|\overline{C}} Q'}}{\bigwedge a.P \parr Q \xrightarrow{D} \bigwedge\alpha.(\bigwedge a.P' \parr Q')}$$

Thirdly, there are three cases for the par context rule. Assume that $P \xrightarrow{\alpha|\overline{C}} P'$ such that $a \notin \mathrm{fn}(C) \cup \alpha$. Hence the following proof trees are interchangeable.

$$\dfrac{\dfrac{P \xrightarrow{\alpha|\overline{C}} P'}{P \parr Q \xrightarrow{\alpha|\overline{C}} P' \parr Q}}{\bigwedge a.(P \parr Q) \xrightarrow{\alpha|\overline{C}} \bigwedge a.(P' \parr Q)} \quad \text{iff} \quad \dfrac{\dfrac{P \xrightarrow{\alpha|\overline{C}} P'}{\bigwedge a.P \xrightarrow{\alpha|\overline{C}} \bigwedge a.P'}}{\bigwedge a.P \parr Q \xrightarrow{\alpha|\overline{C}} \bigwedge a.P' \parr Q}$$

Under the same conditions as above the following proof trees are interchangeable.

$$\dfrac{\dfrac{P \xrightarrow{\alpha|\overline{C}} P'}{P \parr Q \xrightarrow{\alpha|\overline{C}} P' \parr Q}}{\bigwedge a.(P \parr Q) \xrightarrow{\alpha+a|\overline{C}} P' \parr Q} \quad \text{iff} \quad \dfrac{\dfrac{P \xrightarrow{\alpha|\overline{C}} P'}{\bigwedge a.P \xrightarrow{\alpha+a|\overline{C}} P'}}{\bigwedge a.P \parr Q \xrightarrow{\alpha+a|\overline{C}} P' \parr Q}$$

Now, assume that $Q \xrightarrow{\alpha|\overline{C}} Q'$ and b are such that $\alpha \cap \mathrm{fn}(P) = \emptyset$ and $b \notin \mathrm{fn}(Q') \cup \alpha$. Hence the following proof trees are interchangeable.

$$\dfrac{\dfrac{Q \xrightarrow{\alpha|\overline{C}} Q'}{P\left\{{}^{b}/_{a}\right\} \parr Q \xrightarrow{\alpha|\overline{C}} P\left\{{}^{b}/_{a}\right\} \parr Q'}}{\bigwedge a.(P \parr Q) \xrightarrow{\alpha|\overline{C}} \bigwedge b.(P\left\{{}^{b}/_{a}\right\} \parr Q')} \quad \text{iff} \quad \dfrac{Q \xrightarrow{\alpha|\overline{C}} Q'}{\bigwedge a.P \parr Q \xrightarrow{\alpha|\overline{C}} \bigwedge a.P \parr Q'}$$

Hence the least congruence $\sim_0$ which contains alpha conversion, such that if $a \notin \mathrm{fn}(Q)$ then $\bigwedge a.P \parr Q \sim_0 \bigwedge a.(P \parr Q)$ and $\bigwedge a.\bigwedge b.P \sim_0 \bigwedge b.\bigwedge a.P$ is a bisimulation.

Consider the case of associativity. There are fourteen cases. Only the two most involved cases are considered, since the remaining cases present no further problems.

Assume that $P \xrightarrow{\alpha_0|\overline{C}} P'$, $Q \xrightarrow{\alpha_1|\overline{D}} Q'$ and $R \xrightarrow{C\otimes D\otimes E} R'$ hold such that $\alpha_0 \cap \mathrm{fn}(Q,R) = \emptyset$ and $\alpha_1 \cap \mathrm{fn}(R) = \emptyset$. Consider the proof trees below. The forwards direction follows immediately. The converse direction holds since, by alpha conversion, α_1 can be chosen such that $\alpha_1 \cap \mathrm{fn}(P) = \emptyset$ and $\alpha_0 \cap \alpha_1 = \emptyset$. Hence, since also $P \xrightarrow{\alpha_0|\overline{C}} P'$ is follows that $\alpha_1 \cap \mathrm{fn}(P') = \emptyset$.

Hence the following trees are interchangeable, up to alpha conversion of α_1.

$$\dfrac{\dfrac{P \xrightarrow{\alpha_0|\overline{C}} P' \quad Q \xrightarrow{\alpha_1|\overline{D}} Q'}{P \parr Q \xrightarrow{\alpha_0+\alpha_1|\overline{C\otimes D}} P' \parr Q' \quad R \xrightarrow{C\otimes D\otimes E} R'}}{(P \parr Q) \parr R \xrightarrow{E} \bigwedge\alpha_0.\bigwedge\alpha_1.((P' \parr Q') \parr R')}$$

$$\text{iff} \qquad \dfrac{P \xrightarrow{\alpha_0'|\overline{C}} P' \quad \dfrac{Q \xrightarrow{\alpha_1|\overline{D}} Q' \quad R \xrightarrow{C\otimes D\otimes E} R'}{Q \parr R \xrightarrow{C\otimes E} \bigwedge\alpha_1.(Q' \parr R')}}{P \parr (Q \parr R) \xrightarrow{E} \bigwedge\alpha_0.(P' \parr \bigwedge\alpha_1.(Q' \parr R'))}$$

Consider a second case. Assume that $P \xrightarrow{\alpha_0|\overline{C}} P'$, $Q \xrightarrow{\alpha_1|\overline{D}} Q'$ and $R \xrightarrow{C\otimes D\otimes E} R'$ hold such that $(\alpha_0 \cup \alpha_1) \cap \mathrm{fn}(Q) = \emptyset$.

Consider the forwards implication presented below. By alpha conversion, α_0 can be chosen such that $\alpha_0 \cap \mathrm{fn}(R) = \emptyset$ and $\alpha_0 \cap \alpha_1 = \emptyset$. Furthermore, since $\alpha_1 \cap \mathrm{fn}(P) = \emptyset$ and $\alpha_0 \cap \alpha_1 = \emptyset$ hold then $P \xrightarrow{\alpha_0|C} P'$ yields $\alpha_1 \cap \mathrm{fn}(P') = \emptyset$.

The reverse implication is symmetric. By alpha conversion, α_1 can be chosen such that $\alpha_1 \cap P = \emptyset$ and $\alpha_0 \cap \alpha_1 = \emptyset$. Furthermore, since $\alpha_0 \cap \mathrm{fn}(Q) = \emptyset$ and $\alpha_0 \cap \alpha_1 = \emptyset$ hold then $Q \xrightarrow{\alpha_1|D} Q'$ yields $\alpha_0 \cap \mathrm{fn}(Q') = \emptyset$.

Hence, up to the alpha conversion of α_0 and α_1, the following proof trees are interchangeable.

$$\dfrac{\dfrac{P \xrightarrow{\alpha_0|\overline{C}} P' \quad Q \xrightarrow{C\otimes D\otimes E} Q'}{P \parr Q \xrightarrow{D\otimes E} \bigwedge\alpha_0.(P' \parr Q') \quad R \xrightarrow{\alpha_1|\overline{D}} R'}}{(P \parr Q) \parr R \xrightarrow{E} \bigwedge\alpha_1.(\bigwedge\alpha_0.(P' \parr Q') \parr R')}$$

$$\text{iff} \qquad \dfrac{P \xrightarrow{\alpha_0|\overline{C}} P' \quad \dfrac{Q \xrightarrow{C\otimes D\otimes E} Q' \quad R \xrightarrow{\alpha_1|\overline{D}} R'}{Q \parr R \xrightarrow{C\otimes E} \bigwedge\alpha_1.(Q' \parr R')}}{P \parr (Q \parr R) \xrightarrow{E} \bigwedge\alpha_0.(P' \parr \bigwedge\alpha_1.(Q' \parr R'))}$$

Hence the least congruence $\sim_0$ which contains alpha conversion, such that $(P \parr Q) \parr R \sim_0 P \parr (Q \parr R)$, if $a \notin \mathrm{fn}(Q)$ then $\bigwedge a.P \parr Q \sim_0 \bigwedge a.(P \parr Q)$, $P \parr Q \sim_0 Q \parr P$ and $\bigwedge a.\bigwedge b.P \sim_0 \bigwedge b.\bigwedge a.P$ is a bisimulation. □

4.5.2 The semiring of queries

Bisimulation reveals some canonical algebraic properties of queries. Firstly, queries form an commutative idempotent semiring. Semirings are ubiquitous in computer science. A notable feature of semirings is that the ideals of a semiring form a semiring. Commutativity refers to the tensor product, which is not necessarily commutative [71].

Proposition 4.10 (Queries form a semiring)**.** $(U, \otimes, \mathrm{I})$ *is a commutative monoid, as follows.*

$$(U \otimes V) \otimes W \sim (U \otimes V) \otimes W \qquad U \otimes V \sim V \otimes U \qquad U \otimes \mathrm{I} \sim U$$

$(U, \oplus, 0)$ *is idempotent commutative monoid, as follows.*

$$U \oplus U \sim U \qquad (U \oplus V) \oplus W \sim U \oplus (V \oplus W) \qquad U \oplus V \sim V \oplus U \qquad U \oplus 0 \sim U$$

Furthermore, $\otimes$ *distributes over* $\oplus$ *and* 0 *annihilates with* $\otimes$*, as follows.*

$$U \otimes (V \oplus W) \sim (U \otimes V) \oplus (U \otimes W) \qquad U \otimes 0 \sim 0$$

Hence, $(U, \otimes, \oplus, \mathrm{I}, 0)$ *is a commutative idempotent semiring.*

Proof. Each case is verified by rewriting the derivation of input transitions.

Consider the case of idempotency of choice. Assume that $U \xrightarrow{C} P$ and, without loss of generality, the left branch is chosen. The following two proof trees are interchangeable.

$$\frac{U \xrightarrow{C} P}{U \oplus U \xrightarrow{C} P} \qquad \text{iff} \qquad U \xrightarrow{C} P$$

Hence the least equivalence relation $\sim_0$ such that $U \oplus U \sim_0 U$ is a bisimulation.

Consider the case of associativity of choice. There are three cases to consider. If $U \xrightarrow{C} P$ holds then the following trees are interchangeable.

$$\frac{\dfrac{U \xrightarrow{C} P}{U \oplus V \xrightarrow{C} P}}{(U \oplus V) \oplus W \xrightarrow{C} P} \qquad \text{iff} \qquad \frac{U \xrightarrow{C} P}{U \oplus (V \oplus W) \xrightarrow{C} P}$$

If $V \xrightarrow{D} Q$ then the following trees are interchangeable.

$$\frac{\dfrac{V \xrightarrow{D} Q}{U \oplus V \xrightarrow{D} Q}}{(U \oplus V) \oplus W \xrightarrow{D} Q} \qquad \text{iff} \qquad \frac{\dfrac{V \xrightarrow{D} Q}{V \oplus W \xrightarrow{D} Q}}{U \oplus (V \oplus W) \xrightarrow{D} Q}$$

The third case is symmetric to the first case. Hence the least equivalence relation such that $U \oplus (V \oplus W) \sim_0 (U \oplus V) \oplus W$ is a bisimulation.

Consider the case of commutativity of choice. Assuming that $U \xrightarrow{C} P$ holds, then following proof trees are interchangeable.

$$\frac{U \xrightarrow{C} P}{U \oplus V \xrightarrow{C} P} \quad \text{iff} \quad \frac{U \xrightarrow{C} P}{V \oplus U \xrightarrow{C} P}$$

Hence the least equivalence relation $\sim_0$ such that $U \oplus V \sim_0 V \oplus U$ is a bisimulation.

Consider the case of the unit of choice. Since $\not\models 0$ never holds, 0 never makes a transition. Hence there is only one possible choice and the following proof trees are interchangeable.

$$\frac{U \xrightarrow{C} P}{U \oplus 0 \xrightarrow{C} P} \quad \text{iff} \quad U \xrightarrow{C} P$$

Hence the least equivalence relation $\sim_0$ such that $U \oplus 0 \sim_0 U$ is a bisimulation.

Consider the case of associativity of tensor. Assuming that $U \xrightarrow{C} P$, $V \xrightarrow{D} Q$ and $W \xrightarrow{E} R$, the following proof trees are interchangeable.

$$\frac{U \xrightarrow{C} P \quad \dfrac{V \xrightarrow{D} Q \quad W \xrightarrow{E} R}{V \otimes W \xrightarrow{D \otimes E} Q \parr R}}{U \otimes (V \otimes W) \xrightarrow{C \otimes D \otimes E} P \parr (Q \parr R)} \quad \text{iff} \quad \frac{\dfrac{U \xrightarrow{C} P \quad V \xrightarrow{D} Q}{U \otimes V \xrightarrow{C \otimes D} P \parr Q} \quad W \xrightarrow{E} R}{(U \otimes V) \otimes W \xrightarrow{C \otimes D \otimes E} (P \parr Q) \parr R}$$

Let $\sim_0$ be the least congruence such that $U \otimes (V \otimes W) \sim_0 (U \otimes V) \otimes W$ and $P \parr (Q \parr R) \sim_0 (P \parr Q) \parr R$, and the conditions for associativity in Proposition 4.9 hold. Hence $\sim_0$ is a bisimulation.

Consider the case of commutativity of tensor. Assume that $U \xrightarrow{C} P$ and $V \xrightarrow{D} Q$ hold. By commutativity of the labels, the following proof trees are interchangeable.

$$\frac{U \xrightarrow{C} P \quad V \xrightarrow{D} Q}{U \otimes V \xrightarrow{C \otimes D} P \parr Q} \quad \text{iff} \quad \frac{V \xrightarrow{D} Q \quad U \xrightarrow{C} P}{V \otimes U \xrightarrow{C \otimes D} Q \parr P}$$

Let $\sim_0$ be the least congruence $\sim_0$ such that $V \otimes U \sim_0 U \otimes V$ and $P \parr Q \sim_0 Q \parr P$. Hence, by Proposition 4.9 and the above, $\sim_0$ is a bisimulation.

Consider the unit of tensor. Assume that $U \xrightarrow{C} P$ and note that $\models \mathrm{I}$ always holds. Hence the following proof trees are interchangeable, by the unit of labels.

$$\frac{U \xrightarrow{C} P \quad \mathrm{I} \xrightarrow{\mathrm{I}} \bot}{U \otimes \mathrm{I} \xrightarrow{C \otimes \mathrm{I}} P \parr \bot} \quad \text{iff} \quad U \xrightarrow{C \otimes \mathrm{I}} P$$

Hence the least equivalence relation $\sim_0$ such that $U \otimes \mathrm{I} \sim_0 U$ and $P \parr \bot \sim_0 P$, by Proposition 4.9, is a bisimulation.

Consider the case of distributivity. Without loss of generality, assume that $U \xrightarrow{C} P$ and $V \xrightarrow{D} Q$. The following proof trees are interchangeable.

$$\dfrac{U \xrightarrow{C} P \quad \dfrac{V \xrightarrow{D} Q}{V \oplus W \xrightarrow{D} Q}}{U \otimes (V \oplus W) \xrightarrow{C \otimes D} P \parr Q} \quad \text{iff} \quad \dfrac{\dfrac{U \xrightarrow{C} P \quad V \xrightarrow{D} Q}{U \otimes V \xrightarrow{C \otimes D} P \parr Q}}{(U \otimes V) \oplus (U \otimes W) \xrightarrow{C \otimes D} P \parr Q}$$

Therefore the least equivalence relation $\sim_0$, such that $U \otimes (V \oplus W) \sim_0 (U \otimes V) \oplus (U \otimes W)$ is a bisimulation.

Consider the case of annihilation. Since $\not\vDash 0$ never holds, then 0 makes no transitions. Now, suppose that $U \otimes 0$ makes a transition. Then $U \xrightarrow{C} P$ and $0 \xrightarrow{D} Q$, for some D and Q, but no such D or Q exist, yielding a contradiction. Hence, the least relation $\sim_0$ such that $U \otimes 0 \sim_0 0$ is a bisimulation. □

Idempotent semirings have a natural preorder, given by $S \leq T$ iff $S \oplus T \sim T$. Hence queries have this natural preorder. It is easy to check that this preorder becomes a partial order when queries are quotiented by bisimulation.

Proposition 4.11. *$\leq$ is a partial order over queries quotiented by bisimulation.*

Proof. Reflexivity follows by idempotency of choice. Transitivity follows since given $U \leq V$ and $V \leq W$, clearly $W \sim W \oplus (V \oplus U) \sim W \oplus U$, thus $U \leq W$. Reflexivity follows since $U \leq V$ and $V \leq U$ yields that $U \sim U \oplus V \sim V$. □

An immediate consequence of Proposition 4.10 is that zero is the least query, since $0 \oplus U \sim U$ defines $0 \leq U$. Another consequence is that choice is a colimit, as follows. A colimit is a least upper bound of queries, so is expressed using the natural partial order over queries.

Proposition 4.12. *Choice is a colimit of its branches. That is, $V \leq W$ and $U \leq W$, if and only if $V \oplus U \leq W$.*

Proof. Assume that $U \leq W$ and $V \leq W$. Hence the following reasoning holds.

$$\begin{aligned} (U \oplus V) \oplus W &\sim (U \oplus W) \oplus (V \oplus W) && \text{by distributivity} \\ &\sim W \oplus W && \text{by the assumptions} \\ &\sim W && \text{by idempotency} \end{aligned}$$

Conversely, assume that $U \oplus V \leq W$. Hence the following reasoning holds.

$$\begin{aligned} U \oplus W &\sim U \oplus U \oplus V \oplus W && \text{by the assumption} \\ &\sim U \oplus V \oplus W && \text{by idempotency} \\ &\sim W && \text{by the assumption} \end{aligned}$$

A symmetric proof works for $V \leq W$. Hence choice is a colimit. □

Another immediate consequence of Proposition 4.10 is that choice and tensor are monotone operators. Monotonicity demonstrates that the natural order is preserved by the operators.

Corollary 4.13. *Choice and tensor are monotone operators.*

Proof. Assuming that $U \leq V$, the following establishes that $W \otimes U \leq W \otimes V$.

$$\begin{array}{lll} (W \otimes U) \oplus (W \otimes V) & \sim W \otimes (U \oplus V) & \text{by distributivity} \\ & \sim W \otimes V & \text{by the assumption} \end{array}$$

Similarly, assuming that $U \leq V$, the following establishes that $W \oplus U \leq W \oplus V$.

$$\begin{array}{lll} W \oplus U \oplus W \oplus V & \sim W \oplus U \oplus V & \text{by idempotency} \\ & \sim W \oplus V & \text{by the assumption} \end{array}$$

□

The preorder over queries can be used to optimise queries. If a query offers a choice between a query and less deterministic query, the more deterministic branch may be eliminated. For instance, in related work [110], is claimed that the follows to queries are not the same.

$$U \texttt{ OPTIONAL } (V \texttt{ OPTIONAL } W) \quad \text{and} \quad (U \texttt{ OPTIONAL } V) \texttt{ OPTIONAL } W$$

Under the interpretation of `OPTIONAL` in the calculus the following holds by distributivity, commutativity and idempotency.

$$U \otimes ((V \otimes (W \oplus \mathrm{I})) \oplus \mathrm{I}) \leq U \otimes ((V \oplus \mathrm{I}) \otimes (W \oplus \mathrm{I}))\,,$$

Hence the first query is more deterministic than the second query, so answers fewer questions.

4.5.3 The select quantifiers as colimits

A single rule is sufficient to capture the algebra of the select quantifier. From this algebra common equalities can be derived. The derived rules are suitable for the optimisation technique of flattening nested selects used in relational algebra [43]. The proof of commutativity of quantifiers requires the notion of capture avoiding substitution to be assumed. Capture avoiding substitution is a weaker assumption than alpha conversion. The presence of the tensor in the rule is required to prove that $\bigvee a.U \otimes V \leq \bigvee a.(U \otimes V)$, when $a \notin \mathrm{fn}\,(V)$. This

Proposition 4.14. *Selects are colimits of substitutions. So,* $U\left\{{}^{b}/_{a}\right\} \otimes V \leq W$ *for all b, if and only if* $\bigvee a.U \otimes V \leq W$.

Proof. Assume that $U \xrightarrow{C} P$ and $V \xrightarrow{D} Q$ holds. Hence the following proof trees are interchangeable.

$$\frac{U\left\{{}^{b}/_{a}\right\} \xrightarrow{C} P \quad V \xrightarrow{D} Q}{U\left\{{}^{b}/_{a}\right\} \otimes V \xrightarrow{C \otimes D} P \parr Q} \qquad \text{iff} \qquad \frac{\dfrac{U\left\{{}^{b}/_{a}\right\} \xrightarrow{C} P}{\bigvee a.U \xrightarrow{C} P} \quad V \xrightarrow{D} Q}{\bigvee a.U \otimes V \xrightarrow{C \otimes D} P \parr Q}$$

Now assume that $(U\left\{{}^{b}/_{a}\right\} \otimes V) \oplus W \sim W$ for all b and consider $(\bigvee a.U \otimes V) \oplus W$. If the left branch is chosen then $\bigvee a.U \otimes V \xrightarrow{C \otimes D} P \parr Q$. Hence, by the above, $U\left\{{}^{b}/_{a}\right\}U \xrightarrow{C \otimes D} P \parr Q$ for some b. Thus by the bisimulation assumption there exists some R such that $W \xrightarrow{C \otimes D} R$ and $R \sim P \parr Q$. Therefore $\bigvee a.U \otimes V \leq W$.

Conversely, assume that $(\bigvee a.U \otimes V) \oplus W \sim W$ and consider $(U\left\{{}^{b}/_{a}\right\} \otimes V) \oplus W$. If the left branch is chosen then $U\left\{{}^{b}/_{a}\right\} \otimes V \xrightarrow{C \otimes D} P \parr Q$. Hence by the above, $\bigvee a.U \otimes V \xrightarrow{C \otimes D} P \parr Q$. Thus by the bisimulation assumption there exists some R such that $W \xrightarrow{C \otimes D} R$ and $R \sim P \parr Q$. Therefore $U\left\{{}^{b}/_{a}\right\} \otimes V \leq W$. □

Existential quantifiers have been known to be colimits since the work of Lawvere [88]. Lawvere shows that existential quantification is left adjoint to capture avoiding substitution; while universal quantification is right adjoint to capture avoiding substitution. This elegant definition allows all other algebraic properties of quantifiers to be derived. Thus common properties of select quantifiers can be established easily using the above proposition. These include the dual properties to those established for blank node quantifiers with respect to the monoid of parallel processes. For the select quantifier these properties are established with respect to the monoid of synchronous queries.

Corollary 4.15. *Immediate consequences are that, select commutes, distributes over choice, is annihilated by true and distributes over tensor. Furthermore, alpha conversion of bound variables is verified.*

$$\bigvee a.\bigvee b.S \sim \bigvee b.\bigvee a.S \qquad \bigvee a.(S \oplus T) \sim \bigvee a.S \oplus \bigvee a.T$$
$$\bigvee a.\mathrm{I} \sim \mathrm{I} \qquad \bigvee a.(S \otimes T) \sim \bigvee a.S \otimes T \quad a \notin \mathrm{fn}(T)$$

Proof. Consider the case of a quantified unit. For all substitutions $\mathrm{I}\left\{{}^{b}/_{a}\right\} = \mathrm{I}$, so $\mathrm{I} \leq U$ if and only if $\bigvee a.\mathrm{I} \leq U$, since select is a colimit. By taking U to be I, it holds that $\bigvee a.\mathrm{I} \leq \mathrm{I}$. Conversely, by taking U to be $\bigvee a.\mathrm{I}$ it holds that $\mathrm{I} \leq \bigvee a.\mathrm{I}$.

Consider the distributivity of select quantifiers over choice. The following reasoning demonstrates the exchange of choice and select, since $U\left\{{}^{b}/_{a}\right\} \oplus V\left\{{}^{b}/_{a}\right\} = (U \oplus V)\left\{{}^{b}/_{a}\right\}$.

$$\begin{array}{lll}
\bigvee a.U \oplus \bigvee a.V \leq W & \text{iff } \bigvee a.U \leq W \text{ and } \bigvee a.V \leq W & \text{since choice is a colimit} \\
& \text{iff } U\left\{{}^{b}/_{a}\right\} \leq W \text{ and } V\left\{{}^{b}/_{a}\right\} \leq W \text{ for all } b & \text{since select is a colimit} \\
& \text{iff } (U \oplus V)\left\{{}^{b}/_{a}\right\} \leq W \text{ for all } b & \text{since choice is a colimit} \\
& \text{iff } \bigvee a.(U \oplus V) \leq W & \text{since select is a colimit}
\end{array}$$

Hence by taking W to be $\bigvee a.(U \oplus V)$ it follows that $\bigvee a.U \oplus \bigvee a.V \leq \bigvee a.(U \oplus V)$. Similarly, by taking W to be $\bigvee a.U \oplus \bigvee a.V$ it follows that $\bigvee a.(U \oplus V) \leq \bigvee a.U \oplus \bigvee a.V$.

Consider the case of distributivity of select over tensor. Assume that $a \notin \mathrm{fn}(V)$, so $U\left\{{}^{b}/_{a}\right\} \otimes V = (U \otimes V)\left\{{}^{b}/_{a}\right\}$. Hence the following reasoning holds.

$$\begin{array}{lll} \bigvee a.U \otimes V \leq W & \text{iff } U\left\{{}^{b}/_{a}\right\} \otimes V \leq W & \text{since select is a colimit} \\ & \text{iff } \mathrm{I} \otimes (U \otimes V)\left\{{}^{b}/_{a}\right\} \leq W & \text{since } a \notin \mathrm{fn}(V) \\ & \text{iff } \mathrm{I} \otimes \bigvee a.(U \otimes V) \leq W & \text{since select is a colimit} \end{array}$$

So, by taking W to be $\bigvee a.U \otimes V$ it follows that $\bigvee a.(U \otimes V) \leq \bigvee a.U \otimes V$. Similarly, by taking W to be $\bigvee a.(U \otimes V)$ is follows that $\bigvee a.U \otimes V \leq \bigvee a.(U \otimes V)$.

Consider the commutativity of select quantifiers. If the names are identical then the quantifiers trivially commute. Suppose that the bound names a, b are distinct, so $a \neq b$.

To proceed, two intuitive assumptions for substitutions are made. Firstly, substitutions using a fresh name can be composed, so for any $\{{}^{e}/_{a}\}\left\{{}^{b}/_{e}\right\} = \left\{{}^{b}/_{a}\right\}$, where e is fresh. Secondly, independent substitutions commute, so if e, f are distinct names then $\{{}^{c}/_{e}\}\left\{{}^{d}/_{f}\right\} = \left\{{}^{d}/_{f}\right\}\{{}^{c}/_{e}\}$. So, assuming that e, f are distinct fresh names the following holds.

$$\begin{array}{lll} U\left\{{}^{f}/_{b}\right\}\{{}^{c}/_{a}\}\left\{{}^{d}/_{f}\right\} & = U\left\{{}^{f}/_{b}\right\}\{{}^{e}/_{a}\}\{{}^{c}/_{e}\}\left\{{}^{d}/_{f}\right\} & \text{by composition of substitutions} \\ & = U\{{}^{e}/_{a}\}\left\{{}^{f}/_{b}\right\}\left\{{}^{d}/_{f}\right\}\{{}^{c}/_{e}\} & \text{by commutivity of substitutions} \\ & = U\{{}^{e}/_{a}\}\left\{{}^{d}/_{b}\right\}\{{}^{c}/_{e}\} & \text{by composition of substitutions} \end{array}$$

Furthermore, since $a \neq b$, $(\bigvee b.U)\{{}^{c}/_{a}\} = \bigvee f.(U\left\{{}^{f}/_{b}\right\}\{{}^{c}/_{a}\})$, by capture avoiding substitution and similarly, $(\bigvee a.U)\left\{{}^{d}/_{b}\right\} = \bigvee e.(U\{{}^{e}/_{a}\}\left\{{}^{d}/_{b}\right\})$. Hence the following reasoning holds.

$$\begin{array}{lll} \bigvee a.\bigvee b.U \leq V & \text{iff } (\bigvee b.U)\{{}^{c}/_{a}\} \leq V \text{ for all } c & \text{since select is a colimit} \\ & \text{iff } \bigvee f.(U\left\{{}^{f}/_{b}\right\}\{{}^{c}/_{a}\}) \leq V \text{ for all } c & \text{by capture avoiding substitution} \\ & \text{iff } U\left\{{}^{f}/_{b}\right\}\{{}^{c}/_{a}\}\left\{{}^{d}/_{f}\right\} \leq V \text{ for all } c, d & \text{since select is a colimit} \\ & \text{iff } U\{{}^{e}/_{a}\}\left\{{}^{d}/_{b}\right\}\{{}^{c}/_{e}\} \leq V \text{ for all } c, d & \text{by the above lemma} \\ & \text{iff } \bigvee e.(U\{{}^{e}/_{a}\}\left\{{}^{d}/_{b}\right\}) \leq V \text{ for all } d & \text{since select is a colimit} \\ & \text{iff } (\bigvee a.U)\left\{{}^{d}/_{b}\right\} \leq V \text{ for all } d & \text{by capture avoiding substitution} \\ & \text{iff } \bigvee b.\bigvee a.U \leq V & \text{since select is a colimit} \end{array}$$

So, by taking V to be $\bigvee b.\bigvee a.U$ it holds that $\bigvee a.\bigvee b.U \leq \bigvee b.\bigvee a.U$. Hence, by symmetry of argument, $\bigvee a.\bigvee b.U \sim \bigvee b.\bigvee a.U$.

Capture avoiding substitution (as assumed for the previous case) is a weaker assumption than alpha conversion. Alpha conversion of select quantifiers can be derived as follows. Assume that

$U\{^c/_a\} \xrightarrow{C} P$ and note that $U\{^c/_a\} = U\left\{^b/_a\right\}\{^c/_b\}$, where b is fresh.

$$\begin{array}{lll} \bigvee a.U \leq V & \text{iff } U\{^c/_a\} \leq V & \text{since select is a colimit} \\ & \text{iff } U\left\{^b/_a\right\}\{^c/_b\} \leq V & \text{since } b \text{ is fresh} \\ & \text{iff } \bigvee b.(U\left\{^b/_a\right\}) \leq V & \text{since select is a colimit} \end{array}$$

Thus by taking V to be $\bigvee a.U$ is follows that $\bigvee b.(U\left\{^b/_a\right\}) \leq \bigvee a.U$. Similarly, by taking V to be $\bigvee b.(U\left\{^b/_a\right\})$ if follows that $\bigvee a.U \leq \bigvee b.(U\left\{^b/_a\right\})$. Thus alpha conversion is a bisimulation. □

Note that the above corollary also ensures that select quantification is monotone. Assume $P \leq Q$ and consider the select quantifiers.

$$\begin{array}{lll} \bigvee a.P \oplus \bigvee a.Q & \sim \bigvee a.(P \oplus Q) & \text{by Corollary 4.15} \\ & \sim \bigvee a.(Q) & \text{by the assumption} \end{array}$$

Hence $\bigvee a.P \leq \bigvee a.Q$ by the reasoning above. Monotonicity of operators is used to establish further results.

4.5.4 The algebra of iteration

The following rules of regular algebra hold. The first of the rules is sufficient to demonstrate that $*V \otimes U$ is a fixed point of the (monotone) map $W \mapsto U \oplus (V \otimes W)$. The second rule demonstrates that $*V \otimes U$ is the least such fixed point. The formulation below, was proven to be complete, with respect to the equational theory of Kleene algebras, by Kozen [83].

Proposition 4.16. *An iterated query expands as follows* $*U \sim \mathrm{I} \oplus (U \otimes *U)$. *Furthermore, if* $U \oplus (V \otimes W) \leq W$ *then* $*V \otimes U \leq W$.

Proof. The derivation of transitions of $*U$ can be normalised by reorganising the derivation. A contraction on the left after another contraction is converted to a contraction on the right, as follows.

$$\dfrac{\dfrac{\dfrac{*U \xrightarrow{C} P \quad *U \xrightarrow{D} Q}{*U \otimes *U \xrightarrow{C\otimes D} P \parr Q} \qquad *U \xrightarrow{E} R}{*U \otimes *U \xrightarrow{C\otimes D\otimes E} (P \parr Q) \parr R}}{*U \xrightarrow{C\otimes D\otimes E} (P \parr Q) \parr R} \quad \text{yields} \quad \dfrac{*U \xrightarrow{C} P \qquad \dfrac{\dfrac{*U \xrightarrow{D} Q \quad *U \xrightarrow{E} R}{*U \otimes *U \xrightarrow{D\otimes E} Q \parr R}}{*U \otimes *U \xrightarrow{C\otimes D\otimes E} P \parr (Q \parr R)}}{*U \xrightarrow{C\otimes D\otimes E} P \parr (Q \parr R)}$$

If weakening appears on the left of a contraction then the contraction can be removed.

$$\dfrac{\dfrac{*U \xrightarrow{\mathrm{I}} \bot \quad *U \xrightarrow{C} P}{*U \otimes *U \xrightarrow{\mathrm{I}\otimes C} \bot \parr P}}{*U \xrightarrow{\mathrm{I}\otimes C} \bot \parr P} \quad \text{yields} \quad *U \xrightarrow{C} P$$

By applying the above two transformation recursively there are three possible outcomes. The three outcomes correspond to the transitions of $\mathrm{I} \oplus (U \otimes *U)$. The first case is that a transition reduces to a single weakening rule, which is matched by choosing the unit, as follows.

$$*U \xrightarrow{\mathrm{I}} \bot \qquad \text{iff} \qquad \frac{\mathrm{I} \xrightarrow{\mathrm{I}} \bot}{\mathrm{I} \oplus (U \otimes *U) \xrightarrow{\mathrm{I}} \bot}$$

The second case is that the transformation results in a dereliction on the left, as follows, since both contraction and weakening are removed by normalisation.

$$\dfrac{\dfrac{\dfrac{U \xrightarrow{C} P}{*U \xrightarrow{C} P} \quad *U \xrightarrow{D} Q}{*U \otimes *U \xrightarrow{C \otimes D} P \parr Q}}{*U \xrightarrow{C \otimes D} P \parr Q} \qquad \text{yields} \qquad \dfrac{\dfrac{U \xrightarrow{C} P \quad *U \xrightarrow{D} Q}{U \otimes *U \xrightarrow{C \otimes D} P \parr Q}}{\mathrm{I} \oplus (U \otimes *U) \xrightarrow{C \otimes D} P \parr Q}$$

The third case is that the exponential is derived directly from dereliction. By applying contraction and weakening on the right, the third case is reduced to the second case.

Therefore the least equivalence relation $\sim_0$ such that $*U \sim_0 \mathrm{I} \oplus (U \otimes *U)$ and $(P \parr Q) \parr R \sim_0 P \parr (Q \parr R)$ and $\bot \parr P \sim_0 P$ is a bisimulation. This established the expansion property.

The expansion property is sufficient to show that $U \otimes *V$ is a fixed point of the monotone map $F : W \mapsto U \oplus (V \otimes W)$. Monotonicity of F follows since tensor and choice are monotonic operators, as demonstrated in Proposition 4.13. This fixed point property is formulated as $F(U \otimes *V) \leq U \otimes *V$, which is demonstrated as follows.

$$\begin{aligned} (U \otimes *V) \oplus U \oplus (V \otimes U \otimes *V) &\sim U \oplus \mathrm{I} \oplus V \otimes *V) && \text{by distributivity} \\ &\sim U \otimes *V && \text{by expansion} \end{aligned}$$

The remaining proof demonstrates that $U \otimes *V$ is not only a fixed point of F but also the least fixed point of F. This least fixed point property can by demonstrated by establishing that if $F(W) \leq W$ then $U \otimes *V \leq W$.

The proof works by first establishing two lemmas from the assumption $U \oplus (V \otimes W) \leq W$. The proof then proceeds by induction on the length of a derivation of a transition of $U \otimes *V$ to verify that $U \otimes *V \leq W$. Notice that this proof combines inductive and co-inductive reasoning.

For the first lemma, assume that $U \oplus (V \otimes W) \leq W$, and consider the transitions of $U \oplus (V \otimes W)$. There are two cases to consider. If $U \xrightarrow{C} P$ then, since $U \oplus (V \otimes W) \oplus W \sim W$, given the first transition below, there exists a Q such that the second transition below holds and $P \sim Q$.

$$\frac{U \xrightarrow{C} P}{U \oplus (V \otimes W) \xrightarrow{C} P} \qquad \text{yields} \qquad W \xrightarrow{C} Q$$

For the second lemma, suppose that $V \xrightarrow{C} P$ and $W \xrightarrow{D} Q$. Since $U \oplus (V \otimes W) \oplus W \sim W$, given the first transition below, there exists a R such that the second transition below holds and $R \sim P \parr Q$.

$$\dfrac{\dfrac{V \xrightarrow{C} P \quad W \xrightarrow{D} Q}{V \otimes W \xrightarrow{C \otimes D} P \parr Q}}{U \oplus (V \otimes W) \xrightarrow{C \otimes D} P \parr Q} \qquad \text{yields} \qquad W \xrightarrow{C \otimes D} R$$

Having established the above two lemmas, consider that transitions of $U \otimes *V$. Normalise the transitions of $*V$, as above, by transforming contraction and weakening on the left. Hence there are three cases to consider, as follows.

Suppose that $U \xrightarrow{C} P$ and $*V$ uses the weakening rule. By the first lemma, $U \xrightarrow{C} P$ yields $W \xrightarrow{C} Q$, such that $P \sim Q$. Hence the first transition below yields the second transition.

$$\dfrac{U \xrightarrow{C} P \quad *V \xrightarrow{\mathrm{I}} \bot}{U \xrightarrow{C \otimes \mathrm{I}} P \parr \bot} \qquad \text{yields} \qquad W \xrightarrow{C} Q$$

Now, suppose that $U \xrightarrow{C} P$, that $V \xrightarrow{D} Q$ and $*V$ uses only dereliction. By the first lemma, $U \xrightarrow{C} P$ yields $W \xrightarrow{C} R$, where $P \sim R$. So, by the second lemma, $V \xrightarrow{D} Q$ and $W \xrightarrow{C} R$ yield that $W \xrightarrow{C \otimes D} S$, where $S \sim Q \parr R \sim P \parr Q$. Hence the first transition below yields the second transition.

$$\dfrac{U \xrightarrow{C} P \quad \dfrac{V \xrightarrow{D} Q}{*V \xrightarrow{D} Q}}{U \otimes *V \xrightarrow{C \otimes D} P \parr Q} \qquad \text{yields} \qquad W \xrightarrow{C \otimes D} S$$

Consider the case of contraction. Assume that for some bounded length of derivation, if $U \otimes *V \xrightarrow{E} R$ then $W \xrightarrow{E} R'$ for some R' such that $R \sim R'$. Consider a one step longer derivation of a transition of $U \otimes *V$. This transition can be normalised to the form of the first transition below. Furthermore, due to expansion, $U \otimes *V \sim U \oplus (V \otimes *V \otimes U)$ holds; thus by bisimulation the second transition below also holds.

$$\dfrac{\dfrac{\dfrac{\dfrac{V \xrightarrow{D} Q}{*V \xrightarrow{D} Q} \quad *V \xrightarrow{E} R}{*V \otimes *V \xrightarrow{D \otimes E} Q \parr R}}{*V \xrightarrow{D \otimes E} Q \parr R} \quad U \xrightarrow{C} P}{U \otimes *V \xrightarrow{C \otimes D \otimes E} P \parr Q \parr R} \qquad \text{iff} \qquad \dfrac{\dfrac{V \xrightarrow{D} Q \quad \dfrac{*V \xrightarrow{E} R \quad U \xrightarrow{C} P}{*V \otimes U \xrightarrow{E \otimes C} R \parr P}}{V \otimes *V \otimes U \xrightarrow{C \otimes D \otimes E} P \parr Q \parr R}}{U \oplus (V \otimes *V \otimes U) \xrightarrow{C \otimes D \otimes E} P \parr Q \parr R}$$

Thus $*V \otimes U \xrightarrow{E \otimes C} R \parr P$ holds. By induction, $*V \otimes U \xrightarrow{E \otimes C} R \parr P$ yields that $W \xrightarrow{E \otimes C} R'$, where $R \parr P \sim R'$. So, by the second lemma, $V \xrightarrow{D} Q$ and $W \xrightarrow{E \otimes C} R'$ yield that $W \xrightarrow{C \otimes D \otimes E} S$, where $S \sim Q \parr R' \sim P \parr Q \parr R$.

All cases are covered; hence $*V \otimes U \leq W$. □

The following proposition demonstrates some commonly used equations that hold as a consequence of Proposition 4.16. Historically, Redko demonstrated that no finite collection of equations could axiomatise iteration [117]. Therefore adding any equations to this proposition can never produce a complete characterisation of iteration, hence cannot replace Proposition 4.16.

Corollary 4.17. *Immediate consequences of Proposition 4.16 include the following. Iteration is idempotent, converts additives to multiplicatives and can be denested, as follows.*

$$**U \sim *U \qquad *(U \oplus V) \sim *U \otimes *V \qquad *(*U \otimes V) \sim \mathrm{I} \oplus (V \otimes *(V \oplus U))$$

Proof. Consider the case of the conversion of a choice into a tensor. For one direction $*(U \oplus V) \leq *U \otimes *V$, first establish three inequations as follows.

The inequation $\mathrm{I} \leq *U \otimes *V$ holds due to the following.

$$\begin{array}{lll} \mathrm{I} \oplus (*U \otimes *V) & \sim \mathrm{I} \oplus *U \oplus (*U \otimes V \otimes *V) & \text{by expansion} \\ & \sim \mathrm{I} \oplus \mathrm{I} \oplus (U \otimes *U) \oplus (*U \otimes V \otimes *V) & \text{by expansion} \\ & \sim *U \otimes *V & \text{by idempotency and expansion} \end{array}$$

The inequation $U \otimes *U \otimes *V \leq *U \otimes *V$ holds due to the following.

$$\begin{array}{lll} (U \otimes *U \otimes *V) \oplus (*U \otimes *V) & \sim (U \otimes *U \otimes *V) \oplus *V \oplus (U \otimes *U \otimes *V) & \text{by expansion} \\ & \sim *V \oplus (U \otimes *U \otimes *V) & \text{by idempotency} \\ & \sim *U \otimes *V & \text{by expansion} \end{array}$$

A similar argument shows that $V \otimes *U \otimes *V \leq *U \otimes *V$.

Combing the above inequations establishes the following inequation.

$$\begin{array}{lll} \mathrm{I} \oplus (U \oplus V) \otimes *U \otimes *V & \sim \mathrm{I} \oplus U \otimes *U \otimes *V \oplus V*U*V & \text{by distributivity} \\ & \leq *U*V & \text{by idempotency} \end{array}$$

Hence, by the fixed point rule, $*(U \oplus V) \otimes \mathrm{I} \leq *U*V$, as required.

For the converse, first establish a lemma. Note that $*V \oplus (V \otimes *V) \leq *V$, by idempotency and expansion; therefore, by the fixed point rule, $*W*W \leq *W$. Therefore the following establishes the result.

$$\begin{array}{lll} *U*V & \leq *(U \oplus V) \otimes *(U \oplus V) & \text{by monotonicity} \\ & \leq *(U \oplus V) & \text{by the above lemma} \end{array}$$

Idempotency follows from $*U \otimes *U \leq *U$. By expansion and idempotency $\mathrm{I} \oplus (*U \otimes *U) \leq *U$, hence by the fixed point rule $\mathrm{I} \otimes **U \leq *U$. The converse holds since $U \leq *U$, by expansion and idempotency, hence by monotonicity $*U \leq **U$.

Denesting is proven by first establishing the intermediate result that $*(U \oplus V) \sim *V \otimes *(U \otimes *V)$ holds. This follows in one direction since.

$$\begin{array}{lll} *(U \oplus V) & \sim *(U \otimes \mathrm{I}) \otimes *V & \text{since choice converst to tensor} \\ & \leq *(U \otimes *V) \otimes *V & \text{by monotonicity} \end{array}$$

The converse direction holds by the following.

$$\begin{array}{lll} *V \oplus (U \otimes *V \otimes *(U \oplus V)) \oplus *(U \oplus V) & & \\ \quad \sim *V \oplus (U \otimes *U \otimes *(V \oplus V)) \oplus (*U \otimes *V) & & \text{since choice converts to tensor} \\ \quad \sim *V \otimes (\mathrm{I} \oplus (U \otimes *U) \oplus *U) & & \text{by idempotency and distributivity} \\ \quad \sim *V \otimes *U & & \text{by expansion} \\ \quad \sim *(V \oplus U) & & \text{since choice converts to tensor} \end{array}$$

Hence, by the least fixed point rule, the following inequation holds.

$$*V \otimes *(U \otimes *V) \leq *(V \oplus U)$$

Thus the lemma is established. The following reasoning therefore holds.

$$\begin{array}{lll} *(U \otimes *V) & \sim \mathrm{I} \oplus (U \otimes *V \otimes *(U \otimes *V)) & \text{by expansion} \\ & \sim \mathrm{I} \oplus (U \otimes *(U \oplus V)) & \text{by the lemma} \end{array}$$

Thus denesting is established. □

A classic consequence of the rules in Corollarly 4.17 is that queries without select can always be denested to a single iteration [84]. However, select breaks denesting since iteration and select do not commute. For instance the following query requires two iterations. The result is that for each of the first continuation triggered, zero or more instances of the second continuation are triggered. This query can be expressed using sub-queries in the current SPARQL Query working draft [66].

$$*\bigvee a.\bigvee n.\Big((|(a\ \mathit{name}\ n)|\ ;\ P) \otimes *\bigvee e.(|(a\ \mathit{email}\ e)|\ ;\ Q)\Big)$$

Iteration can be expressed as a colimit of repeated queries. This is a strictly more general property than Proposition 4.16 [82]. Since all constructs are colimits which distribute over tensor, the ideals generated by queries form a (commutative) quantale, as exploited by Montanari, Hoare and others [26, 71]. Quantales are related to spectral theory, which is related to information retrieval techniques used by search engines [77]. Commutative quantales also have an elegant representation theory in terms of locally compact Hausdorff spaces, via the Gelfand-Naimark representation [103]. Clarification of these connections is future work.

Proposition 4.18. *Iteration is a colimit of powers of queries. So, $U^n \otimes V \leq W$ for all n, if and only if $*U \otimes V \leq W$.*

Proof. Consider the base case. $U^0 = \mathrm{I}$ and $\mathrm{I} \oplus *U \sim \mathrm{I} \oplus \mathrm{I} \oplus U \otimes *U \sim *U$, by expansion and idempotency. Therefore $U^0 \leq *U$.

Assume that $U^n \leq *U$ and consider $U^{n+1} = U^n \otimes U$. By the following reasoning $U^{n+1} \leq *U$.

$$\begin{aligned}
(U^n \otimes U) \oplus *U &\sim (U^n \otimes U) \oplus \mathrm{I} \oplus (U \otimes *U) && \text{by expansion} \\
&\sim \mathrm{I} \oplus U \otimes (U^n \oplus *U) && \text{by distributivity} \\
&\sim \mathrm{I} \oplus (U \otimes *U) && \text{by induction} \\
&\sim *U && \text{by expansion}
\end{aligned}$$

So, by induction, $U^n \leq *U$ for all n. Hence if $V \otimes *U \leq W$ then $V \otimes U^n \leq W$ for all n, by monotonicity of tensor.

Conversely, assume that $V \otimes U^n \leq W$ for all n and consider the transitions of $V \otimes *U$. There are three cases, corresponding to the rules of iteration.

Suppose that only weakening is used to evaluate iteration. Hence the unit transition can be used as follows.

$$\frac{V \xrightarrow{C} P \quad *U \xrightarrow{\mathrm{I}} \bot}{V \otimes *U \xrightarrow{C \otimes \mathrm{I}} P \parr \bot} \qquad \text{yields} \qquad \frac{V \xrightarrow{C} P \quad U^0 \xrightarrow{\mathrm{I}} \bot}{V \otimes U^0 \xrightarrow{C \otimes \mathrm{I}} P \parr \bot}$$

Similarly, if only dereliction is used, then the same effect can be achieve using a single query as follows.

$$\frac{V \xrightarrow{C} P \quad \dfrac{U \xrightarrow{D} Q}{*U \xrightarrow{D} Q}}{V \otimes *U \xrightarrow{C \otimes D} P \parr Q} \qquad \text{yields} \qquad \frac{V \xrightarrow{C} P \quad U^1 \xrightarrow{D} Q}{V \otimes U^1 \xrightarrow{C \otimes D} P \parr Q}$$

For contraction, firstly assume that the following holds for $k = m$ and $k = n$, such that $Q \sim Q'$.

$$\frac{V \xrightarrow{C} P \quad *U \xrightarrow{D} Q}{V \otimes *U \xrightarrow{C \otimes D} P \parr Q} \qquad \text{yields} \qquad \frac{V \xrightarrow{C} P \quad U^k \xrightarrow{D} Q'}{V \otimes *U \xrightarrow{C \otimes D} P \parr Q'}$$

Also, by induction on n, $U^m \otimes U^n \sim U^{m+n}$. The base case follows from the unit of multiplication, since $U^m \otimes U^0 \sim U^m$. The induction step follows since $U^m \otimes U^{n+1} \sim U^{m+n} \otimes U$. Hence if $U^m \otimes U^n \xrightarrow{D \otimes E} Q' \parr R'$ such that $Q \sim Q'$ and $R \sim R'$, then by bisimulation $U^{m+n} \xrightarrow{D \otimes E} S$ such that $S \sim Q \parr R$. Hence the following holds.

$$\frac{V \xrightarrow{C} P \qquad \dfrac{\dfrac{*U \xrightarrow{D} Q \quad *U \xrightarrow{E} R}{*U \otimes *U \xrightarrow{D \otimes E} Q \parr R}}{*U \xrightarrow{D \otimes E} Q \parr R}}{V \otimes *U \xrightarrow{D \otimes E} P \parr (Q \parr R)} \qquad \text{yields} \qquad \frac{V \xrightarrow{C} P \quad U^{m+n} \xrightarrow{D \otimes E} S}{V \otimes *U \xrightarrow{D \otimes E} P \parr S}$$

Hence by induction on the derivation of $V \otimes *U \xrightarrow{C} P$, there is some n such that $V \otimes U^n \xrightarrow{C} Q$ and $P \sim Q$. Hence by the assumption $W \xrightarrow{C} R$ such that $R \sim Q \sim P$. Therefore $V \otimes *U \leq W$, as required. □

4.5.5 Embeddings of Boolean Algebras

Kozen demonstrates that Boolean algebras can be embedded in Kleene algebras [84]. The 'tests' of Kozen correspond to 'constraints' in SPARQL. Bisimulation verifies that the Boolean algebra of constraints embeds in the Kleene algebra is the same manner with similar consequences.

Proposition 4.19. *The Boolean algebra of constraints embeds in queries. Using standard classical implication, $\phi \Rightarrow \psi$ if and only if $\phi \leq \psi$. Or is choice, and is tensor, exists is select and an iterated constraint is always true.*

$$\phi \vee \psi \sim \phi \oplus \psi \qquad \phi \wedge \psi \sim \phi \otimes \psi \qquad \exists a.\phi \sim \bigvee a.\phi \qquad \mathrm{I} \sim *\phi$$

Proof. Consider the embedding of classical implication in queries. Assume that $\phi \Rightarrow \psi$, so if $\vDash \phi$ then $\vDash \psi$. Hence if $\phi \xrightarrow{\mathrm{I}} \bot$ then $\psi \xrightarrow{\mathrm{I}} \bot$. Hence the least equivalence relation $\sim_0$ such that if $\phi \Rightarrow \psi$ then $\phi \oplus \psi \sim_0 \phi$ is a bisimulation.

For the converse, assume that $\phi \oplus \psi \sim \psi$ and that $\vDash \phi$ holds. Hence $\phi \oplus \psi \xrightarrow{\mathrm{I}} \bot$, so there exists P such that $\psi \xrightarrow{\mathrm{I}} P$ and $P \sim \bot$. Thus $\vDash \psi$ must hold.

Consider the case of an interated constraint. An immediate consequence of the above is that $\phi \leq \mathrm{I}$, since I is the top element of the Boolean algebra, so $\mathrm{I} \oplus (\mathrm{I} \otimes \phi) \leq \mathrm{I}$. Hence, by the fixed point rule for iteration, $\mathrm{I} \otimes *\phi \leq \mathrm{I}$. For the converse consider the following due to expansion and idempotency, $\mathrm{I} \oplus *\phi \sim \mathrm{I} \oplus \mathrm{I} \oplus (\phi \otimes *\phi) \sim *\phi$. Hence $\mathrm{I} \leq *\phi$.

Consider the case of disjunction. Assume that $\vDash \phi \vee \psi$ holds, which follows only if $\vDash \phi$ holds or $\vDash \psi$ holds. Without loss of generality assume that $\vDash \phi$ holds. Hence the following proof trees are interchangeable.

$$\dfrac{\dfrac{\vDash \phi}{\phi \xrightarrow{\mathrm{I}} \bot}}{\phi \oplus \psi \xrightarrow{\mathrm{I}} \bot} \quad \text{iff} \quad \dfrac{\dfrac{\vDash \phi}{\vDash \phi \vee \psi}}{\phi \vee \psi \xrightarrow{\mathrm{I}} \bot}$$

Hence the least equivalence relation $\sim_0$ such that $\phi \vee \psi \sim_0 \phi \oplus \psi$ is a bisimulation.

Consider the case of conjunction. Assume that $\vDash \phi \vee \psi$ holds, which follows only if $\vDash \phi$ holds and $\vDash \psi$ holds. Hence the following proof trees are interchangeable.

$$\dfrac{\dfrac{\vDash \phi}{\phi \xrightarrow{\mathrm{I}} \bot} \quad \dfrac{\vDash \psi}{\psi \xrightarrow{\mathrm{I}} \bot}}{\phi \otimes \psi \xrightarrow{\mathrm{I}} \bot \parr \bot} \quad \text{iff} \quad \dfrac{\dfrac{\vDash \phi \quad \vDash \psi}{\vDash \phi \wedge \psi}}{\phi \wedge \psi \xrightarrow{\mathrm{I}} \bot}$$

Hence the least equivalence relation $\sim_0$ such that $\phi \wedge \psi \sim_0 \phi \otimes \psi$ and $\bot \parr \bot \sim_0 \bot$ is a bisimulation.

Consider the case of existential quantification of constraints. Assume that $\vDash \exists x.\phi$ holds. Thus there exists some v such that $\vDash \phi\{^v/_x\}$ holds, so the following are interchangeable.

$$\dfrac{\dfrac{\vDash \phi\{^v/_x\}}{\vDash \exists x.\phi}}{\exists x.\phi \xrightarrow{\mathrm{I}} \bot} \quad \text{iff} \quad \dfrac{\dfrac{\vDash \phi\{^v/_x\}}{\phi\{^v/_x\} \xrightarrow{\mathrm{I}} \bot}}{\bigvee x.\phi \xrightarrow{\mathrm{I}} \bot}$$

Hence the least equivalence relation $\sim_0$ such that $\bigvee x.\phi \sim \exists x.\phi$ is a bisimulation. □

As with classical implication, the preorder over triples can be embedded in the partial order over processes. However, since alias assumptions are only a preorder. If $C \sim D$ then it holds that $C \sqsubseteq D$ and $D \sqsubseteq C$, which is weaker than equality. Maintaining distinction of names is important for applications where β is not fixed over time.

Proposition 4.20. *$C \sqsubseteq D$ if and only if $C \leq D$.*

Proof. Assume that $C \sqsubseteq D$ and consider $C \oplus D$. Given that $C' \sqsubseteq C$, by transitivity, is holds that $C' \sqsubseteq D$. Hence the following implication holds.

$$\dfrac{C' \sqsubseteq C}{|C| \xrightarrow{C'} \bot} \quad \text{yields} \quad \dfrac{C' \sqsubseteq D}{|D| \xrightarrow{C'} \bot}$$

Hence $|C| \oplus |D| \sim |D|$ as required.

Conversely, assume that $C \oplus D \sim D$. Since, $C \xrightarrow{C} \bot$, by bisimulation, there exists P such that $D \xrightarrow{C} P$ such that $P \sim \bot$. This can only follow from $D \xrightarrow{C} \bot$ which follows from $C \sqsubseteq D$, as required. □

4.5.6 The algebra for continuations

The multiplicatives then, par and times and the units are related in the following manner. Combined with the previous rules the properties of then are established. The second rule shows that 'then' can be replaced by the unit delay (as in [5]).

Proposition 4.21. *An empty continuation can be removed, a continuation can be decomposed into the guard and a unit delayed process, and two continuations can be combined in a single par continuation, as follows.*

$$\mathrm{I} \,;\, \bot \sim \mathrm{I} \qquad U \otimes (\mathrm{I} \,;\, P) \sim U \,;\, P \qquad (U \,;\, P) \,;\, Q \sim U \,;\, (P \parr Q)$$

Proof. Consider the case of the empty continuation. The following proof trees are interchangeable.

$$\dfrac{\mathrm{I} \xrightarrow{\mathrm{I}} \bot}{\mathrm{I} \,;\, \bot \xrightarrow{\mathrm{I}} \bot \parr \bot} \quad \text{iff} \quad \mathrm{I} \xrightarrow{\mathrm{I}} \bot$$

Hence the least equivalence relation $\sim_0$ such that $\mathrm{I} \sim_0 \mathrm{I}\,;\bot$ and $\bot \sim_0 \bot$ is a bisimulation.

Consider the decomposition of continuations and assume that $U \xrightarrow{C} Q$. Hence the following proof trees are interchangeable.

$$\dfrac{U \xrightarrow{C} Q \qquad \dfrac{\mathrm{I} \xrightarrow{\mathrm{I}} \bot}{\mathrm{I}\,;P \xrightarrow{\mathrm{I}} \bot \parr P}}{U \otimes (\mathrm{I}\,;P) \xrightarrow{C\otimes \mathrm{I}} Q \parr (\bot \parr P)} \qquad \text{iff} \qquad \dfrac{U \xrightarrow{C} Q}{U\,;P \xrightarrow{C} Q \parr P}$$

Therefore the least equivalence relation $\sim_0$ such that $U\,;P \sim U \sim_0 U \otimes (\mathrm{I}\,;P)$ and $P \sim_0 \bot \parr P$ is a bisimulation.

Consider the relation between par and then and assume that $U \xrightarrow{C} R$. The following trees are interchangeable.

$$\dfrac{\dfrac{U \xrightarrow{C} R}{U\,;P \xrightarrow{C} R \parr P}}{(U\,;P)\,;Q \xrightarrow{C} (R \parr P) \parr Q} \qquad \text{iff} \qquad \dfrac{U \xrightarrow{C} R}{U\,;(P \parr Q) \xrightarrow{C} R \parr (P \parr Q)}$$

Let $\sim_0$ be the least congruence such that $(U\,;P)\,;Q$ and $U\,;(P \parr Q)$ and $(P \parr Q) \parr R \sim_0 P \parr (Q \parr R)$, and the relations associated associativity in Proposition 4.9 hold. Therefore, by the above, $\sim_0$ is a bisimulation. □

Some immediate consequences of the above proposition are the following. The first reveals a tight correspondence between tensor and par. The second indicates that names which are not bound in the continuation can be tightened to encompass the query only. The third shows that then is monotonic in its first argument.

Corollary 4.22. *Further properties of continuations.*

$$(\mathrm{I}\,;P) \otimes (\mathrm{I}\,;Q) \sim \mathrm{I}\,;(P \parr Q)$$

$$\bigvee a.U\,;P \sim \bigvee a.(U\,;P) \quad a \notin \mathrm{fn}\,(P) \qquad (U \oplus V)\,;P \sim (U\,;P) \oplus (V\,;P)$$

Proof. The first follows from decomposition and the relation between par and then, as follows.

$$(\mathrm{I}\,;P) \otimes (\mathrm{I}\,;Q) \sim (\mathrm{I}\,;P)\,;Q \sim \mathrm{I}\,;(P \parr Q)$$

The second follows from decomposition and distributivity of colimits over tensor, where $a \notin \mathrm{fn}\,(P)$, as follows.

$$\bigvee a.U\,;P \sim \bigvee a.U \otimes (\mathrm{I}\,;P) \sim \bigvee a.(U \otimes (\mathrm{I}\,;P)) \sim \bigvee a.(U\,;P)$$

The third follows from decomposition and distributivity of choice, as follows.

$$(U \oplus V)\,;P \sim (U \oplus V) \otimes (\mathrm{I}\,;P) \sim U \otimes (\mathrm{I}\,;P) \oplus (V \otimes (\mathrm{I}\,;P)) \sim (U\,;P) \oplus (V\,;P)$$

□

The algebraic properties demonstrate that the unit delay is the key feature to understanding the algebra of time. Future work would drop the operator 'then.'

4.5.7 Examples of optimisations

The algebra can be applied to optimise queries for distribution. In the example below the first query is rewritten as the tensor product of two queries.

$$*\bigvee a.((|(a\ knows\ b_2)|\ ;P) \oplus (|(a\ knows\ b_3)|\ ;Q)) \\ \sim *\bigvee a.(|(a\ knows\ b_2)|\ ;P) \otimes\ *\bigvee a.(|(a\ knows\ b_3)|\ ;Q)$$

The second query above is better for distribution. The tensor product allows two smaller queries to be immediately evaluated in parallel. The tighter scope of the select quantifiers reduces the branching when potential values to select are considered.

In the following example the scope of the select quantifier encompasses the whole query in its first form. In the second form the select quantifier is tightened to encompass only the relevant branch of the query.

$$\bigvee x.(|(a\ name\ x)| \otimes |(a\ member\ b)|\ ;P) \sim \bigvee x.(|(a\ name\ x)|\ ;P) \otimes |(a\ member\ b)|$$

The second form above is better for distribution, since it consists of the tensor product of two parts. One part is easy to answer and the other part contains everything bound by the select quantifier, including the continuation.

The distribution of queries across clusters of servers is a major problem for processing Linked Data [67].

4.6 Towards Full Completeness

The proofs in this chapter amount to a soundness result for the algebra established, with respect to bisimulation over queries and processes. To establish full completeness for the algebra it must be demonstrated that, if any two processes are bisimilar, then the bisimulation can be derived using only the algebraic properties presented. For instance, full completeness for the algebra of the finite π-calculus demonstrates that if two processes are bisimilar then the bisimilarity can be established using only the algebraic properties established by Milner, Parrow and Walker [99].

Insufficient algebraic properties have been established to demonstrate full completeness for the calculus introduced. The algebra is sufficiently complete to consider queries by themselves, but not queries embedded in arbitrary processes. Furthermore it is known, due to Redko [117], that

a finite axiomatisation of bisimulation is impossible for this calculus using equivalences only (in contrast to the finite π-calculus which uses only equalities). Thus any complete algebra for the calculus must make use of a partial order, as in Proposition 4.16 for instance. In this section, obstacles to establishing full completeness are highlighted. Thus future research challenges are identified.

4.6.1 Weak completeness results

A weak completeness result can be established. The weak completeness result is that any input labelled transition of the calculus can be expressed using the algebra. In a sense, this result demonstrates that the labelled transition can be disposed of and replaced by the algebra. A similar observation is made by Milner [98], where he reformulates the labelled transition system of the π-calculus using commitments, which are simulations. Milner's commitment relation commits a process to one of perhaps several execution paths, similarly to the commitment relation in Chapter 3.

A labelled transition represents a commitment to refine one query to another query. The refined query asks for a specific observation corresponding to the label, and then a specific continuation corresponding to the redux. From this perspective, there is no "arrow of time," as suggest by the conventional notation for a reduction, rather an "arrow of refinement." An arrow of refinement makes clear the relationship between the original process, the label and the redux.

Theorem 4.23. *If* $U \xrightarrow{E} P$ *then* $|E| \,;\, P \leq U$.

Proof. The proof proceeds by induction on the derivation of a labelled transition.

Consider the input axiom. Suppose that $|C| \xrightarrow{D} \bot$, where $D \subseteq C$. By Proposition 4.20 $|D| \leq |C|$ and by Proposition 4.21 the following holds: $|D| \sim |D| \otimes \mathrm{I} \sim |D| \otimes (\mathrm{I} \,;\, \bot) \sim |D| \,;\, \bot$. Hence $|D| \,;\, \bot \leq |C|$, as required.

Suppose that $\phi \xrightarrow{\mathrm{I}} \bot$ where $\vDash \phi$. Since $\vDash \phi$, by classical logic, $\vDash \mathrm{I} \to \phi$, so by Proposition 4.19, $\mathrm{I} \leq \phi$. Hence, by Proposition 4.21, $\mathrm{I} \,;\, \bot \leq \phi$ as required.

Consider the trigger guard rule. Consider $U \,;\, Q \xrightarrow{E} P \,;\, Q$, which follows from $U \xrightarrow{E} P$. By induction, it holds that $E \,;\, P \leq U$. Hence $(E \,;\, P) \,;\, Q \leq U \,;\, Q$, by monotonicity of the first argument of 'then'. Hence, by Proposition 4.21, $E \,;\, (P \parr Q) \leq U \,;\, Q$ as required.

Consider the tensor rule. Consider $U \otimes V \xrightarrow{E \otimes D} P \parr Q$ which follows from $U \xrightarrow{E} P$ and $V \xrightarrow{D} Q$. By induction, it holds that $E \,;\, P \leq U$ and $D \,;\, Q \leq V$. Hence, by monotonicity of tensor $(E \,;\, P) \otimes (D \,;\, Q) \leq U \otimes V$. Furthermore, by Proposition 4.21, the following holds.

$$\begin{aligned}(E \,;\, P) \otimes (D \,;\, Q) &\sim E \otimes (\mathrm{I} \,;\, P) \otimes D \otimes (\mathrm{I} \,;\, Q) \\ &\sim E \otimes D \otimes (\mathrm{I} \,;\, (P \parr Q)) \\ &\sim (E \otimes D) \,;\, (P \parr Q)\end{aligned}$$

Hence $(E \otimes D)\,;(P \parr Q) \leq U \otimes V$ as required.

Consider the choice rules. Without loss of generality consider the left branch, and suppose that $U \oplus V \xrightarrow{E} P$ follows from $U \xrightarrow{E} P$. By induction, it holds that $E\,;P \leq U$. Furthermore, by Proposition 4.12, it holds that $U \leq U \oplus V$. Hence, by transitivity, $E\,;P \leq U \oplus V$ as required.

Consider the select rules. Suppose that $\bigvee a.U \xrightarrow{E} P$ follows from $U\{^b/_a\} \xrightarrow{E} P$. By induction, it holds that $E\,;P \leq U\{^b/_a\}$. Furthermore, by Proposition 4.14, it holds that $U\{^b/_a\} \leq \bigvee a.U$. Hence, by transitivity, $E\,;P \leq \bigvee a.U$ as required.

Consider the weakening axiom. Suppose that $*U \xrightarrow{\mathrm{I}} \bot$. By Proposition 4.16, it holds that $\mathrm{I} \leq *U$. Hence, by Proposition 4.21, $\mathrm{I}\,;\bot \leq *U$ as required.

Consider the dereliction rule. Suppose that $*U \xrightarrow{E} P$ follows from $U \xrightarrow{E} P$. By induction, it holds that $E\,;P \leq U$. Furthermore, by Proposition 4.16, it holds that $U \leq *U$. Hence, by transitivity, it holds that $E\,;P \leq *U$ as required.

Consider the contraction rule. Suppose that $*U \xrightarrow{E} P$ follows from $*U \otimes *U \xrightarrow{E} P$. By induction, it holds that $E\,;P \leq *U \otimes *U$. Furthermore, by Proposition 4.16, $\mathrm{I} \leq *U$ so, by monotonicity of tensor, it holds that $U \otimes \mathrm{I} \leq *U \otimes *U$. Hence, by transitivity, it holds that $E\,;P \leq *U \otimes *U$ as required.

The result follows by induction. □

The converse of Theorem 4.23 states that if a query can be refined to an observation followed by a continuation using the algebra, then the corresponding labelled transition holds. The formulation, in Theorem 4.24, is weaker than the converse since the redux is considered up to bisimulation [122]. Thus the natural preorder defined by the algebra and the labelled transition system, quotiented by bisimulation have the same operational power. The proof is a trivial consequence of the definition of the natural partial order.

Theorem 4.24. *If* $|E|\,;P \leq U$ *then* $U \xrightarrow{E} P'$, *where* $P \sim P'$.

Proof. Clearly $|E|\,;P \xrightarrow{E} P$. Hence, assuming that $|E|\,;P \leq U$, it follows that $U \xrightarrow{E} P'$ where $P \sim P'$. □

This is a weak completeness result. A stronger completeness result would be best studied in a generalised version of the calculus with symmetric inputs and outputs. Such a calculus would be more general than required for the Linked Data application domain. Such a full completeness result would be of interest to the process calculus community, where few complete algebraic models of useful process calculi exist. In contrast, soundness is sufficient for the Linked Data community, so this short coming is not a major problem for this work.

4.6.2 Simulation as a coinductive refinement

Simulation is considered as a preorder over processes. Simulation is a relaxation of the definition of bisimulation, giving the greatest coinductive preorder rather than the greatest coinductive equivalence. Simulation is introduced to make up for inadequacies of the natural partial order over idempotent semirings. Simulation allows a number of further algebraic properties of processes to be established.

There is a major issue with the natural partial order provided by the idempotent semiring of queries. The natural partial order only applies to queries, so cannot be used to order arbitrary processes. More importantly, the natural partial order is not context closed everywhere. In particular, context closure fails for the right of the 'then' operator. This is due to the classic non-equality of process calculi exposed by bisimulation.

Proposition 4.25. *The natural partial order over processes $\leq$ is not monotone in all contexts.*

Proof. A simple counter example is sufficient. Clearly $|C| \leq |C| \oplus |D|$. Consider the context which guards that query with the unit transition. But $\mathrm{I} ; (|C| \oplus |D|)$ is not bisimilar to $(\mathrm{I} ; |C|) \oplus (\mathrm{I} ; |C| \oplus |D|)$. Thus $\mathrm{I} ; |C| \leq \mathrm{I} ; (|C| \oplus |D|)$ cannot be established. □

The notion of simulation is defined next. Simulation provides a coinductive definition which captures the notion that one process can do everything that another process can do. Therefore, the process on the left of the simulation relation is more deterministic than the process on the right.

Definition 4.26 (Simulation)**.** Simulation, written $\preceq$, is the greatest preorder such that the following holds, for any label l. If $P \preceq Q$ and $P \xrightarrow{l} P'$ then there exists some Q' such that $Q \xrightarrow{l} Q'$ and $P' \preceq Q'$.

Note that simulation only defines a preorder and not a partial order over queries quotiented by bisimulation. This is due to the classic property that mutual simulation does not yield bisimulation, where mutual simulation is a simulation relation in both directions. The classic example is that both of the following simulations hold; but the two processes are not bisimilar in general.

$$(\mathrm{I} ; (D \oplus E)) \oplus (\mathrm{I} ; D) \preceq (\mathrm{I} ; (D \oplus E)) \oplus (\mathrm{I} ; E) \qquad (\mathrm{I} ; (D \oplus E)) \oplus (\mathrm{I} ; E) \preceq (\mathrm{I} ; (D \oplus E)) \oplus (\mathrm{I} ; D)$$

It is trivial that bisimulation yields mutual simulation. So mutual simulation provides a coarser equivalence than bisimulation, i.e. fewer processes can be distinguished using mutual simulation. Mutual simulation is however finer than trace equivalence.

Fortunately, the natural partial order over queries is also sound with respect to the notion of simulation. Thus anything established using the natural preorder can also be established using simulation. Soundness is demonstrated as follows. The proof is trivial.

Lemma 4.27. *If $U \preceq V$, then $U \preceq V$.*

Proof. Assume that $U \oplus V \sim V$ and that $U \xrightarrow{E} P$. Hence by bisimulation $V \xrightarrow{E} Q$ such that $P \sim Q$. Since an equivalence relation is a preorder, $P \preceq Q$. Thus $U \preceq V$, by definition. □

Simulation is context closed, similarly to the context closure of bisimulation (Lemma 4.6). This is not surprising, since simulation is just a relaxation of the definition of bisimulation. Context closure for preorders is a monotonicity property. A monotone map preserves an ordering. Thus context closure is equivalent to requiring that all operators are monotone, as follows. The proof is easy.

Lemma 4.28 (Monotonicity)**.** *Simulation is monotone for all contexts. That is if $P \preceq Q$ then $CP \preceq CQ$, for all contexts for processes C.*

Proof. It is known that $\preceq$ is monotone for almost all contexts. Monotonicity of $\oplus$ and $\otimes$ follows by Corollary 4.13. Monotonicity of select quantifiers is established by Corollary 4.15. Iteration is monotone since it is the least fixed point of monotone operators, by Tarski's fixed point theorem. The monotonicity of parallel composition and blank node quantifiers follows by the same argument as in Lemma 4.6.

The natural preorder, $\preceq$ is a simulation, by Lemma 4.27. Hence $\preceq$ is monotone for all the above contexts. Since $U\,;P \sim U \otimes (\mathrm{I}\,;P)$, by Lemma 4.21, the only remaining case to consider is $\mathrm{I}\,;P$. Assume that $P \preceq Q$. There is only one transition of $\mathrm{I}\,;P$ to consider, and the following holds.

$$\mathrm{I}\,;P \xrightarrow{\mathrm{I}} P \qquad \text{yields} \qquad \mathrm{I}\,;Q \xrightarrow{\mathrm{I}} Q$$

Furthermore, $P \preceq Q$ by the assumption; hence $\mathrm{I}\,;P \preceq \mathrm{I}\,;Q$. Thus all contexts are monotone, as required. □

Note that the above proof demonstrates that simulation is just the coinductive extension of the natural partial order, to the context guarded by the unit transition. This is similar to simulation in modal logics where implication is extended coinductively to ensure that modalities are monotone [109]. For instance given a modal operator box, if $P \preceq Q$ then $\Box P \preceq \Box Q$, is established coinductively. Thus a strong correspondence between modalities and the unit guard is anticipated.

The above notion of simulation is defined over the labelled transition system. A natural notion of refinement can be defined over the reduction system as follows.

Definition 4.29 (Contextual refinement)**.** Contextual refinement is the greatest context closed, reduction closed preorder order $\preceq'$. A reduction closed preorder $\preceq_0$ is such that, if $P \preceq_0 P'$ and $P \rhd Q$, then there exists some Q' such that $P' \rhd Q'$ and $P' \preceq_0 Q'$.

This leads to the following result, which verifies that simulation is sound with respect to contextual refinement. The proof follows immediately from established results.

Theorem 4.30 (Simulation is a contextual refinement)**.** *If $P \preceq Q$ then $P \preceq' Q$.*

Proof. By Lemma 4.28, $\preceq$ is context closed. By Lemma 4.3, $\preceq$ is reduction closed. Hence, $\preceq$ is a contextual refinement. □

Thus simulation can be used as a proof technique to establish that one process is refined by another process. The above theorem verifies that simulation is correct, since it is sound with respect to contextual refinement. Contextual refinement is the natural preorder which refines behaviours and can be applied in any context. Thus contextual refinement is the correct notion to compare simulation to. This is analogous to the soundness of bisimulation with respect to contextual equivalence.

4.6.3 Some algebraic properties of simulation

Many simulation relations have already been established using the preorder over semirings. Some remaining preorders which are established using simulation directly include the following. These properties establish the possible ways in which processes compose in parallel can interact.

Proposition 4.31. *The following simulations hold. Firstly, a parallel composition can commit to a left merge. Secondly, two processes can commit to interact. Thirdly, the unit delay preserves blank nodes.*

$$U \,;\, P \preceq U \parr P \qquad U \,;\, C \preceq (U \otimes |C|) \parr C \qquad \mathrm{I} \,;\, \bigwedge a.P \preceq \bigwedge a.(\mathrm{I} \,;\, P)$$

Proof. Consider the simulation of 'then' (left merge) by a par operation. Suppose that $U \xrightarrow{E} P$ holds. Given that the first transition below holds the second transition holds.

$$\frac{U \xrightarrow{E} Q}{U \,;\, P \xrightarrow{E} Q \parr P} \quad \text{yields} \quad \frac{U \xrightarrow{E} Q}{U \parr P \xrightarrow{E} Q \parr P}$$

The continuations are equal so $U \,;\, P \preceq U \parr P$.

Consider the case of a left merge which simulates an interaction. Assume that $U \xrightarrow{E} P$ holds. Given that the first transition below holds the second transition holds, by selecting a particular pattern of interactions.

$$\frac{U \xrightarrow{E} P}{U \,;\, C \xrightarrow{E} P \parr C} \quad \text{yields} \quad \frac{\dfrac{\dfrac{C \sqsubseteq C}{U \xrightarrow{E} P \quad |C| \xrightarrow{C} \bot} \qquad \dfrac{C \sqsubseteq C}{C \xrightarrow{\overline{C}} C}}{U \otimes |C| \xrightarrow{E \otimes C} P \parr \bot}}{U \otimes |C| \parr C \xrightarrow{E} P \parr \bot \parr \bot \parr C}$$

Furthermore $P \parr C \sim P \parr \bot \parr \bot \parr C$, hence $U \,;\, C \preceq U \otimes |C| \parr C$ is a simulation.

Consider the interaction of the unit delay with a blank node quantifier. There is only one possible transition, as follows.

$$\mathrm{I} \,;\, \bigwedge\!\!\!\bigvee a.P \xrightarrow{\mathrm{I}} \bigwedge\!\!\!\bigvee a.P \qquad \text{yields} \qquad \frac{(\mathrm{I} \,;\, P) \xrightarrow{\mathrm{I}} P}{\bigwedge\!\!\!\bigvee a.(\mathrm{I} \,;\, P) \xrightarrow{\mathrm{I}} \bigwedge\!\!\!\bigvee a.P}$$

This is an instance of a more general result, which states that the unit delay preserves limits. □

Note that by extending the calculus with deletes the analogous property to the second property above would be $U \preceq (U \otimes C^{\bot}) \parr C$. Thus interactions in the calculus are characterised by simulations.

In a more general calculus these properties could be defined elegantly as a convolution [121]. The convolution which appears in parallel composition is recognised by Bergstra [20]. The left merge of Bergstra appears as the operation 'then' in this calculus, so then should be extended to all processes. The par operator should be decomposed into two operators $\parr$ and $\|$. The first is a commitment to an interaction the second is the convolution product. For true concurrency, two parallel processes may occur simultaneously using distinct resource, as enabled by the tensor product. The convolution product could decompose as follows.

$$P \,\|\, Q \sim (P \,;\, Q) \oplus (Q \,;\, P) \oplus (P \parr Q) \oplus (P \otimes Q)$$

This allows any process to be expressed as an update, which is used in a completeness proof [7]. Note that a traditional convolution in mathematics allows either one side or the other to act, as in the shuffle bialgebra [19], but does not account for interactions. An algebra which also includes the commitment to interact may be an extension of the concept of a bialgebra, such as a Hopf algebra, which extends bialgebras with a group-like inverse called the antipode [29, 30]. Existing work in this area is limited, so an exclusive study would be required to investigate this hypothesis.

Proposition 4.20 established that queries are covariant to the preorder over triples. In contrast, the following result demonstrates that stored triples are contravariant to the preorder over triples.

Proposition 4.32. *If $C \sqsubseteq D$ then $D \preceq C$.*

Consider bisimulation in the full calculus with deletes. Clearly deletes are covariant, by a similar argument to Proposition 4.20, which establishes queries are covariant. So queries and deletes are covariant; whereas stored triples are contravariant, by the above proposition.

It is conventional that positive formulae are covariant and linearly negated terms are contravariant. Negation should form a self-dual adjunction, which must be contravariant [15]. Unfortunately, in the syntax defined, stored triples are positive and deletes are negative. This suggests

that there is an oversight in the syntax of the calculus. It is natural to naïvely assume that deletes are negative and the thing deleted is positive, so it is easy to see why this oversight has been tolerated. The naïve syntax allows an intuitive reading of the calculus. A more precise syntax would use negation for stored triples and positive formulae for deletes.

Using the simulation preorder an new operator can be defined, which adds further clarity to the meaning of the weak completeness result 4.23. An adjoint operator to 'then', called left division $\diagdown$ can be defined as follows. This is possible since using simulation as the preorder over processes.

$$E \,;\, P \preceq U \qquad \text{iff} \qquad P \preceq E \diagdown U$$

This adjoint operator provides an alternative approach to labelled transitions, as in the non-commutative quantales of Abramsky and Vickers [6]. An update divided on the left by the label is the least upper bound of all its potential continuations. This operator can be extended to all updates in a more complete calculus, along the lines of Conway's treatment of input differentiation [41].

Note that the left division operator is different from the adjoint to tensor. The adjoint to tensor, called linear implication, would be defined as follows.

$$E \otimes V \preceq P \qquad \text{iff} \qquad V \preceq E \multimap P$$

The above operation $E \multimap P$ is the commitment of P to interact synchronously with the linear negation of E. The basic interactions of the calculus are examples of this operator. For instance, if $C^{\perp} \parr (C \otimes D)$ is interpreted as $C \multimap (C \otimes D)$ then the following reasoning holds.

$$C \otimes D \preceq C \otimes D \qquad \text{iff} \qquad D \preceq C \multimap (C \otimes D)$$

This is the form of the interaction rules discussed in this section. Hence the interactions in the calculus could be captured using an explicit adjoint to the spatial tensor.

A complete algebra for the calculus would involve developing the notions introduced in this section. The adjoint operators, left division and linear implication, should be explored. These adjoints characterise commitments to temporal and spatial actions, which are required for the fundamental elements of observation and interaction. Both of these adjoints can be formulated as quotients.

The other notion that should be developed is the algebra of convolutions. A convolution of two parallel process should transform the processes into a single non-deterministic process, which accounts for their possible interactions. This would allow the interleavings and communications of the processes to be characterised using algebra. Both of these investigations require the language of the calculus to be extended.

4.6.4 Weak cut elimination results

To be able to call the calculus a logic at least a cut elimination result must be established. The purpose of the discussion in this section is to emphasise that there is a clear strategy for obtaining a full cut elimination theorem for the calculus. Furthermore, some of the work in obtaining a cut elimination result has already been achieved.

Consider a candidate cut rule for the syndication calculus, suggested in Sec. 3.5.4 and repeated defined below. $P, Q, \ldots$ are arbitrary processes, while A is a formula in Multiplicative Linear Logic with triples as atoms.

$$\frac{P \parr A \rhd P' \quad Q \parr A^{\perp} \rhd Q'}{P \parr Q \rhd P' \parr Q'}$$

Now note that the following property holds in Linear Logic. The property below defines a self-dual adjunction, which algebraically characterises linear negation [134, 51, 15].

$$P \leq Q^{\perp} \parr R \qquad \text{iff} \qquad Q \otimes P \leq R$$

The above property of linear negation is enough to prove completeness of the labelled transition system with respect to the reduction system. The argument is as follows. Under the algebraic semantics it is possible to show the following correspondences.

$$\begin{array}{lll}
A^{\perp} \parr P \rhd Q & \text{iff} \quad \mathrm{I} \,;\, Q \leq A^{\perp} \parr P & \text{by the semantics of the reduction system} \\
& \text{iff} \quad A \otimes (\mathrm{I} \,;\, Q) \leq P & \text{by the above adjunction} \\
& \text{iff} \quad A \,;\, Q \leq P & \text{by algebraic properties of tensor and then} \\
& \text{iff} \quad P \xrightarrow{A} Q & \text{by the semantics of the labelled transition system}
\end{array}$$

Thus, under the translation described above, the formulae in the candidate cut rule translate as follows.

$$\begin{array}{rcl}
P \parr A \rhd P' & \text{iff} & P \xrightarrow{A^{\perp}} P' \\
P \parr A^{\perp} \rhd P' & \text{iff} & P \xrightarrow{A} P' \\
P \parr Q \parr \perp \rhd P' \parr Q' & \text{iff} & P \parr Q \xrightarrow{\mathrm{I}} P' \parr Q'
\end{array}$$

The result is that the following rule is exactly the cut rule in the reduction system proposed above.

$$\frac{P \xrightarrow{A} P' \quad Q \xrightarrow{A^{\perp}} Q'}{P \parr Q \xrightarrow{\mathrm{I}} P' \parr Q'}$$

The above rule will be familiar to readers who have used labelled transition systems. For instance it is used in the π-calculus to define the interaction of inputs and outputs [99]. Variations on this rule are used in the labelled transition systems in this work. The corresponding rule to the above rule in the labelled transition system of the syndication calculus is called the cut rule to emphasise the connections highlighted here.

Thus the argument that cut elimination is partially achieved in this work is as follows. The deductions in the labelled transition system use the cut rule. By the soundness of the labelled transition system with respect to the reduction system, Lemma 4.3, all unit labelled transitions also hold in the reduction system. Furthermore, the reduction system does not use any cut rule. Thus the cut rule can always be eliminated from a transition. Hence the proof that the labelled transition is sound with respect to the reduction system is the cut elimination result.

A restriction on this cut elimination result is that the cut formulae are restricted to the formulae on the labels. The formulae on the labels correspond to formulae in Multiplicative Linear Logic. Hence continuations are not considered in the cut elimination result highlighted here. Thus this weak cut elimination result is relevant to single step reductions only.

The big question is how cut elimination extends to continuations. What is $(\mathrm{I} \,;\, P)^{\perp}$ and how does it interact with $\mathrm{I} \,;\, P$ through cut? Understanding this question would help cut elimination to be extended from one step reductions, to arbitrary simulations. The goal would be a sound and complete logic for simulation in the syndication calculus, with a full cut elimination property. Such a logic is likely to require a modern proof calculus similar to the calculus of structures [61].

4.7 Conclusions on the Algebra

This section establishes an alternative semantics for the calculus introduced in the previous chapters. A labelled transition system is shown to be sound with respect to the reduction system. Furthermore, the notion of bisimulation in the labelled transition system is sound with respect to equivalence in the reduction system. Bisimulation is used to verify an algebra over queries, which extends existing notions of an algebra for SPARQL Query. An algebra of queries is useful when tackling problems associated with Linked Data, such as distributed query planning. Such problems are currently being pursued by the Linked Data community [132, 67].

The soundness result for the labelled transition system is Theorem 4.8. This follows from the fact that the labelled all unit transitions in the labelled transition system are commitments in the reduction system and that bisimulation holds in all contexts. Only a weak completeness result holds. The weak completeness result is that all reduction holds as unit transitions. To prove completeness, a suitable context for each label must be constructed. However, this is not possible without changing the calculus.

The soundness result for the algebra is established by the results in Section 4.5. Each of the algebraic properties introduced is proven to hold with respect to bisimulation. Thus applying the algebraic rules preserves bisimulation and therefore, by soundness of bisimulation, structural congruence. Ideally this should be a complete algebra, as is known to exist for the finite π-calculus. However, a complete algebra is trickier to obtain, but probably exists in a slightly larger calculus, as discussed in Section 4.6. The development of a complete algebra is the

deepest open question revealed by this work [83]. A calculus with a complete algebra would make a powerful case as a foundation for programming languages. This is objective has guided many design decisions throughout this work.

The queries form a commutative idempotent semiring, which provides a natural partial order over queries. This partial order is used to characterise choice, selects and iteration as colimits. Also, iteration is the least fixed point of a monotonic map over queries, hence queries form a Kleene algebra. A preorder over URIs allows small permissible mismatches between content and queries to be resolved, capturing key features of the RDFS standard. Also, a Boolean algebra of constraints is naturally embedded in queries, to provide further control. The algebra demonstrates several canonical algebras tightly integrated in one framework.

Chapter 5

Type Systems for Read–Write Linked Data

Type systems provide a light approach to verifying programs. A type, when assigned to a term in a language, indicates something about how that term can be used. Type systems are particularly useful when types guarantee static properties, such as ensuring that certain runtime errors do not occur in a program. As background material a simply typed λ-calculus is outlined. The simply typed λ-calculus is the inspiration for developing a simple type system for the syndication calculus.

A type system for the syndication calculus is introduced in this chapter. Firstly, the types for names are described. Names have no internal structure, so types are simply guidelines for where a name may appear in relation to other names in data. The lack of structure allows simple types to be assigned to names in several ways. Three different approaches to simple types are described which allow varying amounts of information to be lifted from the data to the type system. A subtype system is defined over the types to capture aspects of the RDF Schema (RDFS) standard, which improves interoperability between type systems.

The relationship between the simple types and the calculus are formally established. Typed versions of the syntax and operational semantics are defined. The type rules for the syntax formally define when a term is well typed. The type rules for the operational semantics deal with some dynamic type checks which cannot be guaranteed statically in general. The type system and an operational semantics are proven to be compatible, ensuring that the simple types preserve their intended properties.

5.1 Motivating Examples for the Type System

The calculus introduced in this work deals with different and potentially incompatible information. A type system can be used to ensure that incompatibilities do not arise. There are however

many possible perspectives on the type system.

A prudent separation of types has already made, between URIs and literals, in the rules of the select quantifier. The select quantifier rule has two versions: one selects URIs; the other selects literals. This is an improvement over the SPARQL recommendations which has one pool of variables for both URIs and literals [115]. The improvement avoids scenarios where, at runtime, literals end up where only URIs should appear and vice versa.

Separating identifiers for URIs and literals allows greater control over design decision for the type system. Two separate type systems can be defined: one for URIs; another for literals. The type systems for URIs and literals are very different, so require a distinct style of type system. Also the type system for either URIs or literals may vary depending on the application. Hence a separate type system leads to more modular definitions. Changes to the definition of the type system for URIs do not affect the definition of the type system for literals.

The separation of types for URIs and literals is an improvement over the RDFS specification [33]. The specification mixes up the two type systems. A major flaw of the RDFS specification is to insist that the top type of the type system for literals (*rdfs:Literal*) is a subclass of the top type for resources (*rdfs:Resource*). Such requirements are big mistakes which should be avoided in an implementation. Thus some liberty is taken in this section where W3C standards are interpreted.

This section returns to an example from the beginning of the previous two chapters. It shows three ways in which the W3C standards for XML Schema Datatypes and RDFS can be interpreted. Each of the three interpretations demonstrates how some information, which was handled as data and be treated in a type system. The approach with minimal information lifted to the type system is presented first; while an approach with a lot of information lifted to the type system is presented last. All approaches presented differ significantly from the approaches of Horrocks and Pan [108] and the approach of Pérez et al [127], which treat RDFS entirely at the level of data. This is the first treatment of types and classes for Linked Data using a genuine type system.

5.1.1 Basic XML Schema Data Types

The minimal type system that can and should be applied to processes is considered first. This type system treats all URIs as equal; thus it is sufficient to distinguish identifiers for URIs from identifiers for literals as achieved already. A basic conventional type system can be applied to literals, which avoids basic runtime errors. The type system avoids basic runtime errors at little cost to the user, with little controversy in terms of modelling. Thus, this type system represents the very lest type system which should be implemented for languages based on the calculus.

Consider the substantial example query from the beginning of the previous two chapters, reproduced below. The query involves select quantifiers which discover literals. The select quantifiers

bind variables which appear in constraints. The constraints involve operations which apply to strings of characters, including equality checks, string concatenation and regular expressions.

$$\bigvee a.\left(\begin{array}{l} \bigvee z.\left(\begin{array}{l}\left(\begin{array}{l}\bigvee x,y.\left(\begin{array}{l}|(a\ \textit{foaf:givenName}\ x)|\\ |(a\ \textit{foaf:familyName}\ y)|\\ (z = x + \text{`}\ \ \text{'} + y)\end{array}\right)\\ \oplus\\ |(a\ \textit{foaf:name}\ z)|\end{array}\right)\\ (z \in \texttt{`J.* Armstrong'})\end{array}\right)\\ \left(\begin{array}{l}|(a\ \textit{rdf:type dbp:Athlete})|\\ \oplus\\ |(a\ \textit{rdf:type dbp:Artist})|\end{array}\right);\\ P\end{array}\right)$$

In the example query, if the select quantifier tries to discover a number, then the constraint will not be satisfied. Hence the operational semantics determine that the query will not execute. The reason for the query not executing is not because the constraint evaluates to false, which would have been legitimate; but because the types were wrong so the constraint could not possibly be evaluated in the first place. Indeed the regular expression check may throw a runtime error.

Furthermore, a type error may occur in the continuation process. Suppose that the continuation process requires a string, but instead receives a natural number. Then without further dynamic type checks which ensure that only correct types are passed, the continuation process may throw a type error. Such errors should be picked up in advance using static typing, before the literal of the wrong type is passed to the continuation process.

To avoid these basic runtime errors, the query can be typed annotating the select quantifiers with datatypes. A typed version of the running example is presented below. Each of the select quantifiers for literals is annotated with a string datatype; while selected URIs are left without annotations.

$$\bigvee a.\left(\begin{array}{l} \bigvee z\colon \textsc{String}.\left(\begin{array}{l}\left(\begin{array}{l}\bigvee x\colon \textsc{String}, y\colon \textsc{String}.\left(\begin{array}{l}|(a\ \textit{foaf:givenName}\ x)|\\ |(a\ \textit{foaf:familyName}\ y)|\\ (z = x + \text{`}\ \ \text{'} + y)\end{array}\right)\\ \oplus\\ |(a\ \textit{foaf:name}\ z)|\end{array}\right)\\ (z \in \texttt{`J.* Armstrong'})\end{array}\right)\\ \left(\begin{array}{l}|(a\ \textit{rdf:type dbp:Athlete})|\\ \oplus\\ |(a\ \textit{rdf:type dbp:Artist})|\end{array}\right);\\ P\end{array}\right)$$

Clearly a type checker can be defined and developed for the above annotated process. The precise datatypes employed and the functions which apply to them is detailed in the XML Schema

Datatype standard [92]. For instance, the string data type above corresponds to the *xsd:string* type. There are design decisions to be made, such as choosing between a Hindley/Milner approach or a subtype based approach to data types [68, 101]. However, all such approaches are well understood for the basic datatypes involved; thus all design decisions are left to the W3C working group. It is less obvious what the type of URI. This is a new question, thus is the only question investigated further in this chapter.

5.1.2 RDFS top level classes as types

Consider an approach to typing URIs inspired by the RDFS standard. In the RDFS standard there are top level classes such as classes and predicates. A type system can be developed which lifts only these top level notions to the type system. Other classes in the RDFS standard are instead treated as data.

Consider the running example. Two URIs appear as the object of the *rdf:type* predicate, the URIs *dbp:Athlete* and *dbp:Artist* . These URIs are RDFS classes according to the RDFS specification. In this example these classes are treated like any other URIs which appear in data. They can be bound by select quantifiers, and appear in triples. Such URIs which represent RDFS classes are given the type CLASS. This correspondence between classes and types is indicated by the assignment of the form *dbp:Athlete* : CLASS in the type environment below.

Four predicates are used in the running example. Three of the predicates are assigned the same type in the type assumptions below. An assignment of the type $\mathsf{p}(\top, \textsc{String})$ to a URI, asserts that the URI is a predicate which can be used to relate any URI to a string literal object. This is consistent with the use of the predicates in the example. The type annotations for the predicates ensure that the variables in the object position are of type STRING.

The third predicate, *rdf:type* in the environment below, is a special predicate from the RDF vocabulary. The type assigned to *rdf:type* ensures that it is used to relate any URI to a URI which is of type class. In this way, the annotation of a resource with a class is treated like any other triple — at the level of data.

Select quantifiers which discover names are annotated with types. In the running example, the name a is assigned the top type $\top$. This top type places no restriction on the URI discovered. Thus any URI which matches the patterns described by the query will be sufficient. The annotated select quantifier, and the type assumptions on the left of the turnstile, allow the folowing

process to be typed.

$$\begin{array}{l} \textit{dbp:Athlete}\colon \textsc{class},\\ \textit{dbp:Artist}\colon \textsc{class},\\ \textit{foaf:givenName}\colon \mathsf{p}(\top, \textsc{String}),\\ \textit{foaf:familyName}\colon \mathsf{p}(\top, \textsc{String}),\\ \textit{foaf:name}\colon \mathsf{p}(\top, \textsc{String}),\\ \textit{rdf:type}\colon \mathsf{p}(\top, \textsc{class}), \end{array} \vdash \begin{array}{l} \bigvee a\colon \top.\\ \left(\begin{array}{l} \bigvee z\colon \textsc{String}.\\ \left(\begin{array}{l} \left(\begin{array}{l} \bigvee x\colon \textsc{String}, y\colon \textsc{String}.\\ \left(\begin{array}{l} |(a\ \textit{foaf:givenName}\ x)|\\ |(a\ \textit{foaf:familyName}\ y)|\\ (z = x + \text{`\ \ '} + y) \end{array}\right)\\ \oplus\\ |(a\ \textit{foaf:name}\ z)| \end{array}\right)\\ (z \in \texttt{`J.* Armstrong'}) \end{array}\right)\\ \left(\begin{array}{l} |(a\ \textit{rdf:type}\ \textit{dbp:Athlete})|\\ \oplus\\ |(a\ \textit{rdf:type}\ \textit{dbp:Artist})| \end{array}\right);\\ P \end{array}\right) \end{array}$$

By typing the above process, it is guaranteed that URIs with distinguished rôles are used consistently in their assigned rôle. The distinguished rôles correspond approximately to the top level types of RDFS and data type predicates of OWL. In the example above, the rôles used are class, predicate with string object, predicate with class object and arbitrary unrestricted resources.

Consider now aliases in the above example. The following aliases may be assumed, which indicate that the two classes in the query are subclasses of the class *foaf:Person*.

$$\textit{dbp:Athlete} \sqsubseteq \textit{foaf:Person}, \qquad \textit{dbp:Artist} \sqsubseteq \textit{foaf:Person}$$

The above aliases can be used when refining and evaluating the above process. For instance, it can be proven that the above query is a refinement of the more general query which refers only to the class *foaf:Person* instead of the disjunction of the two classes used above. The above alias assumptions define a preorder over URIs, which extends point-wise to triples. This emphasises that classes, other than top level classes, are treated as data.

This approach to types for URIs cleans up the RDFS specification. Two separate levels for RDFS classes are provided — the data level and the type level. This contrasts to the original RDFS specification which has an infinite nesting of classes, and is non-well founded. For instance the class *rdfs:Class* is instance of *rdfs:Class* .

The two level approach is also simpler than the approach of Horrocks and Pan, which introduces four levels, an instance layer, an ontology layer, a language layer and a meta-language layer [108]. Effectively, their instance and ontology layers are collapsed here into data; while, their language and meta-language layers are collapsed here into types. The extra layers appear unnecessary, since they are not supported by compelling case studies in applications.

5.1.3 RDF classes as types

A third option for typing processes can be achieved in the type system introduced in this chapter. In the previous approach, the top level classes are static type information; while other classes are dynamic data. Types are static since they never change; whereas data is dynamic since it can be changed by an update. The third approach motivated here allows further classes to be lifted to the level of types. Some classes may remain as data, so a compromise between static types and dynamic data can be found to suit a specific application.

Consider the URIs of type CLASS from the running example, i.e. *foaf:Person* , *dbp:Athlete* and *dbp:Artist* . Now instead for considering them as URIs of type CLASS, consider them as the three new types PERSON, ATHLETE and ARTIST. These new types are no longer of type CLASS, which is only used to type dynamic classes; hence a complex higher-order type system is avoided.

Now consider the alias assumption from the previous section. The types ATHLETE and PERSON are now treated as reserved types, rather than arbitrary URIs. Hence the alias assumption no longer apply, since they work at the level of data. Instead the following subtype assumptions are made.

$$\text{ATHLETE} \leq \text{PERSON}, \qquad \text{ARTIST} \leq \text{PERSON}$$

These subtype assumptions are used in a subtype system. The subtype system determines whether one type subsumes another type. The subtype system makes the type system more flexible.

The query in the running example can be modified to lift the classes in the data to the type system. The triples where the URIs for the classes appear in the object position of *rdf:type* are removed. The corresponding types are then appear in the type annotation for the URI in the subject position of the *rdf:type* predicate. This results in the following process, where the union type allows either type to be satisfied.

$$\begin{array}{l} \textit{foaf:givenName}\colon \mathsf{p}(\text{PERSON}, \text{STRING}) , \\ \textit{foaf:familyName}\colon \mathsf{p}(\text{PERSON}, \text{STRING}) , \\ \textit{foaf:name}\colon \mathsf{p}(\text{PERSON}, \text{STRING}) , \end{array} \vdash \left(\begin{array}{l} \bigvee a\colon \text{ATHLETE} \cup \text{ARTIST}. \\ \left(\begin{array}{l} \bigvee z\colon \text{STRING}. \\ \left(\begin{array}{l} \left(\begin{array}{l} \bigvee x\colon \text{STRING}, y\colon \text{STRING}. \\ \left(\begin{array}{l} |(a\ \textit{foaf:givenName}\ x)| \\ |(a\ \textit{foaf:familyName}\ y)| \\ (z = x + \text{`` '' } + y) \end{array} \right) \end{array} \right) \\ \oplus \\ |(a\ \textit{foaf:name}\ z)| \end{array} \right) \\ (z \in \texttt{`J.* Armstrong'}) \end{array} \right) \end{array} \right) ; P \right)$$

The above query is subtly different to the query in the previous sections. In the above process, the continuation process can guarantee that the URI received will always be annotated with the class ATHLETE or the class ARTIST. This is due to a type preservation property which is proven in this chapter. In contrast, in the previous sections the continuation process only knows that

the URI received was annotated with the class *dbp:Athlete* or *dbp:Artist* during the particular atomic transition in which the URI was discovered.

Using the subtype assumptions, the subtype system can check that ATHLETE ∪ ARTIST ≤ PERSON. Thus a query which uses the annotation PERSON in place of the annotation ATHLETE ∪ ARTIST is a more general query.

Note that the sub type system ensures that p(⊤, STRING) is stronger than the type p(PERSON, STRING). The subtype system ensures that the following subtype relation holds.

$$\mathsf{p}(\top, \text{PERSON}) \leq \mathsf{p}(\text{PERSON}, \text{STRING})$$

Thus assigning the type p(⊤, STRING) to the three FOAF name predicates is a sufficient type assertion to type the example above. Explicitly, the following type assumptions would be sufficient to type the above process.

$$\textit{foaf:givenName}: \mathsf{p}(\top, \text{STRING}), \textit{foaf:familyName}: \mathsf{p}(\top, \text{STRING}), \textit{foaf:name}: \mathsf{p}(\top, \text{STRING}),$$

Thus the top level approach, also discussed in Sec. 5.1.2, can accommodate the RDFS level approach, motivated in this section. This chapter proceeds to formalise a type system which accommodates both approaches to typing URIs which have now been motivated intuitively.

5.2 An Introduction to Type Systems

Before defining the type system for the syndication calculus an established simple type system is discussed. In a well typed programming language, the properties of a system are guaranteed by the type system. A typical type system can avoid basic programming errors by forbidding configurations which are meaningless and may lead to an error, such as using a number in place of a string in a regular expression.

Type checking is often performed by a Hindley/Milner style type system [68, 44]. A Hindley/Milner type system assigns a type to each syntactic structure. One type is assigned to natural numbers; while a disjoint type is assigned to strings of characters. This leads to a tractable syntax directed type system for conventional data, such as the literals in RDF. Such type systems are well understood and, for this work, assumed to be in the safe hands of the XML Schema Datatypes working group [25, 92].

However, URIs have no internal structure that can determine their type. Types for URIs are merely propositions which guide how the URI is used. Fortunately there exist well understood propositional type systems, based on intuitionistic propositional logics, briefly introduced in the next section. In this case the relationship between the types is tractable, as defined by a subtype system.

$$\begin{array}{rlr} \tau ::= & \textsc{a} & \text{atomic propositions} \\ \mid & \tau \Rightarrow \tau & \text{the arrow type} \end{array} \qquad \begin{array}{rlr} \Gamma ::= & a\colon \tau & \text{a type assertion} \\ \mid & \Gamma, \Gamma & \text{composition of environments} \\ \mid & \epsilon & \text{the empty environment} \end{array}$$

$$\begin{array}{rll} t ::= & a & \text{name} \\ \mid & t\ t & \text{application} \\ \mid & \lambda a\colon \tau.t & \text{abstraction} \end{array}$$

FIGURE 5.1: The syntax from simple types, type environments and typed λ terms.

The relationships between types and URIs must be explicitly specified. This requires a balance between static and dynamic typing, which is pursued in this chapter. Furthermore, care must be taken to ensure that a Hindley/Milner type system for data and propositional type system for URIs do not naïvely combine to form an intractable convoluted mess. This section exposes some issues with treating RDF types using conventional type theory.

5.2.1 An established type system

As background material, a simple but profound type system is defined. In the simply typed λ-calculus terms are assigned propositions as types. The properties ensured by types in the simply typed λ-calculus is that termination of a well typed program is guaranteed (it is strongly normalising); whereas in the untyped λ-calculus termination is undecidable. The syntax of the types and terms is presented in Fig. 5.1.

The syntax of types is built from atomic propositions, indicated by small capitals. Propositional types are built using the arrow type. The arrow type represents functions which perform a transformation from one propositional type type to another propositional type.

The syntax of the terms of the λ-calculus is built from names, which are identifiers with no internal structure. Functions are built using abstraction, which is indicated by the λ quantifier with a type annotation. The λ quantifier binds a name in a term. The bound name is a place holder for a term of the type indicated by the type annotation. Terms are composed using application. The term on the right is the term passed to the term on the left.

Associations between names and types are indicated by a type environment, which allows several type assertions to be expressed separated by commas. Type environments can only assign one type to an identifier. Type environments are used to form type judgements. A type judgement consists of a type environment separated by a turnstile from a term annotated with a type. A type judgement holds if it can be derived from the type axioms and rules in Fig. 5.2.

The axioms of the type system state that if an environment assumes that an identifier is of a particular type, then the identifier is indeed of that type. The type rule for abstraction internalises

$$\epsilon, \Gamma \equiv \Gamma \qquad (\Gamma_0, \Gamma_1), \Gamma_2 \equiv \Gamma_0, (\Gamma_1, \Gamma_2) \qquad \Gamma_0, \Gamma_1 \equiv \Gamma_1, \Gamma_0 \qquad \Gamma, \Gamma \equiv \Gamma$$

$$\frac{\Gamma \vdash t : \tau_0}{\Gamma, a : \tau_1 \vdash t : \tau_0} \qquad a : \tau \vdash a : \tau \qquad \frac{\Gamma, a : \tau_0 \vdash t : \tau_1}{\Gamma \vdash \lambda a : \tau_0 . t : \tau_0 \Rightarrow \tau_1} \qquad \frac{\Gamma \vdash t : \tau_0 \Rightarrow \tau_1 \quad \Gamma \vdash u : \tau_0}{\Gamma \vdash t\, u : \tau_1}$$

FIGURE 5.2: Type rules for the simply typed λ-calculus.

a type assumption as a λ expression. The type rule for application applies a term of a function type to a term of the correct input type, which results in a term of the output type.

The structural rules for the simply typed λ-calculus are weakening, exchange, and contraction. Weakening is captured by a rule. The structural rules for exchange and contraction are captured using a structural congruence which allows type environments to be reordered and multiple occurrences of a type assertions to be eliminated. The structural congruence can be applied at any point.

The type tree bellow is an example of a type judgement. The example uses a projection function, which takes two terms as arguments and returns only the first term. Two identifiers of suitable types are applied to the projection function.

$$\dfrac{\dfrac{\dfrac{\dfrac{c : \text{A}, d : \text{B}, a : \text{A}, b : \text{B} \vdash a : \text{A}}{c : \text{A}, d : \text{B}, a : \text{A} \vdash \lambda b : \text{B}.a : \text{B} \Rightarrow \text{A}}}{c : \text{A}, d : \text{B} \vdash \lambda a : \text{A}.\lambda b : \text{B}.a : \text{A} \Rightarrow (\text{B} \Rightarrow \text{A})} \quad c : \text{A}, d : \text{B} \vdash c : \text{A}}{c : \text{A}, d : \text{B} \vdash (\lambda a : \text{A}.\lambda b : \text{B}.a)\, c : \text{B} \Rightarrow \text{A}} \qquad c : \text{A}, d : \text{B} \vdash d : \text{B}}{c : \text{A}, d : \text{B} \vdash ((\lambda a : \text{A}.\lambda b : \text{B}.a)\, c)\, d : \text{A}}$$

The above type judgement can be normalised. The normalisation works by reducing abstractions applied to a terms. The type tree for the term replaces the bound name in the abstraction. So in the following example the name c replaces the bound name a. This reduces the tree to another well typed tree, shown below.

$$\dfrac{\dfrac{c : \text{A}, d : \text{B}, b : \text{B} \vdash c : \text{A}}{c : \text{A}, d : \text{B}, \vdash \lambda b : \text{B}.c : \text{B} \Rightarrow \text{A}} \quad c : \text{A}, d : \text{B} \vdash d : \text{B}}{c : \text{A}, d : \text{B} \vdash (\lambda b : \text{B}.c)\, d : \text{A}}$$

The normalisation process can be applied again resulting the following axiom. The normalisation process corresponds exactly to the execution of a program in the λ-calculus.

$$c : \text{A}, \vdash c : \text{A}$$

The correspondence between proof theory and typed programming languages can be extended to accommodate powerful features. However, Gallier warns that the approach is objective; it supports more subjective approaches to engineering of programming languages rather than replacing them: “Thus, although it is natural to view a program as a proof, the specification of

a program as a proposition proved by that proof, and the execution of a program as proof normalisation, it is abusive to claim that this is what programming is all about." [51] None the less, normalisation results are inspiration for many investigations into the foundations of diverse programming languages including those for concurrency [3].

5.2.2 Structural operational semantics

In a seminal note which has revolutionised approaches to specifying programming languages, Plotkin introduced a new perspective on operational semantics [112]. Programs are described in terms of transition systems; while the type system is defined using type environments. The transitions system and the type system are then demonstrated to be compatible. Thus, it can be proven that a well typed program, which satisfies the operational semantics will always reduce to a well typed program.

The structural operational semantics approach to type systems is more ad-hoc than normalisation in the λ-calculus. Therefore, the techniques may be applied to a wider range of programming languages, such as imperative languages. Since the calculi introduced in this work have an operational semantics a type system can be developed and verified according to the techniques of Plotkin. The main results of this chapter are produced in this manner.

5.3 Light Types for URIs and Literals

Many structural constraints, often expressed using an ontology [74], are not tackled in this work. The issue is that many invariants on structures which are imposed by an ontology require a global perspective on data.

Apparently simple invariants such as, "Resources with a nick name must also have a full name," are difficult to impose in an open environment. If delete removes one nickname, is there another nickname that maintains the invariant? This cannot be confirmed without knowing the extent of the entire store. Furthermore, a query which checks for a triple indicating a nickname might be unsuccessful. An unsuccessful query does not mean that the triple does not exist, only that the query could not be satisfied using the given computational resources.

A compromise between ontologies and pure data exists. A light type system which only deals with URIs and literals in triples, rather then structures across several triples, is proposed. Given one triple it is easier to tell whether the subject and object are of the correct type for a predicate. For instance, a type system may allow the assumption that a predicate *surname* relates a person to a string. Thus for any triple in which *surname* is observed as the predicate, the subject of the triple is a person and the object is a string. No knowledge of other triples is required.

It is still naive to assume that triples can be typed. Given a literal, say '3', it is reasonable to assume that '3' is an integer. However given a URI, say *eprint:15017*, what is the type of the

URI? Is it a person? Is it a predicate? The only sure answer is that it is a URI. Inside knowledge may say that the prefix of the URI indicates a paper, however in general such policies are not available. On the Web, few type assumptions can be made about URIs.

Intrinsic problems associated with semi-structured data on the Web are well known. In isolating the essential aspects of semi-structured data, Abiteboul highlights prevailing challenges [2]. Abiteboul argues for a light exchange model with an a-posteriori data guide; which addresses challenges including, eclectic types and a blurring of the distinction between schema and data. A light flexible type system for Linked Data addresses issues highlighted by Abiteboul.

For flexibility the type system works at three levels. Firstly, the XML Schema Datatype standard is reused to form a solid basic type system for RDF, where only literals are typed. Secondly, some types for URIs are moved from the data to the type system as propositional types. Thirdly, the standard inference system from the W3C standard RDFS is adapted to form a subtype system [33]. The subtype system offers flexibility and interoperability between the different strengths of typing. The three perspectives offer a compromise between static typing and dynamic data. In the presence of updates, static types are preserved while dynamic data changes.

5.3.1 A Standardised Type System for Literals

A type system for literals can detect basic programming errors in queries. Literals can appear in constraints in which only literals of a certain type make sense. For instance a regular expression only makes sense over a string of characters. A comparison between literals only makes sense if the two literals are of the same type. An inequality between literals only makes sense if the two literals are of the same type and there is a natural order over that type of data.

The SPARQL Query recommendation defines the operations which appear in filters [115]. The XML Schema Datatypes recommendation is reused to define the types for literals [25]. Literals are well understood, so it assumed that a type system for literals exists. From these standards a basic type system for updates can be defined. The basic type system annotates variables with data types as follows.

$$\bigvee x : \textsc{Date}. \left(\begin{array}{l} (\texttt{`01-01-1960'} \leq x) \\ (x < \texttt{`01-01-1970'}) \\ |(\mathit{document1\ created\ x})| \\ (\mathit{document1\ note\ candidate}) \end{array} \right)$$

The data types can be used in a type system to check that the constraints are correctly typed. In the example above, the constraints are inequalities between dates and a variable which is presumed to be a date. A type system accepts this update. If a constraint that checks the variable for a regular expression is also added then a type error is triggered.

Due to the data type standards, this level of typing can always be applied. Furthermore, a type inference algorithm allows the programmer to specify an update without types and still take

advantage of the type system. A suitable type system and inference algorithm for literals is assumed [35].

5.3.2 Light Propositional Types for RDF

By typing predicates, basic errors can be detected in the structure of triples. When the choice of verb does not match the choice of subject and object, no information needs to be known about the context of a sentence to reject a sentence. A simple sentence such as, "The mountain writes the fish," will always be nonsense.

Without type information it is less obvious that the above sentence is nonsense. Naming the mountain 'Ararat' and the fish 'Nemo' might give, "Ararat writes Nemo." By supposing that Ararat is a persons name and Nemo is a story, the nonsense appears to make sense. However, cultural experience suggests that Ararat refers to Mount Ararat, rather than some person. A mountain cannot be the subject of the verb to write, so the sentence remains nonsense.

The type of a URI is harder to establish than in natural language. What is the type of the URI *res:Mount_Ararat* ? According to DBpedia the type is another URI *dbp:Place* . The relationship between the URI and its type can be represented by the following triple.

res:Mount_Ararat rdf:type dbp:Place .

Note the namespace prefixes *dbp:* and *res:* are used by DBpedia for resources and terminology respectively [28]. The namespace prefix *rdf:* is used for standardised terminology for RDF [78]. The classification of the subject is indicated by the object of the predicate *rdf:type* in a triple, as above. The URIs used for classification are referred to as classes.

The type system can treat classes in two different ways. In the first approach, an RDFS class is modelled as a conventional type. An RDF class distinguished as an type is used to indicate that a URI it is an instance of the given type. Classes that are used as types are static properties of URIs which cannot be updated. In the second approach, almost all RDFS classes are treated just like any other URI. A class treated as a URI can be used in data, so can be linked to and updated as normal. Only universal top level types are used in the type system, as explained in Section 5.3.3.

The type system introduced next is designed so that both approaches to RDF classes may be used. This allows an interplay between applications which use classes as data and those which use classes as static types. This design decision indicates a blurring of the distinction between schema and data, highlighted by Abiteboul [2].

DATA	:=	STRING	string type	τ	::=	A	atomic proposition
	|	DATE	date type		|	$\mathsf{p}(\tau, \tau)$	predicate type
	|	...	etc.		|	$\mathsf{p}(\tau, \text{DATA})$	data predicate type
					|	$\top$	resource type
					|	$\tau \cup \tau$	union type
					|	$\#\tau$	container type

FIGURE 5.3: The syntax of types and type environments.

5.3.2.1 The syntax of propositional types.

The syntax of types which defines propositions which can be assigned to URIs, is presented in Fig. 5.3. The definition of types uses atomic propositions, which depend on the application.

Atomic propositional types. For a type system, a number of atomic propositional types are fixed. Atomic types are indicated by small capitals, such as ARTICLE, PERSON. Atomic types are application specific. They indicate static assumptions about a URI. For instance, an application which plots resources on a map deals with URIs typed by proposition PLACE. A different application which maintains a calendar of concerts may use the proposition EVENT.

Unlike datatypes, which restrict the structure of literals, atomic propositional types do not impose structure on a URI. Atomic propositional types are just syntax which guides how a URI is used.

Predicates between URIs. Atomic types can be used to construct predicate types. A predicate type indicates the type of a subject and object. For instance, the predicate *writes* may relate a person to a document. This is indicated by the predicate type $\mathsf{p}(\text{PERSON}, \text{DOCUMENT})$. The predicate *knows* may relate a person to a person, indicated by type $\mathsf{p}(\text{PERSON}, \text{PERSON})$.

Predicate types can be used to catch basic errors in triples, where the subject or the object does not match the expected type. For instance, a subject of type LOCATION and object of type ANIMAL are not valid for the predicate *writes*, under the assumptions above.

Datatype predicates. Predicates which allow literals as objects, are indicated using datatype predicates. For instance, the predicate *created* relates a document to a date literal, indicated by type $\mathsf{p}(\text{DOCUMENT}, \text{DATE})$. This allows both literals and variables of type DATE to be used as the object of *created*.

In the example in Sec. 5.3.1, the variable appears as the object of a triple with predicate *created*. The type assumption for *created* can be used to check that the type of the variable, matches how the variable is used in the triple. In the example, the variable also appears in a constraint. A

type error occurs if the type of the variable in the triple and the type of the same variable in the constraint do not match.

The resource type. A URIs can be assigned the resource type $\top$, which represents anything that can be assigned a URI. It corresponds to the class *rdfs:Resource* in RDFS [33]. The resource type can always be used to indicate URIs when no further static type information is known. This allows ome very general predicate types, by allowing any URI as the subject or object of a predicate. For instance, the atomic proposition DOCUMENT may be too stringent for the predicates *writes* and *created*. Instead, the types $\mathsf{p}(\textsc{person}, \top)$ and $\mathsf{p}(\top, \textsc{Date})$ can be used where any resource can be written and any resource can be created.

The resource type can only be applied to URIs. Literals have their own top type defined in the datatype standard [25]. By keeping these two top types distinct the type system for URIs and for literals do not interfere with each other.

The union type. The union type offers a compromise between atomic propositions and top. For instance, the predicate *writes* may apply to two atomic propositions ARTICLE and BOOK. The object of the predicate becomes the union type of the two atoms, as follows.

$$\mathsf{p}(\textsc{person}, \textsc{article} \cup \textsc{book})$$

Types for feeds and named graphs. To type named graphs and feeds the container type is introduced. Typically, the subject of a triple is what is described by the triple. By analogy, the subject of a simple sentence is what is described by the sentence. A feed can therefore indicate what type of resources its triples describe. For instance, a feed of articles contains triples with subjects of type article. A feed of articles is indicated by the type $\#\textsc{article}$.

Container types are well suited to feeds. For instance, BBC News delivers feeds of articles, Flickr delivers feeds of photos and Google Calendar delivers feeds of events. However, named graphs are intended to contain diverse triples. The most general container type $\#\top$ indicates a named graph with no restrictions on content.

Container types allow novel features. For instance, the *seeAlso* predicate indicates where to find more information about a resource. A suitable type would be $\mathsf{p}(\top, \#\top)$, which suggests that more information about the subject can be found in a named graph indicated by the object.

Note that types for atomic propositions, predicates and union types are implicit in the RDFS standard. However, there is no standard type for named graphs, since named graphs and RDFS were introduced independently [33, 38]. The named graph type suggests one light approach to typing named graphs in the same spirit a predicate types.

$$\frac{\tau_0 \leq \tau_1}{\tau_0 \leq \tau_1 \cup \tau_2} \qquad \frac{\tau_0 \leq \tau_2}{\tau_0 \leq \tau_1 \cup \tau_2} \qquad \frac{\tau_0 \leq \tau_2 \quad \tau_1 \leq \tau_2}{\tau_0 \cup \tau_1 \leq \tau_2} \qquad \tau \leq \top \qquad \frac{\tau_0' \leq \tau_0 \quad \tau_1' \leq \tau_1}{\mathrm{p}(\tau_0, \tau_1) \leq \mathrm{p}\left(\tau_0', \tau_1'\right)}$$

$$\frac{\tau_0 \leq \tau_1 \quad \mathrm{D}_0 \leq \mathrm{D}_1}{\mathrm{p}(\tau_1, \mathrm{D}_1) \leq \mathrm{p}(\tau_0, \mathrm{D}_0)} \qquad \frac{\tau_0 \leq \tau_1}{\#\tau_1 \leq \#\tau_0} \qquad \frac{\tau_0 \leq \tau_1 \quad \tau_1 \leq \tau_2}{\tau_0 \leq \tau_2} \qquad \tau \leq \tau$$

FIGURE 5.4: Axioms and rules of the subtype system: left injection, right injection, least upper bound, top, predicate, data predicate, feed, transitivity and reflexivity.

5.3.2.2 A subtype system based on RDFS.

Subtypes are essential for enabling some basic scenarios. The subtype system, presented in Fig. 5.4, defines a preorder over types. The subtype system enables interoperability by enabling different strengths of type system to coexist. For instance, data which is heavily typed can still be used if very little type information is required. This light approach to interoperability avoids typical data integration problems, such as the integration of schema with conflicting constraints [46].

The subtype axioms. The subtype relation $\tau_0 \leq \tau_1$ can be read τ_0 is stronger than or equivalent to τ_1. So, if something is of type τ_0, then it is also of type τ_1. The basic axiom of the subtype system, reflexivity, states that every type is at least as strong as itself.

Interoperability of systems can be further enabled by application specific axioms. For instance an application may define types IMAGE and MEDIA. To indicate that an image can also be treated as media, the axiom IMAGE $\leq$ MEDIA can be included. Application specific subtype axioms always relate an atomic proposition to another atomic proposition. They correspond to the subClass predicate in RDFS, lifted to the type system [33].

Subtypes for union types. A union type indicates that a term is of one of two types. For instance, the type ARTICLE is a subtype of ARTICLE $\cup$ IMAGE. This is the left injection of a type into a union type. Similarly, the type IMAGE is a subtype of ARTICLE $\cup$ IMAGE by the symmetric right injection rule.

A union type is the least upper bound of two types. If ARTICLE $\leq$ MEDIA and IMAGE $\leq$ MEDIA are subtype axioms, then ARTICLE$\cup$IMAGE is also bound above by MEDIA. A URI of type ARTICLE$\cup$IMAGE means that the URI may vary between identifying either an article or an image.

Subtypes for resource type. Every type is bound above by the resource type. Thus both PERSON and p(PERSON, PERSON) are bound above by $\top$. If a URI serves the role of a person or a

predicate, then the same URI can be used where no restrictions on the resource apply. Allowing predicates to be resources is a lighter approach than imposing the constraint the URIs for resources and predicates are disjoint, as in OWL [74].

Subtypes for predicates. Predicate types are contravariant in both the subject and object. Contravariance switches the direction of the subtype relation. For instance, a predicate of type $\mathsf{p}(\textsc{person}, \top)$ is also of type $\mathsf{p}(\textsc{person}, \textsc{article})$. Contravariance allows the resource type to be strengthened to the article type. So a predicate which allows anything as the object can certainly have an article as the object.

Data predicates are also contravariant in both arguments. For instance, a predicate of type $\mathsf{p}(\top, \textsc{String})$ can be used as a predicate of type $\mathsf{p}(\textsc{person}, \textsc{String})$. The subtype relation for datatypes is defined in the XML Datatypes standard [25]. For instance, a 'normalised string' from the standard can be used in place of a string. Because datatypes and types for URIs are separate the subtype systems do not interfere.

Subtypes for named graphs and feeds. With subtypes, the type of feeds containing more than one type of resource can be expressed. For instance, $\#(\textsc{article} \cup \textsc{photo})$ is a feed of articles and photos collectively have their union type. The feed constructor is contravariant meaning that the type of the content of the feed may be strengthened. For instance, a feed containing resources which are either articles or photos can be treated as a feed containing only articles by ignoring the photos. This is captured by the subtype relation $\#(\textsc{article} \cup \textsc{photo}) \leq \#\textsc{article}$.

5.3.2.3 Cut Elimination for the Subtype System.

Cut elimination allows the transitivity rule to be eliminated from subtype proofs. Cut elimination is a fundamental result of proof theory, which provides further justification for the rules of the subtype system.

The proof relies on taking the reflexive transitive closure the subtype assumptions as axioms. For instance, consider the atomic types, $\textsc{reporter}$, $\textsc{journalist}$, $\textsc{person}$, and the subtype axioms $\textsc{reporter} \leq \textsc{journalist}$ and $\textsc{journalist} \leq \textsc{person}$. Transitivity must be used to determine the subtype relation $\textsc{reporter} \leq \textsc{person}$, so this subtype assumption is included as a subtype axiom.

Theorem 5.1 (Cut elimination)**.** *Given a proof of a subtype relation $\tau_0 \leq \tau_1$, there exists a normalised proof with the same conclusion which does not use cut (i.e., the transitivity rule in Fig. 5.4).*

A major benefit of cut-elimination is that sub-typing is syntax directed. A sub-type proof can be found by applying the rules which correspond to the type constructors. Transitivity is not syntax directed as it can be applied at any point, so its elimination demonstrates that sub-tying is algorithmic.

Proof. The proof works by transforming a proof of a subtype assertion. The transformation is indicated by $[\![\cdot]\!]$. If the last rule is not a cut rule the rule is simply applied to the premise. If the rule is a cut then one of the following cases apply. The result then follows by structural induction on the depth of the proof tree.

Consider the case where the left branch of a cut is another cut rule. The nested cut rule can be normalised first, as demonstrated by the transformation bellow.

$$\left[\!\!\left[\dfrac{\dfrac{\begin{array}{cc}\pi_0 & \pi_1\\ \tau_0 \le \tau_1 & \tau_1 \le \tau_2\end{array}}{\tau_0 \le \tau_2} \quad \begin{array}{c}\pi_2\\ \tau_2 \le \tau_3\end{array}}{\tau_0 \le \tau_3} \right]\!\!\right] \longrightarrow \left[\!\!\left[\dfrac{\dfrac{\left[\!\!\left[\begin{array}{cc}\pi_0 & \pi_1\\ \tau_0 \le \tau_1 & \tau_1 \le \tau_2\end{array}\right]\!\!\right]}{\tau_0 \le \tau_2} \quad \begin{array}{c}\pi_2\\ \tau_2 \le \tau_3\end{array}}{\tau_0 \le \tau_3} \right]\!\!\right]$$

By induction, the resulting nested proof tree will be cut free, so a different case applies. The same technique can be used when a nested cut appears on the right branch of a cut.

Consider the case of the reflexivity axiom. If the reflexivity axiom appears on either the left or right branch of a cut can be eliminated. The case of the elimination of reflexivity on the left is demonstrated below.

$$\left[\!\!\left[\dfrac{\begin{array}{cc} & \pi\\ \tau_0 \le \tau_0 & \tau_0 \le \tau_1\end{array}}{\tau_0 \le \tau_1} \right]\!\!\right] \longrightarrow \left[\!\!\left[\begin{array}{c}\pi\\ \tau_0 \le \tau_1\end{array} \right]\!\!\right]$$

By induction, the result of the transformation is a cut free proof.

Consider the case of the top type on the right of the cut. The cut can be absorbed by the top type axiom, as follows.

$$\left[\!\!\left[\begin{array}{c}\pi\\ \tau_0 \le \top\end{array} \right]\!\!\right] \longrightarrow \tau_0 \le \top$$

This is trivially a cut free proof.

Consider the case of union introduction rule on the left of a cut. Cut elimination can be applied separately to each of the premises of the union introduction rule, as demonstrated below.

$$\left[\!\!\left[\dfrac{\dfrac{\begin{array}{cc}\pi_0 & \pi_1\\ \tau_0 \le \tau_1 & \tau_1 \le \tau_2\end{array}}{\tau_0 \cup \tau_2 \le \tau_2} \quad \begin{array}{c}\pi_2\\ \tau_2 \le \tau_3\end{array}}{\tau_0 \cup \tau_1 \le \tau_3} \right]\!\!\right] \longrightarrow \left[\!\!\left[\dfrac{\begin{array}{cc}\pi_0 & \pi_2\\ \tau_0 \le \tau_2 & \tau_2 \le \tau_3\end{array}}{\tau_0 \le \tau_3} \right]\!\!\right] \left[\!\!\left[\dfrac{\begin{array}{cc}\pi_1 & \pi_2\\ \tau_1 \le \tau_2 & \tau_2 \le \tau_3\end{array}}{\tau_1 \le \tau_3} \right]\!\!\right]$$

By induction, the result of the transformation is a cut free proof.

Consider the case of the union projection rules. Without loss of generality, consider the left projection. The cut is pushed up the proof tree, as demonstrated below.

$$\left[\!\!\left[\dfrac{\pi_0 \quad \dfrac{\pi_1}{\tau_1 \le \tau_2}}{\dfrac{\tau_0 \le \tau_1}{\tau_0 \le \tau_2 \cup \tau_3}}\right]\!\!\right] \longrightarrow \left[\!\!\left[\dfrac{\dfrac{\pi_0 \quad \pi_1}{\tau_0 \le \tau_1 \quad \tau_1 \le \tau_2}}{\dfrac{\tau_0 \le \tau_2}{\tau_0 \le \tau_2 \cup \tau_3}}\right]\!\!\right]$$

By induction, the result is a cut free proof.

Consider the case of a union projection applied to a union introduction. Without loss of generality consider the left projection. The result is only the left premises of the union introduction rule is required; the irrelevant branch is removed by the elimination step, as demonstrated below.

$$\left[\!\!\left[\dfrac{\dfrac{\pi_0 \quad \pi_1 \quad \pi_2}{\tau_0 \le \tau_1 \quad \tau_1 \le \tau_3 \quad \tau_2 \le \tau_3}}{\dfrac{\tau_0 \le \tau_1 \cup \tau_2 \quad \tau_1 \cup \tau_2 \le \tau_3}{\tau_0 \le \tau_3}}\right]\!\!\right] \longrightarrow \left[\!\!\left[\dfrac{\dfrac{\pi_0 \quad \pi_1}{\tau_0 \le \tau_1 \quad \tau_1 \le \tau_3}}{\tau_0 \le \tau_3}\right]\!\!\right]$$

By induction, the result of the transformation is a cut free proof.

Consider the case of cut applied to two predicate subtype rules. In this case the contravariant premises of each subtype rule are combined, as follows.

$$\left[\!\!\left[\dfrac{\dfrac{\pi_0 \quad \pi'_0 \quad \pi_1 \quad \pi'_1}{\tau_1 \le \tau_0 \quad \tau'_1 \le \tau'_0 \quad \tau_2 \le \tau_1 \quad \tau'_2 \le \tau'_1}}{\dfrac{\mathrm{p}\left(\tau_0, \tau'_0\right) \le \mathrm{p}\left(\tau_1, \tau'_1\right) \quad \mathrm{p}\left(\tau_1, \tau'_1\right) \le \mathrm{p}\left(\tau_2, \tau'_2\right)}{\mathrm{p}\left(\tau_0, \tau'_0\right) \le \mathrm{p}\left(\tau_2, \tau'_2\right)}}\right]\!\!\right]$$

$$\longrightarrow \dfrac{\left[\!\!\left[\dfrac{\pi_1 \quad \pi_0}{\tau_2 \le \tau_1 \quad \tau_1 \le \tau_0}\right]\!\!\right] \quad \left[\!\!\left[\dfrac{\pi'_0 \quad \pi'_1}{\tau'_2 \le \tau'_1 \quad \tau'_2 \le \tau'_0}\right]\!\!\right]}{\dfrac{\tau_2 \le \tau_0}{\mathrm{p}\left(\tau_0, \tau'_0\right) \le \mathrm{p}\left(\tau_2, \tau'_2\right)}}$$

By induction, each of the new cuts have a cut free proof, so the result of the transformation has a cut free proof.

Consider the case of cut applied to two container types. As with predicate type, cut can be applied to the contravariant premises of the container type rule, a follows.

$$\left[\!\!\left[\dfrac{\pi_1 \quad \dfrac{\pi_3}{\tau_1 \le \tau_0 \quad \tau_2 \le \tau_1}}{\dfrac{\#\tau_0 \le \#\tau_1 \quad \#\tau_1 \le \#\tau_2}{\#\tau_0 \le \#\tau_2}}\right]\!\!\right] \longrightarrow \left[\!\!\left[\dfrac{\dfrac{\pi_3 \quad \pi_1}{\tau_2 \le \tau_1 \quad \tau_1 \le \tau_0}}{\dfrac{\tau_2 \le \tau_0}{\#\tau_0 \le \#\tau_2}}\right]\!\!\right]$$

By induction, the new premise has a cut free proof, so the result of the transformation has a cut free proof.

This covers every case. Hence by structural induction on the proof of a subtype derivation, a cut free proof with the same conclusion exists. □

An immediate consequence of the above proof is that the subtype relation forms a category. The category is such that types are objects, subtype proofs are arrows, reflexivity are the identity arrows and cut is composition. In this category union types are co-products, the resource type is the final object, the container type is a contravariant functor and the predicate type is a contravariant bi-functor. This corollary is elementary category theory, which can be read directly from the cut elimination proof. It demonstrates that the constructs of the type system are common features of type systems. This is left as a side note, to avoid introducing categories in this work.

5.3.2.4 Interoperability of Subtype Systems.

A store may include Linked Data from more than one source with static type information. The subtype system enables interoperability between different subtype type systems. For instance, suppose there exist three stores, for distinct applications. Suppose that one store uses atomic types MUSICIAN and VENUE, while another store uses atomic types PERSON and LOCATION. A third store uses content from both servers, so must handle all four atomic types. Furthermore, the third server is given the subtype assumptions MUSICIAN $\leq$ PERSON and VENUE $\leq$ LOCATION, which improves interoperability of content from both servers.

In the example involving three stores, the subtype systems of the first two stores can be extended to the subtype system of the third store. A subtype system δ_0 is defined to extend to a subtype system δ_1 if and only if the completion of subtype axioms in δ_0 is contained in the completion of subtype axioms in δ_1. Thus if δ_0 extends to δ_1, then all subtype assumptions with respect to δ_0 are subtype assumptions with respect to δ_1. Valid extensions can be checked efficiently using the Dedekind-MacNeille completion [89].

Subtypes ease restrictions imposed by types. Linked Data can involve data from stores with different subtype systems. By extending the subtype systems lightweight interoperability across diverse Linked Data systems can be achieved.

5.3.3 A compromise between light typing and no typing

Another approach to modelling RDFS using a type system is highlighted in this section. The alternative type system is just a restriction of the type system already introduced. Instead of lifting arbitrary classes to the type system, only the top level classes of RDFS are used. This restricted approach to types offers a compromise between using no typing for URIs and using

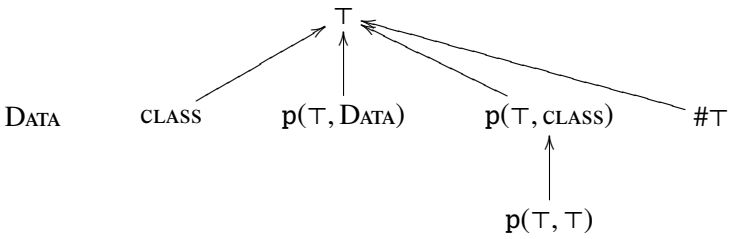

FIGURE 5.5: The subtype relationships between top level types.

arbitrary classes in the type system. By using only top level classes as types the type system become much simpler and cleaner. All other features of RDFS are treated at the level of the terms of the language, rather than the type system. Some features of RDFS are constrained, but at little cost to the applications modelled.

The issue addressed is that the official specification for RDFS uses classes in a convoluted manner. This is acknowledged in related work which constructs a Tarski-style model theory for RDFS [108]. Two levels of classes are identified by Horrocks and Pan. The first level of classes consists of atomic classes such as PERSON or ARTICLE, which are application specific. The second level of classes consists of very general top level classes, which, unlike first level classes, are not application specific. By making a clear distinction between first and second level classes non-well-founded models are avoided. Non-well-founded models relax the axiom of foundation, which in set theory prevents a set from being contained in itself [8]. However, non-well-founded sets are only required for operational behaviour, thus would be excessive if employed to model RDFS.

The top level types are the resource type ($\top$), the class type (CLASS) and several predicate types ($\mathsf{p}(\top, \top)$, $\mathsf{p}(\top, \text{CLASS})$). The datatype predicates, with any subject are included, since the types of literals do not vary between applications. All types that appear in the restricted type system have already been introduced, except the class type.

The class type — CLASS — is just an atomic type in the restricted type system. No further atomic types are used, thus all other RDF classes are treated as URIs which appear in terms. Just as the atomic propositional type PERSON is used to type URIs of type person in the light type system; the propositional type CLASS is used to type URIs which are used as classes in this restricted type system. Thus the predicate *rdf:type* is assigned the type $\mathsf{p}(\top, \text{CLASS})$ and used in terms.

The relationships between the distinguished top level classes are presented in Fig. 5.5. The arrows indicate subtype relationships. The resource type $\top$ is the greatest type, thus dominates the class type. By the contravariance of predicates, a predicate which can relate any two resources can relate any resource to a class. However, most other types are separate. The class type CLASS is not related to any property type. The type system for literals is kept separate from the type system for URIs, so a top level class and a data type cannot be compared using the subtype system. Thus datatype predicates and other predicates are separate.

With only top level classes in the subtype system, there is no need for subtype assumptions. Instead the sub-property and sub-class relations of RDFS are used as assumption in the preorder over URIs. This preorder over URIs is used when answering queries, rather than when typing processes, as in Chapter 4. In this way the sub-property and sub-class relations are simple preorders over URIs which do not interfere with the top level classes; while the top level classes and datatypes are entirely and handled a very simple universal type system. Since both the preorder over terms and top level type system are simple to define, this is the cleanest approach to modelling a RDFS considered in this work.

5.3.3.1 Common misunderstandings about types.

Misunderstandings may arises when the two approaches to using using classes in a type system are considered, illustrated as follows. In the top level approach, a URI *person* can be assigned the type CLASS. This represents a dynamic class in which appears in term as a URI. It may be linked to and updated like any other URI. However, in the first-level approach to classes the URI *Hamish* may be assigned the atomic type PERSON which represents a class lifted to the type system. In this approach the class PERSON has been lifted to the type system, so can no longer be treated like any other URI.

A mistake is to link a dynamic class such as *person*, which is part of a term, and a static class such as PERSON, which is a type. By making this mistake the static type would be of type CLASS and the dynamic URI would be a type. The distinction between data and types becomes convoluted. The RDFS standard unfortunately makes this mistake by not distinguishing between data and types. The result is that serious paradoxes are breached, which may be partially resolved by a higher-order type system [42]. Higher-order type systems are technically complicated and add little to this application domain. Although up to fourth order types have been applied in the CyC project (a project to build an ontology of knowledge in an encyclopedia) [49]. By treating CLASS as a simple proposition, and instances as terms, these problems disappear at little cost.

5.4 The Typed Syndication Calculus

This section introduces the typed calculus. The typed calculus builds on the calculus described in previous sections. The extra type information assigns types to URIs and literals. The type system investigates the feasibility of lifting a small amount of the data to a type system. In particular, the typed calculus provides a model to evaluate the effectiveness of RDFS as a type system for Linked Data [33].

A type system allows data and updates to be statically type checked. This type system ensures that URIs assigned a distinguished rôle are always used consistently. Notice that the data formats and query system exists in major deployments [28]. Also, a preliminary update system is under development, so is considered a requirement for Linked Data. The type system for literals has

certain benefits for catching basic programming errors. However, the type system for URIs is a design decision, rather than a requirement. It depends on the application whether a type system for URIs should be used.

Although, the type system requires a design decision, the barriers imposed by the type system are less than those imposed by traditional database schema. For instance, an application may decide that a URI refers to an article, but the data associated with that article may change. An entirely new vocabulary might be used to replace the data about an article. However the URI of type article remains a URI of type article. As long as the new vocabulary allows articles to be described, then the article can still be described.

As expected from an application dependant type system, care should be taken with what data is part of the type system. For instance, a URI may be of type person. It is reasonable to assume that a person will not morph into a bat, so person is a good choice of static type. However in an application, if a person is a banker, that person may become a bar tender. In this case rôle banker is too strong to be a static type, so should remain part of the data. For flexibility, the RDFS standard can also be considered at the level of data [127].

5.4.1 Type Rules for Linked Data and Updates

RDF content and updates are typed to ensure that URIs used in RDF content are consistent with type assumptions. This section presents type rules for both RDF content and updates. The type rules for updates ensure that an update is only well typed if it updates well typed RDF content. A type rule for each construct of content is provided in Fig. 5.7. In a type judgement, the turnstile $\vdash$ separates the context on the left, represented by a type environment, from the well typed term on the right.

5.4.1.1 Type Environments for names and literals.

Type environments are finite partial functions from names to types. Syntactically, a type assignment is a name–type pair. If the pair $Alice : \textsc{person}$ occurs, the URI *Alice* is said to be assigned type PERSON. Similarly, type assignments allow variables to be assigned datatypes. The type environment is built from comma separated type assignments. Type environment composition is associative, with the empty environment as a unit, as indicated by the congruence over type environments in Fig. 5.6.

The type system uses the standard structural rules exchange and contraction. Exchange allows the order of type assignments to be changed. Contraction allows two identical assumptions can be reduced to a single assmption. For instance $Alice : \textsc{person}, Alice : \textsc{person}$ is equivalent to $Alice : \textsc{person}$. Exchange and contraction are captured by the congruence over type environments in Fig. 5.6. These structural rules are standard for type systems. The congruence can always be applied to the environment on the left of the turnstile in a type judgement.

$$
\begin{array}{rll}
\alpha ::= & a : \tau & \text{name assignment} \\
\mid & x : \textsc{Data} & \text{variable assignment} \\
\mid & \epsilon & \text{empty environment} \\
\mid & \alpha, \alpha & \text{environment composition}
\end{array}
$$

$$
\alpha, \epsilon \equiv \alpha \qquad \alpha_0, (\alpha_1, \alpha_2) \equiv (\alpha_0, \alpha_1), \alpha_2 \qquad \alpha_0, \alpha_1 \equiv \alpha_1, \alpha_0 \qquad \alpha, \alpha \equiv \alpha
$$

FIGURE 5.6: A syntax for type environments and structural rules over type environments: unit, associativity, exchange, contraction and weakening.

To ensure that environments are partial functions from URIs and variables to types and datatypes respectively, type environments must satisfy the following condition. If a URI or variable occurs in two type assignments within one type environment, then in each case the URI must be assigned the same type. Two type environments are compatible if and only if their composition still satisfies this constraint. For instance, the type environment *Alice* : PERSON and the type environment *Alice* : BOOK are incompatible. It is useful to denote the domain of a type environment α, by $\mathrm{dom}(\alpha)$.

5.4.1.2 Axioms, weakening, subsumption and literals.

The type system uses the standard axiom scheme, which states that, assuming that a URI is of a particular type, the URI is of the given type. Thus if *Ossetia* is assumed to be an article, then *Ossetia* is an article. The same shape of axiom applies to variables, but types and datatypes do not overlap so the axioms are separate.

The weaken environment rule allows unused type assignments to be added to the context. So weakening can be applied to an axiom to give, if *Ossetia* is an article and *Exchange* is an article, then *Ossetia* is an article. The subsumption rule allows the subtype system to be applied at any point. So, if *Ossetia* is an article, then *Ossetia* is anything, by the resource axiom from the subtype system. Weakening and subsumption enable the intuitive presentation of the type system in Fig. 5.7.

Data literals are defined independently from the calculus. For the purpose of examples, intuitive type judgements are assumed to hold, such as $\vdash$ '09-09-2008' : DATE or $\vdash$ `'Hamish'` : STRING. Technical details are left to the standards [25].

5.4.1.3 Type rules for triples and simple RDF content.

Predicate types indicate the subject and object of predicate. The type rules for triples ensure that the subject and object are of the correct type. For instance, suppose that *author* is of type p(ARTICLE, PERSON). If *Ossetia* is of type ARTICLE and *Hamish* is type PERSON then the triple (*Ossetia author Hamish*) is well typed.

$$a\colon \tau \vdash a\colon \tau \qquad x\colon \textsc{Data} \vdash x\colon \textsc{Data} \qquad \frac{\alpha \vdash a\colon \tau_0 \quad \alpha \vdash p\colon \mathrm{p}(\tau_0,\tau_1) \quad \alpha \vdash b\colon \tau_1}{\alpha \vdash (a\ p\ b)\ :\ \tau_0}$$

$$\frac{\alpha \vdash a\colon \tau \quad \alpha \vdash p\colon \mathrm{p}(\tau,\textsc{Data}) \quad \alpha \vdash e\colon \textsc{Data}}{\alpha \vdash (a\ p\ e)\ :\ \tau} \qquad \frac{\alpha \vdash a\colon \#\tau \quad \alpha \vdash C\colon \tau}{\alpha \vdash \mathcal{G}_a C\colon \#\tau} \qquad \frac{\alpha, a\colon \tau_0 \vdash P\colon \tau_1}{\alpha \vdash \bigwedge a\colon \tau_0.P\colon \tau_1}$$

$$\vdash \bot\colon \tau \qquad \frac{\alpha \vdash P\colon \tau \quad \alpha \vdash P\colon \tau}{\alpha \vdash P \parr P\colon \tau} \qquad \frac{\alpha_0 \vdash P\colon \tau}{\alpha_0, \alpha_1 \vdash P\colon \tau} \qquad \frac{\alpha \vdash P\colon \tau_0 \quad \tau_0 \leq \tau_1}{\alpha \vdash P\colon \tau_1}$$

FIGURE 5.7: Type rules for RDF content and named graphs: name assignment, variable assignment, type triple, type triple with literal object, type named graph, type blank node, type nothing, type par, weakening and subsumption.

Data predicates are typed using a similar rule. The data predicate type indicates the type of the subject and the datatype of the object. For instance, suppose that the predicate *name* is of type $\mathrm{p}(\textsc{person}, \textsc{String})$. Given that the name *Hamish* is of type PERSON and the literal `'Hamish'` is of data type STRING, then the triple (*Hamish name* `'Hamish'`) is well typed.

In both cases a well typed triple takes on the type of the subject. So the first triple above is of type ARTICLE and the second triple is of type PERSON. Only the type of the subject of the triple is indicated. This allows collections of triples with the same type of subject to be identified. For instance, some content may consist of triples with subjects which are articles. As noted in Sec. 5.3, typing triples according to the type of the subject is an application specific choice. The resource type can be used to indicate that the type of the subject is irrelevant.

Triples and processes are composed using par. The type rule for par allows two triples of the same type to be composed. For instance, the two triples in this section can be composed. Subtyping is applied to weaken the types of both triples to the appropriate union type.

$$\textit{Hamish}\colon \textsc{person}, \textit{Ossetia}\colon \textsc{article}, \textit{author}\colon \mathrm{p}(\textsc{article}, \textsc{person})\,, \textit{name}\colon \mathrm{p}(\textsc{person}, \textsc{String})$$

$$\vdash \begin{array}{l}(\textit{Hamish give\_name}\ \texttt{'Hamish'})\,,\\ (\textit{Ossetia author Hamish})\end{array} \ :\ \textsc{person} \cup \textsc{article}$$

The type judgement indicates that the locality contains triples which describe either people or articles.

5.4.1.4 Type rules for blank nodes.

The blank node quantifier binds names which represent blank nodes. In the typed calculus, bound names are annotated with a type information. The rule for blank nodes first types some RDF assuming that the blank node is a normal URI. The rule then internalises the type information as a quantifier.

The following example internalises three type assumptions, which represent blank nodes. Two blank nodes indicate that they are two separate events. The third blank nodes is of the resource type. The scope of the quantifier indicates that the same resource judged both events but no information is known about that resource.

$$\begin{array}{l} \mathit{judge}\colon \mathtt{p}(\top,\top)\,, \\ \mathit{date}\colon \mathtt{p}(\top,\textsc{Date}) \end{array} \vdash \bigwedge a\colon \top. \left(\begin{array}{l} \bigwedge \mathit{event1}\colon \textsc{event}. \\ \left(\begin{array}{l} (\mathit{event1}\ \mathit{judge}\ a)\,, \\ (\mathit{event1}\ \mathit{date}\ \text{`13-01-2011'}) \end{array} \right), \\ \bigwedge \mathit{event2}\colon \textsc{event}. \\ \left( \begin{array}{l} (\mathit{event2}\ \mathit{judge}\ a)\,, \\ (\mathit{event2}\ \mathit{date}\ \text{`14-01-2011'}) \end{array} \right) \end{array} \right) : \textsc{event}$$

Subjects bound by typed blank nodes help determine the type of the content. In the above example, since the subject of all triples are events, the whole resource is of type event.

5.4.1.5 Type rules for named graphs.

The type of a named graph indicates the type of content that may be contained in the named graph. For instance, a named graph of type #ARTICLE has content of type ARTICLE. The following named graph models a feed named *Caucuses*, appearing below. The four triples in the named graph have subjects which are articles, so the feed is well typed.

$$\begin{array}{l} \mathit{title}\colon \mathtt{p}(\textsc{article},\textsc{String})\,, \\ \mathit{published}\colon \mathtt{p}(\textsc{article},\textsc{Date})\,, \\ \mathit{editor}\colon \mathtt{p}(\#\textsc{article},\textsc{person})\,, \\ \mathit{Caucuses}\colon \#\textsc{article}, \\ \mathit{Hamish}\colon \textsc{person}, \\ \mathit{Ossetia}\colon \textsc{article}, \\ \mathit{exchange}\colon \textsc{article} \end{array} \vdash \begin{array}{l} (\mathit{Caucuses}\ \mathit{editor}\ \mathit{Hamish})\,, \\ \mathcal{G}_{\mathit{Caucuses}} \left(\begin{array}{l} (\mathit{Ossetia}\ \mathit{title}\ \texttt{`Ossetia invaded'})\,, \\ (\mathit{Ossetia}\ \mathit{published}\ \text{`09-09-2008'})\,, \\ (\mathit{exchange}\ \mathit{title}\ \texttt{`Stock collapse'})\,, \\ (\mathit{exchange}\ \mathit{published}\ \text{`08-10-2008'}) \end{array} \right) : \top \end{array}$$

URIs for a named graph are treated like any other URI. Thus triples can be assigned to named graphs, such as the triple which indicates the editor of the named graph in the example above.

5.4.1.6 Type rules for updates and queries.

A delete, insert or a query have the same type as the RDF content that they act on. This ensures that only RDF content which makes sense can be updated or queried. For instance, the following type judgement holds, which indicates a resource and an update which intends to replace *in* with

$$\frac{\alpha \vdash G : \tau}{\alpha \vdash |G| : \tau} \qquad \frac{\alpha \vdash G : \tau}{\alpha \vdash G^{\perp} : \tau} \qquad \frac{\alpha \vdash \phi}{\alpha \vdash \phi : \tau} \qquad \frac{\alpha \vdash S : \tau \quad \alpha \vdash T : \tau}{\alpha \vdash S \oplus T : \tau}$$

$$\frac{\alpha \vdash S : \tau \quad \alpha \vdash T : \tau}{\alpha \vdash S \otimes T : \tau} \qquad \frac{\alpha, a : \tau \vdash S : \tau}{\alpha \vdash \bigvee a : \tau . S : \tau} \qquad \frac{\alpha, x : \mathrm{D} \vdash S : \tau}{\alpha \vdash \bigvee x : \mathrm{D} . S : \tau} \qquad \frac{\alpha \vdash S : \tau}{\alpha \vdash * S : \tau}$$

FIGURE 5.8: Type rules for updates: type ask, type delete, type filter, type choice, type tensor, type select name, type select literal, type exponential.

out.

$$\begin{array}{l} \mathit{Dmitri} : \textsc{person}, \\ \mathit{status} : \mathsf{p}(\top, \top), \\ \mathit{in} : \top, \mathit{out} : \top \end{array} \vdash \left(\begin{array}{l} (\mathit{Dmitri\ status\ in}), \\ (\mathit{Dmitri\ status\ in})^{\perp} \\ (\mathit{Dmitri\ status\ out}) \end{array} \right) : \textsc{person}$$

In the example above deleted and inserted data has a subject of type person, so the update maintains the type of the context. A type checker can detect malformed triples in a delete or insert before the update is applied.

5.4.1.7 Type rules for select quantifiers.

Select quantifiers consist of a type assignment and an update. The type environment constrains the type of name to select. The example below selects a URI of type person. The type information permits the assumption that a selected URI will be of type person. The object of the triple in both the query and the insert are expected to be of type person.

$$\begin{array}{l} \mathit{article} : \textsc{article}, \\ \mathit{editor} : \mathsf{p}(\textsc{article}, \textsc{person}), \\ \mathit{club} : \top, \\ \mathit{member} : \mathsf{p}(\top, \textsc{person}) \end{array} \vdash \begin{array}{l} \bigvee p : \textsc{person} . \left(\begin{array}{l} |(\mathit{article\ editor\ p})| \\ (\mathit{club\ member\ p}) \end{array} \right), \\ \bigwedge \mathit{Hamish} : \textsc{person} . (\mathit{article\ editor\ Hamish}) \end{array} : \top$$

The above update is in the presence of some data where a blank node appears. The type assignment for the blank node is the same as the type assignment for the select quantifier.

5.4.1.8 Type rules for literals in filters and selects.

A constraint which contains variables or names can be typed. For instance, under the assumption that x: DATE, constraint $x \leq$ '01-01-1950' is well typed. A date literal substituted for x results in a constraint such as '01-05-1886' $\leq$ '01-01-1950', which is also well typed. In the example below, the select quantifier introduces the assumption that x is a date. This type assumption allows the filter and triple in the query to be typed. The update satisfies the following type

judgement.

$$\begin{array}{l} \mathit{Kidnapped}\colon \textsc{book},\\ \mathit{published}\colon \mathsf{p}(\textsc{book}, \textsc{Date}),\\ \mathit{note}\colon \mathsf{p}(\top, \top),\\ \mathit{classic}\colon \top \end{array} \quad \vdash \quad \begin{array}{l} \bigvee x\colon \textsc{Date}.\bigvee \mathit{book}\colon \textsc{book}.\\ \left(\begin{array}{l} (x \leq \text{`01-01-1950'})\\ |(\mathit{book\ published\ x})|\\ (\mathit{book\ note\ classic}) \end{array}\right), \qquad : \textsc{book}\\ (\mathit{Kidnapped\ published}\ \text{`01-05-1886'}) \end{array}$$

Typing literals is the minimal type system for updates. Literals can still be typed without application specific type information for URIs.

5.4.1.9 Type rules for tensor, choice and iteration.

The tensor product ensure that two well typed updates are applied atomically. A choice between two well typed updates is presented. A tensor or choice assumes a type that both components can assume. Iteration does not affect the type of a process. In the following example all components are of type person so the whole update is of type person.

$$\begin{array}{l} \mathit{guard}\colon \textsc{class},\\ \mathit{attendant}\colon \textsc{class},\\ \mathit{porter}\colon \textsc{class},\\ \mathit{type}\colon \mathsf{p}(\top, \textsc{class}) \end{array} \quad \vdash \quad \begin{array}{l} *\bigvee a\colon \textsc{person}.\left(\begin{array}{l} \left(\begin{array}{l} (a\ \mathit{type\ attendant})^{\perp} \end{array}\right)\\ \oplus\\ \left(\begin{array}{l} (a\ \mathit{type\ guard})^{\perp} \end{array}\right)\\ (a\ \mathit{type\ porter}) \end{array}\right), \qquad : \textsc{person}\\ \bigvee b\colon \textsc{person}.(b\ \mathit{type\ attendant}),\\ \bigvee c\colon \textsc{person}.(c\ \mathit{type\ guard}) \end{array}$$

The above example demonstrates a mix of static classes as types and dynamic classes as data. The people are always people, but their rôle changes.

5.4.2 Algorithmic Typing for the Calculus

In the type system in the previous section, the subsumption and weakening rules can be applied at any point. An algorithmic type system controls the use of subsumption and weakening. Subsumption can instead be applied as early as possible. Weakening can be applied as late as possible. The algorithmic type system can be less intuitive but is syntax directed, so easier to work with for proofs and type inference algorithms [101].

Key differences between the rules of the type system and the algorithmic type system are presented in Fig. 5.9. The first variation is that the axioms immediately weaken the type to the correct type required. The second variation is that, when two terms are composed, the type environments are merged whenever they are compatible. This is characterised by the type rule for the tensor product. Merging environments avoids weakening both environments before updates

$$\frac{\tau_0 \leq \tau_1}{a\colon \tau_0 \Vdash a\colon \tau_1} \qquad \frac{\alpha + a\colon \tau_1 \Vdash P\colon \tau_0}{\alpha \Vdash \bigwedge a\colon \tau_1.P\colon \tau_0}\ a \in \mathrm{fn}(P) \qquad \frac{\alpha \Vdash P\colon \tau_0}{\alpha \Vdash \bigwedge a\colon \tau_1.P\colon \tau_0}\ a \notin \mathrm{fn}(P)$$

$$\frac{\alpha_0 \Vdash U\colon \tau \quad \alpha_1 \Vdash V\colon \tau}{\alpha_0, \alpha_1 \Vdash U \otimes V\colon \tau} \qquad \frac{\alpha_0 \Vdash a\colon \tau_0 \quad \alpha_1 \Vdash p\colon \mathrm{p}(\tau_0, \tau_1) \quad \alpha_2 \Vdash b\colon \tau_1 \quad \tau_0 \leq \tau_2}{\alpha_0, \alpha_1, \alpha_2 \Vdash (a\ p\ b)\ :\ \tau_2}$$

FIGURE 5.9: Variations in rules for the algorithmic type system.

are combined. The third variation is that the blank node and select rules, which internalise the type environment permit weakening of the environment. This permitted weakening is expressed using the congruence over environments. Exchange and contraction still apply and + indicates disjoint environments.

The soundness and completeness of the algorithmic type system with respect to the intuitive type system ensures that results carry from one system to the other. The proof begins with a technical lemma. The lemma demonstrates that, for the algorithmic type system, the environment on the left of the turnstile covers exactly the URIs that occur free in the term.

Lemma 5.2. *If* $\alpha \Vdash P\colon \tau$ *then* $\mathrm{fn}(P) = \mathrm{dom}(\alpha)$.

Soundness of the algorithmic type system is established by a straightforward rewrite from an algorithmic type tree to a normal type tree. The effect of the typing is preserved by the rewrite.

Theorem 5.3 (Soundness of algorithmic typing)**.** *If* $\alpha \Vdash P\colon \tau$ *then* $\alpha \vdash P\colon \tau$.

Proof. Soundness is established by a straight forward translation of proof trees. Each algorithmic type rule which involves subtypes can be replaced by their equivalent type rule followed by a subsumption link.

For blank nodes, if $a \notin \mathrm{fn}(P)$ then the algorithmic type rule is transformed into the type rule, preceded by application of weakening, as follows.

$$\frac{\begin{array}{c}\pi\\ \alpha \Vdash P\colon \tau_0\end{array}}{\alpha \Vdash \bigwedge a\colon \tau_1.P\colon \tau_0} \quad \text{yields} \quad \frac{\begin{array}{c}\pi\\ \alpha \vdash P\colon \tau_0\end{array}}{\dfrac{\alpha_0, a\colon \tau_1 \vdash P\colon \tau_0}{\alpha_0 \vdash \bigwedge a\colon \tau_1.P\colon \tau_0}}$$

Hence, each algorithmic type tree corresponds to a type tree with the same conclusion. □

The proof of completeness of the algorithmic type system is a transformation of proof trees which pushes subsumption towards the leaves and weakening towards the root of a type tree.

Theorem 5.4 (Completeness of algorithmic subtyping)**.** *If* $\alpha \vdash P\colon \tau$*, then there exist* α_0, α_1 *such that* $\alpha_0, \alpha_1 \equiv \alpha$ *and* $\alpha_0 \Vdash P\colon \tau$.

Proof. The transformation $[\![\cdot]\!]$ pushes subsumption rules as deep as possible into the proof tree and suspends weakening.

There are two special cases. If two subsumption links appear consecutively, then they can be performed in a single subsumption link using cut in the subtype system. Also, weakening rules can be deleted, since weakening is controlled by the induction hypothesis.

For axioms, subsumption is absorbed by the algorithmic rule.

$$\left[\!\!\left[\dfrac{a\colon \tau_0 \vdash a\colon \tau_0 \qquad \overset{\pi}{\vdash \tau_0 \leq \tau_1}}{a\colon \tau_0 \vdash a\colon \tau_1} \right]\!\!\right] \text{ yields } \dfrac{\overset{\pi}{\vdash \tau_0 \leq \tau_1}}{a\colon \tau_0 \Vdash a\colon \tau_1}$$

For tensor, the subsumption link is pushed up each branch. By structural induction, α_0, α_1 are type environments, such that $\alpha \equiv \alpha_0, \alpha_0'$ and $\alpha \equiv \alpha_1, \alpha_1'$ and also each forms the premises of the conclusions of the respective branches of the resulting tree.

$$\left[\!\!\left[\dfrac{\dfrac{\overset{\pi_0}{\alpha \vdash P\colon \tau_0} \qquad \overset{\pi_1}{\alpha \vdash Q\colon \tau_0}}{\alpha \vdash P \otimes Q\colon \tau_0} \qquad \dfrac{\overset{\pi_2}{\vdash \tau_0 \leq \tau_1}}{\alpha \vdash P \otimes Q\colon \tau_1}}{\alpha \vdash P \otimes Q\colon \tau_1} \right]\!\!\right]$$

yields

$$\dfrac{\alpha_0 \Vdash P\colon \tau_1 \quad \alpha_1 \Vdash Q\colon \tau_1}{\alpha_0, \alpha_1 \Vdash P \otimes Q\colon \tau_1}$$

where

$$\alpha_0 \Vdash P\colon \tau_1 = \left[\!\!\left[\dfrac{\overset{\pi_0}{\alpha \vdash P\colon \tau_0} \qquad \overset{\pi_2}{\vdash \tau_0 \leq \tau_1}}{\alpha \vdash P\colon \tau_1} \right]\!\!\right]$$

and

$$\alpha_1 \Vdash Q\colon \tau_1 = \left[\!\!\left[\dfrac{\overset{\pi_1}{\alpha \vdash Q\colon \tau_0} \qquad \overset{\pi_2}{\vdash \tau_0 \leq \tau_1}}{\alpha \vdash Q\colon \tau_1} \right]\!\!\right]$$

The triple rule absorbs a subsumption link. By induction, there exist type environments $\alpha_0, \alpha_1, \alpha_2$ such that $\alpha \equiv \alpha_0, \alpha_1, \alpha_2, \alpha'$ and the following transformation holds.

$$\left[\!\!\left[\dfrac{\dfrac{\overset{\pi_0}{\alpha \vdash a\colon \tau_0} \quad \overset{\pi_1}{\alpha \vdash p\colon \mathfrak{p}(\tau_0, \tau_1)} \quad \overset{\pi_2}{\alpha \vdash b\colon \tau_1}}{\alpha \vdash (a\ p\ b)\ :\ \tau_0} \qquad \overset{\pi_3}{\vdash \tau_0 \leq \tau_2}}{\alpha \vdash (a\ p\ b)\ :\ \tau_2} \right]\!\!\right]$$

yields

$$\dfrac{\alpha_0 \Vdash a\colon \tau_0 \quad \alpha_1 \Vdash p\colon \mathfrak{p}(\tau_0, \tau_1) \quad \alpha_2 \Vdash b\colon \tau_1 \quad \overset{\pi_3}{\vdash \tau_0 \leq \tau_2}}{\alpha_0, \alpha_1, \alpha_2 \Vdash (a\ p\ b)\ :\ \tau_2}$$

For blank nodes, subsumption is pushed straight up the tree. Consider the following proof tree.

$$\left[\!\!\left[\begin{array}{c} \pi_0 \\ \alpha, a\colon \tau_2 \vdash P\colon \tau_0 \\ \hline \alpha \vdash \bigwedge a\colon \tau_2.P\colon \tau_0 \end{array} \quad \begin{array}{c} \pi_1 \\ \vdash \tau_0 \leq \tau_1 \\ \hline \end{array} \right]\!\!\right] \\ \alpha \vdash \bigwedge a\colon \tau_2.P\colon \tau_1$$

By induction, there is some α_1 and α' such that the following transformation holds, where $\alpha_1, \alpha' \equiv \alpha, a\colon \tau_2$.

$$\left[\!\!\left[\begin{array}{cc} \pi_0 & \pi_1 \\ \alpha, a\colon \tau_2 \vdash P\colon \tau_0 & \vdash \tau_0 \leq \tau_1 \\ \hline \alpha, a\colon \tau_2 \vdash P\colon \tau_1 & \end{array}\right]\!\!\right] = \begin{array}{c} \pi_3 \\ \alpha_1 \Vdash P\colon \tau_1 \end{array}$$

If $a \notin \mathrm{fn}(P)$, then the following proof tree holds.

$$\begin{array}{c} \pi_3 \\ \alpha_1 \Vdash P\colon \tau_1 \\ \hline \alpha_1 \Vdash \bigwedge a\colon \tau_2.P\colon \tau_1 \end{array}$$

If $a \in \mathrm{fn}(P)$ then, by Lemma 5.2 there exists α_1' such that $\alpha_1 \equiv \alpha_1' + a\colon \tau_2$, and the following holds.

$$\begin{array}{c} \pi_3 \\ \alpha_1' + a\colon \tau_2 \Vdash P\colon \tau_1 \\ \hline \alpha_1' \Vdash \bigwedge a\colon \tau_2.P\colon \tau_1 \end{array}$$

Further cases are similar to the above. Thus by induction over the proof trees a transformation from any type tree to an algorithmic type tree exists. □

The algorithmic type system demonstrates that type checking is syntax directed. Even for a light type system, type checking a store at each operational step is costly. A feasible approach is to type check updates.

5.5 The Typed Operational Semantics

Given a well typed update, the expectation is that Linked Data need only be typed once. Well typed updates applied to well typed Linked Data should result in well typed Linked Data, without the need to recheck the Linked Data. The light type system works locally, in the sense that the correctness of one triple is not affected by other triples. Similarly, the commitment relation over updates describes the local behaviour of updates, since unused triples are ignored. The type system and commitment relations therefore work at the same level of granularity, so are compatible.

$$P \parr \bot \equiv P \qquad P \parr (Q \parr R) \equiv (P \parr Q) \parr R \qquad P \parr Q \equiv Q \parr P$$

$$\mathcal{G}_a(C \parr D) \equiv \mathcal{G}_a C \parr \mathcal{G}_a D \qquad \bigwedge a\colon \tau.(P \parr Q) \equiv \bigwedge a\colon \tau.P \parr Q \quad a \notin \mathrm{fn}\,(Q)$$

$$\bigwedge a\colon \tau.\bot \equiv \bot \qquad \wedge a\colon \tau_0.\wedge b\colon \tau_1.P \equiv \wedge b\colon \tau_1.\wedge a\colon \tau_0.P \quad a \neq b \text{ or } \tau_0 = \tau_1$$

FIGURE 5.10: The structural congruence over content and processes: unit, associativity, commutativity, split named graph, distribute blank node, eliminate blank node and commute blank node.

This section demonstrates that the specification of atomic commitments can be extended to ensure that the type system and the commitment relation are compatible. The typed commitment rules introduce minimal assumptions about the context. The assumptions about the context are that names selected in an update are of the correct type. This amounts to a minimal dynamic type check on selected names. Under minimal assumptions about the context, type judgements are preserved by the dynamically typed commitment relation, as verified by Theorem 5.7.

The typed operational semantics are defined by combining the following components. α-conversion of bound names, the structural congruence in Fig. 5.10 and a typed commitment relation in Fig. 5.11. Examples throughout this section illustrate the operational behaviour of typed updates.

5.5.1 The Structural Congruence for Typed Linked Data

The structural congruence for typed content is gathered in Fig. 5.10. The structural congruence captures the commutative monoid formed by par and nothing, which is used for both RDF content and processes. The structural congruence allows blank node quantifiers to distribute over tensor and be eliminated in the presence of nothing. Compatible blank node quantifiers may be swapped. The side condition for swapping type assignments ensures that the composition of the assignments form a partial function. Type environments must be partial functions. The split named graph rule allows named graphs to be decomposed for fine grained updates.

The first type preservation result verifies that the structural congruence preserves types. Thus given a well typed process, processes structurally congruent to the process are well typed. Lemma 5.5 verifies this compatibility between the structural congruence and the type system. The proof makes use of algorithmic typing to simplify proofs.

Lemma 5.5 (Structural congruence preserves types)**.** *Assuming that* $P \equiv Q$*,* $\alpha \vdash P\colon \tau$ *if and only if* $\alpha \vdash Q\colon \tau$*.*

Proof. For each algorithmic type tree and rule of structural congruence, an algorithmic type tree of the equivalent process can be constructed, by Theorems 5.3 and 5.4.

For distributivity of blank nodes over par, assume that $a \notin \mathrm{fn}(Q)$. Also assume that $a \in \mathrm{fn}(P)$ and the following proof tree holds.

$$\dfrac{\dfrac{\alpha_0 + a\colon \tau_0 \Vdash P\colon \tau_1}{\alpha_0 \Vdash \bigwedge a\colon \tau_0.P\colon \tau_1} \quad \alpha_1 \Vdash Q\colon \tau_1}{\alpha_0, \alpha_1 \Vdash \bigwedge a\colon \tau_0.P \parr Q\colon \tau_1}$$

Now $a \notin \mathrm{dom}(\alpha_1)$, by Lemma 5.2, and $a \notin \mathrm{dom}(\alpha_0)$ thus $a \notin \mathrm{dom}(\alpha_0, \alpha_1)$, so the following proof tree holds. The converse is immediate.

$$\dfrac{\dfrac{\alpha_0 + a\colon \tau_0 \Vdash P\colon \tau_1 \quad \alpha_1 \Vdash Q\colon \tau_1}{(\alpha_0, \alpha_1) + a\colon \tau_0 \Vdash P \parr Q\colon \tau_1}}{\alpha_0, \alpha_1 \Vdash \bigwedge a\colon \tau_0.(P \parr Q)\colon \tau_1}$$

Now, assume that $a \notin \mathrm{fn}(P)$ and the following proof tree holds. Clearly $a \notin \mathrm{fn}(P \parr Q)$ so the following proof trees can be interchanged.

$$\dfrac{\dfrac{\alpha_0 \Vdash P\colon \tau_1}{\alpha_0 \Vdash \bigwedge a\colon \tau_0.P\colon \tau_1} \quad \alpha_1 \Vdash Q\colon \tau_1}{\alpha_0, \alpha_1 \Vdash \bigwedge a\colon \tau_0.P \parr Q\colon \tau_1}$$

iff

$$\dfrac{\dfrac{\alpha_0 \Vdash P\colon \tau_1 \quad \alpha_1 \Vdash Q\colon \tau_1}{\alpha_0, \alpha_1 \Vdash P \parr Q\colon \tau_1}}{\alpha_0, \alpha_1 \Vdash \bigwedge a\colon \tau_0.(P \parr Q)\colon \tau_1}$$

Remaining cases are straight forward. The result follows by induction over the derivation of an equivalence. □

The structural congruence covers the reorganisation of content and processes. The structural congruence is always reversible. In contrast, the effect of updates are generally irreversible, so are captured by a commitment relation.

5.5.2 Typed Atomic Commitments

Atomic commitments were introduced in Sec. 3.2 to specify an operational semantics for queries and updates over Linked Data. In Sec. 3.3, atomic commitments were extended to cover key features for syndication. In this section, atomic commitments are extended with a type environment, called the context. Otherwise, the rôle of atomic commitments remains the same. The process on the left indicates exactly the processes consumed. The process on the right indicates the exact processes which replace the processes consumed.

The context for typed atomic commitments indicates a minimal dynamic type check required by a commitment. By minimising dynamic type checks, a feasible type system is enabled. The context represents these minimal type checks as a type environment. Any processes in the vicinity of the commitment must agree on the the assignments of names to types in the context.

$$\vdash C \parr C^{\perp} \rhd \bot \qquad \vdash C \rhd C \qquad \vdash C \parr |C| \rhd C \qquad \vdash \frac{\vDash \phi}{\phi \rhd \bot}$$

$$\frac{\alpha \vdash P \parr U \rhd Q}{\alpha \vdash P \parr (U \oplus V) \rhd Q} \qquad \frac{\alpha \vdash P \parr V \rhd Q}{\alpha \vdash P \parr (U \oplus V) \rhd Q} \qquad \frac{\alpha_0 \vdash P \parr U \rhd P' \quad \alpha_1 \vdash Q \parr V \rhd Q'}{\alpha_0 + \alpha_1 \vdash P \parr Q \parr (U \otimes V) \rhd P' \parr Q'}$$

$$\frac{\alpha \vdash P \parr U \rhd Q}{\alpha \vdash P \parr *U \rhd Q} \qquad \frac{\alpha \vdash P \parr (*U \otimes *U) \rhd Q}{\alpha \vdash P \parr *U \rhd Q} \qquad \vdash *U \rhd \bot$$

$$\frac{\alpha \vdash P \parr U\left\{{}^{b}/_{a}\right\} \rhd Q \quad \vdash \tau_0 \leq \tau_1}{\alpha + b\colon \tau_0 \vdash P \parr \bigvee a\colon \tau_1.U \rhd Q} \qquad \frac{\alpha \vdash P \parr U\{{}^{v}/_{x}\} \rhd Q \quad \vdash v\colon \mathrm{D}}{\alpha \vdash P \parr \bigvee x\colon \mathrm{D}.U \rhd Q}$$

$$\frac{\alpha_0 \vdash P \rhd P' \quad \alpha_1 \vdash Q \rhd Q'}{\alpha_0, \alpha_1 \vdash P \parr Q \rhd P' \parr Q'} \qquad \frac{\alpha + a\colon \tau \vdash P \parr Q \rhd P' \parr Q'}{\alpha \vdash \bigwedge a\colon \tau.P \parr Q \rhd \bigwedge a\colon \tau.P' \parr Q'}\ a \notin \mathrm{fn}(Q, Q')$$

$$\frac{\alpha + a\colon \tau \vdash \mathcal{G}_b P \parr Q \rhd \mathcal{G}_b P' \parr Q'}{\alpha \vdash \mathcal{G}_b \bigwedge a\colon \tau.P \parr Q \rhd \mathcal{G}_b \bigwedge a\colon \tau.P' \parr Q'}\ a \notin \mathrm{fn}(Q, Q', b)$$

FIGURE 5.11: The axioms and rules form atomic commitments: delete axiom, insert axiom, query axiom, choose left rule, choose right rule, tensor rule, filter axiom, dereliction rule, contraction rule, weakening axiom, select name rule, select literal rule, mix rule, blank node rule, named graph rule.

For instance, a context for a commitment relation may indicate that the name *Burns* is of type WRITER. However, if it is assumed elsewhere that *Burns* is a PERSON, then the commitment cannot be applied, since the required context indicates a stronger type. The rules for typed commitments are presented in Fig. 5.11. The higher-order π-calculus similarly constrains the context of a transition using type environments [76].

5.5.2.1 Type safe commitments.

Assuming that a process is well typed, a commitment which only uses axioms requires no dynamic type checks. When there are no type checks most other rules behave like the untyped calculus. For instance, in the example below the deletes and inserts have an empty context, so their tensor product has an empty context.

$$\vdash \left(\begin{array}{l} (\mathit{studio\ status\ closed})^{\perp} \\ (\mathit{studio\ status\ open}) \\ \mathcal{G}_{\mathit{studio}}(\mathit{Dmitri\ status\ out})^{\perp} \\ \mathcal{G}_{\mathit{studio}}(\mathit{Dmitri\ status\ in}) \end{array}\right), \begin{array}{l} (\mathit{studio\ status\ closed}), \\ \mathcal{G}_{\mathit{studio}}(\mathit{Dmitri\ status\ out}) \end{array} \rhd \begin{array}{l} (\mathit{studio\ status\ open}), \\ \mathcal{G}_{\mathit{studio}}(\mathit{Dmitri\ status\ in}) \end{array}$$

Because the context above is empty, any environment which types the process before the commitment also types the process after the commitment.

5.5.2.2 The dynamically typed select quantifier.

The select quantifier introduces the need for dynamic type checks. A select quantifier annotates a name with a type. The type annotation imposes an upper bound on the type of the selected name. For instance, the select quantifier below requires that the selected name is of type PERSON. The commitment selects the name *Dmitri* according to the triple to be deleted. However, the given process does not indicate that *Dmitri* is of type PERSON. The missing assumption is indicated by the context in front of the commitment.

$$Dmitri\colon \textsc{person} \vdash \bigvee a\colon \textsc{person}.(Hamish\ knows\ a)^{\perp}, (Hamish\ knows\ Dmitri) \rhd \perp$$

The context above indicates that the commitment can only be applied safely when *Dmitri* is of type PERSON. Further information required to type the process, such as *knows* is of type p(PERSON, PERSON) and *Hamish* is of type PERSON, is not required for the commitment.

5.5.2.3 The tensor product of commitments with non-empty context.

The tensor product is used to synchronise updates. If two updates each require a context, then the tensor product of the updates composes the contexts. For instance, the following example consists of two commitments where each requires a URI to be of type person. The commitments are composed using the tensor product, so the context indicates that both URIs are of type person.

$$\begin{array}{l} user1\colon \textsc{person}, \\ user2\colon \textsc{person} \end{array} \vdash \begin{array}{l} (user1\ status\ busy), \\ (user2\ status\ ready), \end{array} \left(\begin{array}{l} \bigvee a\colon \textsc{person}. \\ \left(\begin{array}{l} (a\ status\ busy)^{\perp} \\ (a\ status\ ready) \end{array} \right) \\ \bigvee b\colon \textsc{person}. \\ \left(\begin{array}{l} (b\ status\ ready)^{\perp} \\ (b\ status\ busy) \end{array} \right) \end{array} \right) \rhd \begin{array}{l} (user1\ status\ ready), \\ (user2\ status\ busy) \end{array}$$

In the above example, the names in the context are distinct. The tensor rule forces combined contexts to be disjoint. By forcing disjoint contexts, two select quantifiers cannot discover the same name. Consequently, the more controlled 'select distinct' quantifier is modelled from SPARQL Query [115].

In contrast, the tensor rule in Sec. 3.2 models the normal select quantifier in SPARQL Query. The normal select allows different selects to discover the same name. The normal quantifier can be achieved here by removing the constraint that contexts combined using the tensor product are disjoint. Removing the constraint allows the same name to appear in the combined environment, hence contraction may be applied. Contraction allows two different select quantifiers to share the same resource. The two variations on the select quantifier may coexist by extending the type environment in the calculus. A more subtle type environment can control the use of contraction, as investigated in the logic of bunched implications [106].

5.5.2.4 Dynamic type checks for selected literals.

The select literal quantifier annotates a variable with a data type. The annotation constrains the type of a literal discovered using the select literal rule. To enforce the constraint, the select literal rule dynamically type checks the selected literal. In the example below, the literal input by the select quantifier is successfully checked to be a date. The syntax of the literal is enough information to check the type.

$$
\begin{array}{l}
\mathit{Kidnapped}\colon \textsc{book} \\
\quad \vdash \quad \bigvee x\colon \textsc{Date}.\bigvee \mathit{book}\colon \textsc{book}. \\
\qquad \left(\begin{array}{l} (x \leq \text{`01-01-1950'}) \\ |(\mathit{book\ published\ x})| \\ (\mathit{book\ status\ classic}) \end{array}\right), \\
\qquad (\mathit{Kidnapped\ published}\ \text{`01-05-1886'})
\end{array}
\quad \rhd \quad
\begin{array}{l}
(\mathit{Kidnapped\ published}\ \text{`01-05-1886'}), \\
(\mathit{Kidnapped\ status\ classic}),
\end{array}
$$

The select literal performs the dynamic type check immediately. No further information about the literal is required from the environment. In contrast, there is not enough information to check the name *Kidnapped* is a book. This minimum requirement placed on the context is indicated by the type environment.

5.5.2.5 Typed Commitments involving Choice.

The branches in a choice may depend on different contexts. In the update below a person and a string are always selected. The string is immediately type checked and the check for the person is indicated by the context. The update features a third select which demands a name of type place, but alternatively offers the choice of the unit update. In the commitment below the unit branch is chosen, so the third select does not contribute to the context.

$$
\begin{array}{l}
\mathit{Burns}\colon \textsc{person} \\
\vdash \quad \bigvee x\colon \textsc{String}.\bigvee a\colon \textsc{person}. \\
\left(\begin{array}{l}
|(a\ \mathit{email}\ x)| \\
\mathcal{G}_{\mathit{poets}}(a\ \mathit{email}\ x) \\
\left(\begin{array}{l}
\bigvee h\colon \textsc{place}. \\
\left(\begin{array}{l} |(a\ \mathit{home}\ h)| \\ \mathcal{G}_{\mathit{poets}}(a\ \mathit{home}\ h) \end{array}\right) \oplus \mathrm{I}
\end{array}\right)
\end{array}\right), \\
(a\ \mathit{email}\ \texttt{`Rabbie@soton.ac.uk'})
\end{array}
\quad \rhd \quad
\begin{array}{l}
(\mathit{Burns\ address}\ \texttt{`Rabbie@soton.ac.uk'}), \\
\mathcal{G}_{\mathit{poets}}(\mathit{Burns\ address}\ \texttt{`Rabbie@soton.ac.uk'})
\end{array}
$$

The choice between an update and the unit update models the operator `OPTIONAL` in SPARQL Query [115, 110]. Since the unit update is always enabled, the other branch may always be ignored so is optional. This demonstrates that, firstly, optional is not primitive and, secondly, optional works for updates. In related work, optional is borrowed from relational algebra for modelling queries [43].

5.5.2.6 Iterated updates and dynamic types.

Iteration allows multiple copies of an update to be applied. For instance, the following iteration creates two copies of the inner update. Due to the use of tensor in the contraction rule, selected names are forced to be disjoint. In the following example, the context demands three disjoint names of type person.

$$
\begin{array}{l}
\mathit{Alice}\colon \textsc{person}, \mathit{Bob}\colon \textsc{person}, \mathit{Chris}\colon \textsc{person} \\
\vdash \quad \bigvee a\colon \textsc{person}. \\
\quad \left(\begin{array}{l} |(a\ \mathit{type}\ \mathit{journalist})| \\ *\bigvee b\colon \textsc{person}. \\ \quad \left(\begin{array}{l} |(b\ \mathit{type}\ \mathit{photographer})| \\ (a\ \mathit{knows}\ b) \end{array} \right) \end{array} \right), \\
\quad (\mathit{Alice}\ \mathit{type}\ \mathit{journalist}), \\
\quad (\mathit{Bob}\ \mathit{type}\ \mathit{photographer}), \\
\quad (\mathit{Chris}\ \mathit{type}\ \mathit{photographer})
\end{array}
\quad \rhd \quad
\begin{array}{l}
(\mathit{Alice}\ \mathit{type}\ \mathit{journalist}), (\mathit{Alice}\ \mathit{knows}\ \mathit{Bob}), \\
(\mathit{Alice}\ \mathit{knows}\ \mathit{Chris}), (\mathit{Bob}\ \mathit{type}\ \mathit{photographer}), \\
(\mathit{Chris}\ \mathit{type}\ \mathit{photographer})
\end{array}
$$

Now suppose that the whole of the above update is also iterated. Due to the disjunction of environments forced by the tensor product, each journalist is assigned distinct photographers. To allow names to be shared the tensor rule can be relaxed, as discussed above.

5.5.2.7 Commitments for typed blank nodes.

The blank node rule allows a blank node to be used in place of a URI. The typed blank node also indicates a lower bound on the type of URI the blank node can represent. For instance, the example below involves a blank node quantifier annotated with type person. The query demands a name of any type, so the assumption that the blank node is of type person is strong enough for the following commitment.

$$
\begin{array}{l}
\bigvee b\colon \top. \\
\vdash \quad \left(\begin{array}{l} |(b\ \mathit{name}\ \texttt{`Burns'})| \\ (\mathit{Dmitri}\ \mathit{knows}\ b) \end{array} \right),
\end{array}
\quad
\begin{array}{l}
\bigwedge a\colon \textsc{person}. \\
(a\ \mathit{name}\ \texttt{`Burns'})
\end{array}
\quad \rhd \quad
\begin{array}{l}
\bigwedge a\colon \textsc{person}. \\
\left(\begin{array}{l} (a\ \mathit{name}\ \texttt{`Burns'}), \\ (\mathit{Dmitri}\ \mathit{knows}\ a) \end{array} \right)
\end{array}
$$

The context is used to ensure that type of the select quantifier and the blank node quantifier match. The select quantifier introduces to the context an assignment of a name to type person. The blank node rule eliminates that assignment from the context. In the above example, this leaves an empty context so no dynamic checks are required.

5.5.3 Type Preservation for Commitments

Type preservation verifies that given a well typed process the resulting process after a commitment is well typed with respect to the same environment. This means that the use of a URI after

an update is consistent with the use of a URI before the update. For a commitment relation with a non-empty context, the context must agree with the type environment used to type the process. The following substitution lemma is required for selected names and literals.

Lemma 5.6 (Substitution preserves types)**.** *For names,*

$$\text{if } \alpha, a\colon \tau_1 \vdash U\colon \tau \text{ and } \tau_0 \leq \tau_1, \text{ then } \alpha, b\colon \tau_0 \vdash U\left\{{}^{b}/_{a}\right\}\colon \tau.$$

Similarly for literals,

$$\text{if } \alpha, x\colon \mathrm{D} \vdash U\colon \tau \text{ and } \vdash v\colon \mathrm{D}, \text{ then } \alpha \vdash U\{{}^{v}/_{x}\}\colon \tau.$$

The proof of the lemma follows by structural induction. The type preservation theorem also uses type preservation of the structural congruence, Lemma 5.5. The soundness and completeness of the algorithmic type system eliminate the need to consider subsumption and weakening rules, Theorems 5.3 and 5.4.

Theorem 5.7 (Commitments preserve types)**.** *If $\alpha_0 \vdash P \rhd Q$, then $\alpha_0, \alpha_1 \vdash P\colon \tau$ yields that $\alpha_0, \alpha_1 \vdash Q\colon \tau$.*

Proof. The axioms are immediate. The structural induction proof for choose, tensor, select and blank nodes are demonstrated.

Consider the choose rule and assume that the following type tree holds.

$$\dfrac{\alpha_0 \vdash P\colon \tau \qquad \dfrac{\alpha_1 \vdash U\colon \tau \quad \alpha_2 \vdash V\colon \tau}{\alpha_1, \alpha_2 \vdash U \oplus V\colon \tau}}{\alpha_0, \alpha_1, \alpha_2 \vdash P \parr (U \oplus V)\colon \tau}$$

Therefore the following type tree holds.

$$\dfrac{\alpha_0 \vdash P\colon \tau \quad \alpha_1 \vdash U\colon \tau}{\alpha_0, \alpha_1 \vdash P \parr U\colon \tau}$$

Now assume that the choose left rule is used to resolve a commitment, where $\alpha, \alpha' \equiv \alpha_0, \alpha_1, \alpha_2$.

$$\dfrac{\alpha \vdash P \parr U \rhd Q}{\alpha \vdash P \parr (U \oplus V) \rhd Q}$$

By induction, $\alpha \vdash P \parr U \rhd Q$ and $\alpha_0, \alpha_1 \vdash P \parr U\colon \tau$ yields the following type judgement, as required.

$$\alpha_0, \alpha_1, \alpha_2 \vdash Q\colon \tau$$

Consider the tensor rule and suppose that the following type judgement holds.

$$\frac{\dfrac{\alpha_0 \vdash P : \tau \quad \alpha_1 \vdash Q : \tau}{\alpha_0, \alpha_1 \vdash P \parr Q : \tau} \quad \dfrac{\alpha_2 \vdash U : \tau \quad \alpha_3 \vdash V : \tau}{\alpha_2, \alpha_3 \vdash U \otimes V : \tau}}{\alpha_0, \alpha_1, \alpha_2, \alpha_3 \vdash P \parr Q \parr (U \otimes V) : \tau}$$

Hence the following two judgements hold.

$$\alpha_0, \alpha_2 \vdash P \parr U : \tau \quad \text{and} \quad \alpha_1, \alpha_3 \vdash Q \parr V : \tau$$

Now, assume that the following commitment holds, where $\alpha_0, \alpha_2 \equiv \beta_0, \beta_0'$ and $\alpha_1, \alpha_3 \equiv \beta_1, \beta_1'$.

$$\frac{\beta_0 \vdash P \parr U \rhd P' \quad \beta_1 \vdash Q \parr V \rhd Q'}{\beta_0 + \beta_1 \vdash P \parr Q \parr (U \otimes V) \rhd P' \parr Q'}$$

Hence by induction, the following type judgement holds, as required.

$$\frac{\alpha_0, \alpha_2 \vdash P' : \tau \quad \alpha_1, \alpha_3 \vdash Q' : \tau}{\alpha_0, \alpha_1, \alpha_2, \alpha_3 \vdash P' \parr Q' : \tau}$$

Consider the select rule and suppose that the following type tree holds.

$$\frac{\alpha_0 \Vdash P : \tau \quad \dfrac{\alpha_1 + a : \tau_1 \Vdash U : \tau}{\alpha_1 \Vdash \bigvee a : \tau_1 . U : \tau}}{\alpha_0, \alpha_1 \Vdash P \parr \bigvee a : \tau_1 . U : \tau}$$

Assuming that $\tau_0 \leq \tau_1$, by the substitution lemma, $\alpha_1 + a : \tau_1 \vdash U : \tau$ yields $\alpha_1, b : \tau_0 \vdash U\left\{{}^{b}/_{a}\right\} : \tau$, so the following proof tree can be constructed.

$$\frac{\alpha_0 \vdash P : \tau \quad \alpha_1, b : \tau_0 \vdash U\left\{{}^{b}/_{a}\right\} : \tau}{\alpha_0, \alpha_1, b : \tau_0 \vdash P \parr U\left\{{}^{b}/_{a}\right\} : \tau}$$

Also, assume that the following commitment holds, where $\alpha_0, \alpha_1 \equiv \alpha, \alpha'$, for some α'.

$$\frac{\alpha \vdash P \parr U\left\{{}^{b}/_{a}\right\} \rhd Q}{\alpha + b : \tau_0 \vdash P \parr \bigvee a : \tau_1 . U \rhd Q}$$

By the induction hypothesis, $\alpha \vdash P \parr U\left\{{}^{b}/_{a}\right\} \rhd Q$ and $\alpha_0, \alpha_1, b : \tau_0 \vdash P \parr U\left\{{}^{b}/_{a}\right\} : \tau$ yield the following, as required.

$$\alpha_0, \alpha_1, b : \tau_0 \vdash Q : \tau$$

Consider the blank node rule and assume that the following type tree holds.

$$\frac{\alpha_0 \vdash \bigwedge a : \tau_0 . P : \tau \quad \alpha_1 \vdash Q : \tau}{\alpha_0, \alpha_1 \vdash \bigwedge a : \tau_0 . P \parr Q : \tau}$$

Hence, assuming that $a \notin \mathrm{fn}(Q)$, the following type tree holds.

$$\frac{\alpha_0 + a\colon \tau_0 \vdash P' : \tau \quad \alpha_1 \vdash Q\colon \tau}{\alpha_0, \alpha_1 + a\colon \tau_0 \vdash P \parr Q\colon \tau}$$

Hence by induction, there exists α_0', α_1' such that $\alpha_0, \alpha_1 \equiv \alpha_0', \alpha_1'$ and the following type tree holds.

$$\frac{\alpha_0' + a\colon \tau_0 \vdash P' : \tau \quad \alpha_1' \vdash Q' : \tau}{\alpha_0', \alpha_1' + a\colon \tau_0 \vdash P' \parr Q' : \tau}$$

Therefore, assuming that $a \notin \mathrm{fn}(Q')$ the following proof tree holds, as required.

$$\frac{\dfrac{\alpha_0' + a\colon \tau_0 \vdash P' : \tau}{\alpha_0' \vdash \bigwedge a\colon \tau_0.P' : \tau} \quad \alpha_1' \vdash Q' : \tau}{\alpha_0', \alpha_1' \vdash P' \parr Q' : \tau}$$

The remaining cases follow a similar pattern. Therefore, by induction on the structure of a commitment derivation, types are preserved by atomic commitments. □

5.5.3.1 Monotonicity of contexts.

The examples in the previous section indicate the weakest context for a transition. However, the example involving the blank node quantifier uses subtyping in the select name rule to select a stronger name. Similarly, in all examples subtyping allows a stronger type to be used in the environment.

For instance, a context which requires that *Burns* is of type PERSON, is satisfied by a context which instead assigns the subtype WRITER to the same name. In general, Proposition 5.8 verifies that a stronger context can be used in place of a weaker context without breaking a commitment. The preorder extends subtyping point-wise to environments.

Proposition 5.8 (Monotonicity)**.** *If $\alpha_0 \leq \alpha_1$, then $\alpha_1 \vdash P \rhd Q$ yields that $\alpha_0 \vdash P \rhd Q$.*

The proof pushes the strengthening of the context towards the select quantifiers, where it is eliminated. A similar proof shows the monotonicity of typing. Monotonicity facilitates integration by allowing processes to be moved to a stronger environment without further type checks.

5.5.3.2 Recovering the untyped calculus.

The relationship between the typed and untyped calculus is acknowledged through erasure. Erasure removes all type annotations whilst retaining operational behaviour. In particular, the type annotations which appear in select and blank node quantifiers are removed, as defined by the

transformation `erase`. As verified in Proposition 5.9, all transitions possible in the typed calculus are possible in the untyped calculus. For an exact match, the tensor rule is relaxed to remove requirement that the combined contexts are disjoint.

Proposition 5.9. *If* $\alpha \vdash P \rhd Q$, *then* $\texttt{erase}\, P \rhd \texttt{erase}\, Q$.

However, as expected, the converse does not hold. There exist transitions in the untyped calculus that are impossible in the typed calculus. For instance in the following example the blank node can only be selected after erasure, since $\top$ is not a subtype of DOCUMENT.

$$\texttt{erase}\left(\begin{array}{l} \bigwedge a\colon \top.(a\ \mathit{status}\ \mathit{official})\,, \\ \bigvee b\colon \textsc{document}.(b\ \mathit{status}\ \mathit{official})^{\perp} \end{array}\right) \rhd \perp$$

The main differences between the typed commitments in Fig 5.11, and the untyped commitments in Fig. 3.4, are summarised as follows. The select name quantifier inputs a name of a given type, rather than any name. The select literal rule checks the datatype of the literal, rather than accepting any literal. Rules propagate resulting constraints on the context, whereas the untyped calculus does not constrain the context. The blank node quantifier rule simulates URIs of a given type, rather than any URI.

The combination of the subtype system, literal only typing, monotonicity and erasure allow different strengths of type system to be used in different applications.

5.6 Type Inference Algorithms

Type inference reduces constraints imposed by the type system by inferring types from partial type information. An update can be provided untyped by a programmer. An algorithm then automatically infers the missing type annotations. Type inference makes programming easier and improves interoperability with Linked Data systems with different degrees of static type information.

For instance, in the example below the types of *Hamish* and *Dmitri* are unknown, so are assigned fresh type variables X and Y. The constraints $\textsc{x} \leq \textsc{person}$ and $\textsc{y} \leq \textsc{person}$ are obtained by unfolding the tree of the following algorithmic type judgement.

$$\begin{array}{l} \mathit{knows} : \mathrm{p}(\textsc{person}, \textsc{person})\,, \\ \mathit{Hamish} : \textsc{x}, \mathit{Dmitri} : \textsc{y} \end{array} \Vdash (\mathit{Hamish}\ \mathit{knows}\ \mathit{Dmitri}) : \top$$

A type inference algorithm discovers the minimal unifier. The minimal unifier above is $\textsc{x} \mapsto \textsc{person}, \textsc{y} \mapsto \textsc{person}$. This unifier is a substitution, which gives an valid type judgement when applied to the above tree.

The general inference algorithm proceeds as follows. Firstly, apply algorithmic subtyping to obtain a proof tree indicating a set of subtype constraints over types involving fresh atomic types, as in the example above. Secondly, apply unification to the constraints, until either the constraints are rejected or the algorithm terminates successfully. If the constraints are rejected, there is no unifier. If the constraints are accepted, then the minimal unifier is generated [93].

Type inference appears implicitly in the RDFS standard. The rules of RDFS state that given a predicate, the type of the domain of the predicate bounds the type of any subject of the predicate. Similarly, the range of a predicate bounds the type of any object [33]. As demonstrated above, the same effect is achieved by type inference in this work. Type inference is performed at compile time, so incurs no cost to queries or updates.

5.7 Conclusions on the Type System

The calculus is extended with a light type system, which encompasses several use cases in the application domain. The light approach to typing, inspired by RDFS, contrasts to much stronger schema typically used in databases and XML, which require a global perspective on data. This light approach is suited to Linked Data, where data is drawn from many sources. It is expensive to coordinate strong schema in a distributed setting. The type system can be checked locally at the level of individual triples and updates. Local checks are appropriate for a highly distributed system.

The most basic type system checks only literals, which are conventional data so well understood. The more expressive type system also types URIs. A sub-type system is required over these types for flexibility. A basic cut-elimination result is proven for the sub-type system in Theorem 5.1. This cut elimination result is unrelated to cut elimination for the full calculus outlined in previous chapters.

The syndication calculus introduced in previous sections is extended with type annotations. Both the syntax is typed and the operational semantics are typed. The typed syntax, introduced in Section 5.4, establishes the meaning of a well typed update. The typed operational semantics, defined in Section 5.5, carries type information which cannot be guaranteed statically.

The dynamic type checks triggered by the operational semantics occur when a name or literal is discovered in an update. Thus there is an interplay between static and dynamic typing. The main result of this chapter, Theorem 5.7, proves that a statically typed process remains well typed with respect to the operational semantics. This type preservation result is essential for the type system to be recommended.

Another result established in this chapter is that typing is algorithmic. There exists an alternative type system which is completely syntax directed. This alternative type system is sound and complete with respect to the main type system as proven in Theorem 5.3 and Theorem 5.4. This is particularly useful for type inference algorithms which are discusses briefly in Section 5.6.

Chapter 6

Conclusions

The findings of this work can be considered from several perspectives. The main contribution is a new model for Linked Data programming languages. This model can be used to evaluate design decisions in standards, build useful tools, and expose new problems in the foundations of computer science.

Firstly, the model is used to evaluate W3C standards. This evaluation of the standards indicates both strong features and issues. The model is the first to investigate how key standards work together to fulfil their intended purpose.

Secondly, the model is considered as a foundation for prospective tools. Such tools could not be developed confidently without such a suitable foundation, as provided by the model. The main theorems of this work provide the confidence required to proceed with the development of tools in the future.

Thirdly, the model is considered as a contribution to the foundations of computer science itself. The model borrows features from many existing models, but this particular combination of features is new. This case study investigates the demands of a model which approaches the requirements of a real modern application. The application may not initially appear to be vastly complex; yet it is beyond scope of existing models. Thus, there remain technical modelling problems beyond the human communication problems identified in the introduction.

6.1 Evaluation of the Model as Justification for Standards

An aim of this work is to provide a model where properties of key technologies are derived rather than assumed. A model can never claim to be a priori, since is open to be verified by another deeper model. However, an external model which uses conventional techniques can both add further weight to some design decisions and expose weaknesses in other design decisions.

The two technologies which have been tackled in this way are the SPARQL and RDFS standards. For SPARQL, an algebra is verified by defining an operational model then using bisimulation to prove that the algebra holds in the model. The approach to SPARQL is successful — the operational model verifies the expected algebra. For RDFS, the entailment rules are derived using a type system, where the rules of RDFS follow from type inference. The model for RDFS questions the W3C standards — serious mismatches between RDFS types and types in conventional type systems are exposed.

The operational semantics successfully model both SPARQL Query and SPARQL Update. The model verifies that each of the core features of the query and update languages corresponds to concepts found in related calculi and logics. For instance, the `UNION` keyword in SPARQL Query corresponds to internal choice in process calculi and additive disjunction in Linear Logic.

Relevant examples in the SPARQL standard are correctly captured by the model. Also few features of SPARQL Query are redundant — one exception being the outer join. Thus SPARQL Query is sufficiently well designed to be specified using a clean deductive system. A single axiom is sufficient to extend the operational semantics of SPARQL Query to model SPARQL Update.

Further to specifying the operational behaviour of SPARQL Query and SPARQL Update an algebra is derived. The features of this algebra corresponds to well known algebras, widely applied in computer science. These include semirings, Kleene algebras, Boolean algebras and quantifiers as limits and colimits. This is a powerful combination of features which would not normally be considered outside a real demanding application.

The algebra is not complete, hence there remain algebraic features to be discovered. A question exposed is how SPARQL is related to other models with similar algebraic properties such as topological vector spaces. A suitable topological vector space could serve as a denotational model of SPARQL which would add further justification to SPARQL.

The models for RDFS provide a much weaker validation of the standard. The standard provides several features which mean that a clear model of the specification may be unattainable. Related work attempts to build a Tarski-style model theory, but finds that modifications need to be made. This work presents a type system, but still finds that similar modifications need to be made.

A type system for RDFS appears to be manageable only when top level classes are used as types. However, when only top level classes are used the inference rules for RDF types are lost. This cut down type system would merely distinguish between predicates, classes, datatype predicates and everything else.

By permitting more classes as types, the inference rules of RDFS can be derived using type inference in the type system. Type inference need only be applied once at compile time for any process, and is syntax directed, so is very efficient. However, the question that remains is whether significant applications would actually benefit from classes as static type information.

The RDFS standard however has an important rôle in data integration, so should not be ignored. If only the fundamental features of RDFS are used then a simple clear model is obtained. This model has nothing to do with type systems, and works directly over URIs which appear in terms.

The minimal model of RDFS simply allows the subClassOf and subPropertyOf relations to be used as preorders over URIs. This preorder embeds clearly in the operational semantics of queries and updates. Thus a clear correspondence between these standards is specified by the model. Such a clear correspondence between standards is entirely missing from the existing W3C specifications. Furthermore, the preorder embeds cleanly in the algebra where it corresponds to the refinement relation.

6.2 Useful Tools Enabled by the Model

Tools have not yet been produced, but tools can now be developed using this work. The main results of this work provide a solid foundation for several important tools.

An operational semantics of SPARQL Update can be used as a point of reference for compiler engineers who implement the language. However, an operational semantics is perhaps most useful when used to produce tools which assist a compiler engineer. The tools suggested include a model checker, an optimisation tool, a type checker and a type inference mechanism. Here the suggested tools are described.

Ideally a compiler engineer should be able to prove that an implementation of a language is a refinement of the operational semantics of the language. However, there are generally many technical concerns in the development process of a language, which would be expensive to formally verify. It is therefore more realistic to use the rules of the operational semantics as the basis of a model checker.

A model checker would take an atomic commitment performed by an implementation and apply the rules of the operational semantics to verify whether that atomic commitment was legal. If an implementation performs an atomic action which cannot be verified by the operational semantics, then a flaw in the implementation is discovered.

A model checker cannot prove that every possible update matches the operational semantics, since the search space is infinite. However many cases can be systematically checked, increasing confidence that an implementation satisfies the operational semantics.

A tool which assists the verification of an implementation with respect to the operational semantics can be useful. Genuine implementation flaws can be discovered. Furthermore, different implementations which satisfy the same operational semantics can be expected to have similar behaviour. A consistent expected behaviour across implementations is important when multiple implementations are used in a distributed environment, as expected on the Web. However, the

behaviour of correct implementations will not be expected to have identical behaviour. This is due to the existence of many different strategies for resolving non-determinism.

Significantly, if an implementation satisfies the operational semantics then other tools which rely on a correct operational semantics can be used. For instance, the proof of correctness of the algebra, in Chapter 4, uses an operational semantics. This means that the algebra can only be used confidently in an implementation which satisfies the operational semantics. Similarly, the type preservation proof, in Chapter 5, makes use of an operational semantics. Thus type preservation can only be relied upon in an implementation which satisfies the operational semantics.

An optimisation tool is suggested by the algebra in Chapter 4. The algebra allows queries, updates and processes to be rewritten without changing the observable operational behaviour. Two queries which have the same observable operational behaviour differ in efficiency, for instance one query may pose redundant demands.

An optimisation tool would allow queries and updates to be rewritten to a normal form. The normal form would be chosen to be the most efficient form for a given implementation. An optimisation tool can be used on the client side to directly optimise a high level language for Linked Data. However, it is perhaps most useful on the server side for optimising queries and updates received.

Two tools are suggested by the type system in Chapter 5. The first is a type checker. Given a type environment and a process, a type checker can verify whether the process is well typed in the environment. A type type checker can be developed according to the rules of the algorithmic type system. Type checking is useful for picking up small obvious errors.

A type inference tool tends to be more useful than a type checker, since it requires little input from a programmer. A type inference tool takes some program and perhaps some partial type information and infers the remaining type information if possible. A type inference tool would be developed based on the algorithmic type system. Types need only be checked once, due to the type preservation property. Types are also preserved if a process is moved to a stronger type environment or subtype system, by monotonicity of the type environment.

Thus three tools can be developed. A model checker assists the compiler engineer, an optimiser improves implementations and a type inference mechanism assist the programmer. The type tools are easy to develop, since algorithmic typing is syntax directed. The other two tools require search strategies to implement. The complexity of the tools are at least PSPACE-hard, since the equations of Kleene-algebras are PSPACE-hard and the algebra contains a Kleene-algebra [84]. An upper complexity bound is an open question.

A key question is whether the calculus can be efficiently implemented systematically using the operational semantics. The calculus is an operational model so it is clear that it can be naïvely implemented. Similar concurrent languages have been implemented using a virtual machine which keeps track of the available actions and searches to find actions which successfully interact, then triggering their continuations, as in the implementation of the Pict language [111].

Other implementation of process calculi include the Jocaml language. Jocaml modifies the existing concurrency model of the general purpose high-level language Ocaml to fit the model of the Join calculus [39]. Either approach, either defining a new virtual machine or modifying an existing language, could be used to implement the calculus.

However the focus of this calculus is on reading and writing data. Thus an implementation is more likely to take the form similar to a database. Indeed, asking whether models of concurrency can form a principled basis to the design of servers for the Web of Data, may be a fundamentally flawed question. Perhaps, instead the question could be whether databases can be used to implement concurrency models. Should a concurrent programming language be implemented using a preprocessor which compiles the language to an existing database language? This work has been conducted without familiarity with any database research, so only an opinion can be presented. The opinion is that databases and concurrency face the same fundamental issues when a principled approach is embarked upon [64, 47]. What are the algebraic structures, what are the representations of those algebraic structures and the how can those representations be deployed?

6.3 Evaluation of the Model as a Process Calculus

The calculus developed in this work is new. It borrows from established calculi, such as the π-calculus, but includes features which cannot be expressed primitively in existing calculi. The calculus is more expressive than many existing calculi, due to its primitives for synchronous atomic actions. Typically process calculi have only one or two possible atomic actions; whereas this calculus has an many possible atomic actions. The spectrum of atomic actions correspond to the full range of possible queries and updates. Thus the calculus is an interesting contribution as a process calculus in its own right.

A key difference between the syndication calculus and the π-calculus is that atoms are triples instead of channel value pairs. It is proven in Chapter 4, that the channel based π-calculus can be embedded in the syndication calculus extended with pairs of names. In doing so, the interaction rule of the π-calculus is broken down into more primitive operations. Thus the syndication calculus is strictly more expressive than the π-calculus.

A key feature introduced in the calculus is the tensor product. This construct is missing in most existing process calculi, with exceptions including SCCS [97]. The tensor product allows queries and updates which deal with more than one triple. It therefore plays the rôle of the join of queries as used in relational algebra for databases. It provides more control than a traditional join operation, since the resources used for each branch are accounted for. By accounting for resources each branch of the tensor can be evaluated separately, thus in parallel.

The tensor product behaves well with other constructs of the calculus. Along with choice, true and false it forms a semiring. Semirings are natural structures which arise in abstract algebra.

Semirings are generalisations of rings which arise by taking the ideals of rings. Thus the explicit use of tensor takes steps towards relating process algebra to more conventional algebra. An idempotent semiring defines a monoid in a suplattice. Other features including inputs and iteration, are formulated naturally with respect this monoid in a suplattice.

The calculus can be divided into two perspectives, a spatial perspective and a temporal perspective. The tensor construct is spatial as it uses processes in different locations but in the same atomic temporal step. In contrast the 'then' construct for continuations is temporal. The continuation can only proceed once the guard has been triggered. The par construct is both spatial and temporal. Parallel processes may be either used spatially separately in the same temporal step, or temporly separately. Some related models have investigated combinations of spatial and temporal operators [61, 71].

Space is commutative, since at this level of abstraction it only matters that resources are separate not where they are located. However, time is non-commutative. Reordering actions in time can change the meaning of a process. These themes are central to the current hot topic of modern quantum logic, which features commutative operators for space and non-commutative operators for time [40, 91, 134]. Physicists have been studying processes with respect to space and time for hundreds of years, so their models are beyond the syntactic approach of process calculi used in this work. For instance, a model with a metric would provide further principled optimisation opportunities, by measuring the distance between two processes. The main challenging question exposed by this work — to establish a complete algebra for equivalence and refinement in the calculus — may by tackled using models adapted from physics [4].

6.4 Final Remarks

The thesis has a strong conclusion and a subjective conclusion. The strong conclusion is that the query and update languages for the Web of Data socialise well with process calculi. The resulting model provides a foundation for concurrent high-level languages for the Web of Data. The foundation gives rise to a rich and useful algebra. The subjective conclusion is that types for URIs, as introduced by Web standards, have several legitimate interpretations using conventional type systems. Programming languages for the Web of Data can rely on typing conventional literals; but the specific application must be carefully considered if URIs are to be typed.

Bibliography

[1] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *Proceedings of the 28th ACM SIGPLAN-SIGACT*, volume 36, pages 104–115, London, January 17–19 2001. ACM, NY.

[2] Serge Abiteboul. Querying semi-structured data. In Foto Afrati and Phokion Kolaitis, editors, *Database Theory – International Conference on Database Theory*, pages 1–18, Delphi, Greece, January 8-10 1997. Springer, Berlin/Heidelberg.

[3] Samson Abramsky. Computational interpretations of Linear Logic. *Theoretical Computer Science*, 111:3–57, 1993.

[4] Samson Abramsky. What are the fundamental structures of concurrency?: We still don't know! In *Proceedings of the Workshop Essays on Algebraic Process Calculi*, volume 162, pages 37–41. Elsevier, September 29 2006.

[5] Samson Abramsky, Simon Gay, and Rajagopal Nagarajan. Interaction categories and the foundations of typed concurrent programming. In Manfred Broy, editor, *Proceedings of the NATO Advanced Study Institute on Deductive Program Design, Marktoberdorf, Germany*, pages 35–114. Springer, 1995.

[6] Samson Abramsky and Steven Vickers. Quantales, observational logic and process semantics. *Mathematical Structures in Computer Science*, 3(02):161–227, 1993.

[7] Luca Aceto, Bard Bloom, and Frits Vaandrager. Turning SOS rules into equations. *Information and Computation*, 111(1):1 – 52, 1994.

[8] Peter Aczel. *Non-well-founded sets*, volume 14. Center for the Study of Language and Information, Stanford, CA, 1988.

[9] Harith Alani et al. Managing reference: Ensuring referential integrity of ontologies for the Semantic Web. In Asunción Gómez-Pérez and V. Benjamins, editors, *Knowledge Engineering and Knowledge Management: Ontologies and the Semantic Web*, volume 2473, pages 235–246. Springer, Berlin/Heidelberg, 2002.

[10] John Warner Backus. The syntax and semantics of the proposed international algebraic language of the zurich acm-gamm conference. In *Proceedings of the International Conference on Information Processing*, pages 125–132. Oldenbourg, Munich and Butterworth, London, 1959.

[11] David Baelde and Dale Miller. Least and greatest fixed points in linear logic. In Nachum Dershowitz and Andrei Voronkov, editors, *Logic for Programming Artificial Intelligence and Reasoning 2007. Yerevan, Armenia. 15–19 October*, pages 92–106. Springer, 2007.

[12] John C. Baez and Mark Stay. Physics, topology, logic and computation: A rosetta stone. In Bob Coecke, editor, *New Structures for Physics*, volume 813 of *Lecture Notes in Physics*, pages 95–172. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-12821-9_2.

[13] Jean-Franois Baget. RDF entailment as a graph homomorphism. In Yolanda Gil, Enrico Motta, V. Benjamins, and Mark Musen, editors, *The Semantic Web – ISWC 2005*, volume 3729, pages 82–96, Galway, Ireland, November 2005. Springer Berlin/Heidelberg.

[14] Alexandru Baltag, Bob Coecke, and Mehrnoosh Sadrzadeh. Epistemic Actions as Resources. *Journal of Logic and Computation*, 17(3):555–585, 2007.

[15] Michael Barr. *-autonomous categories and linear logic. *Mathematical Structures in Computer Science*, 1(02):159–178, 1991.

[16] David Beckett and Tim Berners-Lee. Turtle – Terse RDF Triple Language. Team submission, W3C, 2008.

[17] Emmanuel Beffara. A concurrent model for linear logic. *Electronic Notes in Theoretical Computer Science*, 155:147–168, May 2006.

[18] Gianluigi Bellin and Philip J. Scott. On the π-calculus and Linear Logic. *Theoretical Computer Science*, 135:11–65, 1994.

[19] David Benson. The shuffle bialgebra. In M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Language Semantics*, volume 298 of *Lecture Notes in Computer Science*, pages 616–637. Springer Berlin / Heidelberg, 1988.

[20] Jan A. Bergstra and Jan Willem Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984.

[21] Tim Berners-Lee. Linked Data. *International Journal on Semantic Web and Information Systems*, 4(2):1, 2006.

[22] Tim Berners-Lee. Read-Write Linked Data. Personal view only. http://www.w3.org/DesignIssues/ReadWriteLinkedData.html, December 2010.

[23] Tim Berners-Lee, Dan Connolly, Lalana Kagal, Yosi Scharf, and Jim Hendler. N3logic: A logical framework for the World Wide Web. *Theory and Practice of Logic Programming*, 8(3):249–269, 2008.

[24] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001.

[25] Paul V. Biron and Ashok Malhotra. XML schema part 2: Datatypes second edition. Recommendation REC-xmlschema-2-20041028, W3C, MIT, Cambridge, MA, 2004.

[26] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Semiring-based constraint satisfaction and optimization. *Journal of the ACM*, 44(2):201–236, 1997.

[27] Christian Bizer. The emerging Web of Linked Data. *IEEE Intelligent Systems*, 24:87–92, 2009.

[28] Christian Bizer et al. DBpedia: A crystallization point for the Web of Data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(3):154–165, 2009.

[29] Richard F. Blute. Hopf algebras and "linear logic". *Mathematical Structures in Computer Science*, 6(02):189–212, 1996.

[30] Richard F. Blute and Philip J. Scott. The shuffle Hopf algebra and noncommutative full completeness. *The Journal of Symbolic Logic*, 63(4):1413–1436, 1998.

[31] Laura Bocchi and Roberto Lucchi. Atomic commit and negotiation in service oriented computing. In Paolo Ciancarini and Herbert Wiklicky, editors, *Coordination Models and Languages*. Springer, Berlin/Heidelberg, 2006.

[32] David Booth and Canyang Kevin. Web services description language (WSDL) version 2.0 part 0: Primer. Recommendation REC-wsdl20-primer-20070626, W3C, MIT, Cambridge, MA, 2007.

[33] Dan Brickley and Ramanathan V. Guha. RDF vocabulary description language 1.0: RDF Schema. Recommendation REC-rdf-schema-20040210, W3C, MIT, Cambridge, MA, 2004.

[34] Luitzen E.J. Brouwer. De onbetrouwbaarheid der logische principes (the untrustworthiness of the principles of logic). *Tijdschrif voor wijsbegeerte*, 2:152–158, 1908.

[35] Allen Brown, Matthew Fuchs, Jonathan Robie, and Philip Wadler. MSL: A model for W3C XML Schema. *Computer Networks*, 39(5):507–521, 2002.

[36] Maria Buscemi and Ugo Montanari. CC-π: A constraint-based language for specifying service level agreements. In Rocco De Nicola, editor, *Programming Languages and Systems*, volume 4421 of *Lecture Notes in Computer Science*, pages 18–32. Springer, Berlin/Heidelberg, 2007.

[37] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for Web Services. In Rocco De Nicola, editor, *Programming Languages and Systems*, volume 4421 of *Lecture Notes in Computer Science*, pages 2–17. Springer, Berlin/Heidelberg, 2007.

[38] Jeremy J. Carroll, Christian Bizer, Pat Hayes, and Patrick Stickler. Named graphs. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(4):247–267, 2005.

[39] Silvain Conchon and Fabrice Le Fessant. Jocaml: Mobile agents for objective-caml. In *Proceedings of the First International Symposium on Agent Systems and Applications Third International Symposium on Mobile Agents*, ASAMA '99, pages 22–29, Washington, DC, 1999. IEEE Computer Society.

[40] Alain Connes. Non-commutative differential geometry. *Publications Mathmatiques de L'IHS*, 62:41–144, 1985.

[41] John Horton Conway. *Regular Algebra and Finite Machines*. Chapman and Hall, London, 1971.

[42] Thierry Coquand and Gerard Huet. The calculus of constructions. *Information and Compututation*, 76:95–120, February 1988.

[43] Richard Cyganiak. A relational algebra for SPARQL. External HPL-2005-170, Hewlett–Packard Laboritories, Bristol, 2005.

[44] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, Albuquerque, New Mexico, January 1982. ACM, New York, NY.

[45] Li Ding, Joshua Shinavier, Tim Finin, and Deborah L. McGuinness. owl:sameAs and Linked Data: An empirical study. In *Proceedings of the WebSci10: Extending the Frontiers of Society On-Line, April 26-27th, 2010, Raleigh, NC.*, 2010.

[46] Ronald Fagin, Phokion Kolaitis, Rene Miller, and Lucian Popa. Data exchange: Semantics and query answering. In Diego Calvanese, Maurizio Lenzerini, and Rajeev Motwani, editors, *Database Theory – International Conference on Database Theory, January 8-10, Siena, Italy*, volume 2572, pages 207–224, Berlin/Heidelberg, 2003. Springer.

[47] Lisbeth Fajstrup, Martin Rauen, and Eric Goubault. Algebraic topology and concurrency. *Theoretical Computer Science*, 357(1-3):241 – 278, 2006.

[48] Roy T. Fielding and Richard N. Taylor. Principled design of the modern Web architecture. *ACM Transactions on Internet Technology*, 2(2):115–150, 2002.

[49] Doug Foxvog. Instances of instances modeled via higher-order classes. In *Foundational Aspects of Ontologies, 28th German Conference on Artificial Intelligence, Koblenz, Germany, September*, volume 1860-4471, pages 46–54, 2005.

[50] Jean Gallier. Constructive logics. Part II: Linear Logic and Proof Nets. Research Report PR2-RR-9, Digital Equipment Corporation, Paris, 1991.

[51] Jean H. Gallier. Constructive logics part I: A tutorial on proof systems and typed λ-calculi. *Theoretical Computer Science*, 110(2):249–339, 1993.

[52] Jesse James Garrett. Ajax: A new approach to Web applications. Published on the Web, February 2005.

[53] Paul Gearon, Alexandre Passant, and Axel Polleres. SPARQL 1.1 Update. Working draft WD-sparql11-update-20110512, W3C, May 2011.

[54] Jean-Yves Girard. Une extension de l'interprétation de gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In *In Proceedings of the 2nd Scandinavian Logic Symposium*. North-Holland, 1970.

[55] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–112, 1987.

[56] Jean-Yves Girard. Truth, modality and intersubjectivity. *Mathematical Structures in Computer Science*, 17:1153–1167, December 2007.

[57] Hugh Glaser, Afraz Jaffri, and Ian Millard. Managing co-reference on the Semantic Web. In *Linked Data on the Web workshop*, Madrid, Spain, April 20 2009.

[58] R. Goré, C. Kupke, and D. Pattinson. Optimal tableau algorithms for coalgebraic logics. In R. Majumdar and J. Esparza, editors, *Proc. TACAS 2010*, Lecture Notes in Computer Science, 2010.

[59] J. Gregorio and B. de hOra. The Atom Publishing Protocol. Proposed Standard rfc5023, Internet Engineering Task Force, Fremont, CA, October 2007.

[60] Daniel Gruhl, Ramanathan V. Guha, David Liben-Nowell, and Andrew Tomkins. Information diffusion through blogspace. In *Proceedings of the 13th international conference on World Wide Web*, WWW '04, pages 491–501, NY, 2004. ACM.

[61] Alessio Guglielmi. A system of interaction and structure. *ACM Transactions on Compututational Logic*, 8, January 2007.

[62] Ramanathan V. Guha. *Contexts: a formalization and some applications*. PhD thesis, Stanford Computer Science Department, Stanford, CA, 1992. STAN-CS-91-1399.

[63] Ramanathan V. Guha, Rob McCool, and Richard Fikes. Contexts for the Semantic Web. In Sheila A. McIlraith, Dimitris Plexousakis, and Frank van Harmelen, editors, *The Semantic Web ISWC 2004*, volume 3298 of *Lecture Notes in Computer Science*, pages 32–46. Springer Berlin / Heidelberg, 2004.

[64] Jeremy Gunawardena. Homotopy and concurrency. In B. Păun, editor, *Current trends in theoretical computer science*, pages 447–459. World Scientific Publishing Co., Inc., River Edge, NJ, 2001.

[65] Harry Halpin and Pat Hayes. When owl:sameAs isn't the same: An analysis of identity links on the Semantic Web. In Christian Bizer, Tom Heath, Tim Berners-Lee, and Michael Hausenblas, editors, *Proceedings of the WWW2010 Workshop on Linked Data on the Web Raleigh, USA, April 27, 2010.*, 2010.

[66] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. SPARQL 1.1 query language. Working Draft WD-sparql11-query-20101014, W3C, MIT, Cambridge, MA, October 2010.

[67] Olaf Hartig et al. Executing SPARQL Queries over the Web of Linked Data. In A. Bernstain et al., editors, *The Semantic Web – ISWC 2009, Chantilly, VA*, volume 5823, pages 293–309. Springer, 2009.

[68] J. Rodger Hindley. The principle type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[69] Jaakko Hintikka. Language-games. In Esa Saarinen, editor, *Game-Theoretical Semantics*, volume 5 of *Studies in Linguistics and Philosophy*, pages 1–26. Springer, 1979.

[70] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, 1978.

[71] C. A. R. Hoare, Bernhard Möller, Georg Struth, and Ian Wehrman. Concurrent Kleene algebra. In Mario Bravetti and Gianluigi Zavattaro, editors, *CONCUR 2009, Bologna, Italy*, volume 5710, pages 399–414. Springer, 2009.

[72] Joshua S. Hodas and Dale Miller. Logic programming in a fragment of Intuitionistic Linear Logic. *Information and Computation*, 110(2):327–365, 1994.

[73] Kohei Honda and Nobuko Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 151(2):437 – 486, 1995.

[74] Ian Horrocks and Peter Patel-Schneider. Reducing OWL entailment to description logic satisfiability. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(4):345–357, 2004.

[75] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transations on Programming Language Systems*, 23:396–450, May 2001.

[76] Alan Jeffrey and Julian Rathke. Contextual equivalence for higher-order π-calculus revisited. *Logical Methods in Computer Science*, 1(4):1–22, 2005.

[77] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.

[78] Graham Klyne and Jeremy Carroll. Resource Description Framework: Concepts and abstract syntax. Recommendation REC-rdf-concepts-20040210, W3C, MIT, Cambridge, MA, 2004.

[79] Naoki Kobayashi and Akinori Yonezawa. ACL – a concurrent Linear Logic programming paradigm. In *Proceedings of the 1993 International Logic Programming Symposium*, pages 279–294. MIT Press, 1993.

[80] Georgi Kobilarov et al. Media meets Semantic Web: How the BBC uses DBpedia and Linked Data to make connections. In Lora Aroyo, editor, *The Semantic Web: Research and Applications. 6th European Semantic Web Conference*, pages 723–737, Heraklion, Greece, May 31 – June 4 2009. Springer, Berlin/Heidelberg.

[81] Jacek Kopecký. WSDL RDF mapping: Developing ontologies from standardized XML languages. In John Roddick et al., editors, *Advances in Conceptual Modeling – Theory and Practice*, volume 4231 of *Lecture Notes in Computer Science*, pages 312–322. Springer Berlin / Heidelberg, 2006.

[82] Dexter Kozen. On Kleene algebras and closed semirings. In Rovan, editor, *Proceedings on Mathematical Foundations of Computer Science*, volume 452, pages 26–47. Springer-Verlag, 1990.

[83] Dexter Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 110:366–390, 1994.

[84] Dexter Kozen. Kleene algebra with tests. *ACM Transactions on Programing Languages and Systems*, 19:427–443, May 1997.

[85] Joachim Lambek. On the calculus of syntactic types. In R. Jacobson, editor, *Structure of Language and its Mathematical Aspects*, Providence, 1961. American Mathematical Society.

[86] Peter John Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[87] Ora Lassila and Ralph R. Swick. The Resource Description Framework (RDF) model and syntax specification. Recommendation REC-rdf-syntax-19990222, W3C, MIT, Cambridge, MA, 1999.

[88] F. William Lawvere. Adjointness in foundations. *Dialectica*, 23(3-4):281–296, 1969.

[89] Holbrook Mann MacNeille. Extensions of partially ordered sets. *Proceedings of the National Academy of Sciences of the United States of America*, 22(1):45–50, 1936.

[90] Sergio Maffeis and Philippa Gardner. Behavioural equivalences for dynamic Web data. *Journal of Logic and Algebraic Programming: Algebraic Process Calculi. The First Twenty Five Years and Beyond. III*, 75(1):86–138, 2008.

[91] Shahn Majid. Physics for algebraists: Non-commutative and non-cocommutative Hopf algebras by a bicrossproduct construction. *Journal of Algebra*, 130(1):17–64, 1990.

[92] Ashok Malhotra, David Peterson, Shudi Gao, C. M. Sperberg-McQueen, and Henry S. Thompson. XML Schema Definition Language 1.1 part 2: Datatypes. Working Draft WD-xmlschema11-2-20091203, W3C, MIT, Cambridge, MA, 2009.

[93] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[94] Per Martin-Löf. An intuitionistic theory of types. In *Notes of Giovanni Sambin on a series of lectures given in Padova, Univ. of Padova, Italy*. Bibliopolis, Napoli, June 1984.

[95] Massimo Merro and Matthew Hennessy. Bisimulation congruences in safe ambients. In *Principles of programming languages*, pages 71–80. ACM, 2002.

[96] Robin Milner. *A calculus of communicating systems*, volume 92. Springer, NJ, 1980.

[97] Robin Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3):267 – 310, 1983.

[98] Robin Milner. The polyadic π-calculus: A tutorial. In F. L. Bauer, W. Brauer, and H. Schwichttenburg, editors, *Logic and Algebra in Specification*. Springer, New York, 1993.

[99] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I and II. *Information and Computation*, 100(1):1–40, 1992.

[100] Robin Milner and Davide Sangiorgi. Barbed bisimulation. In W. Kuich, editor, *Automata, Languages and Programming*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer Berlin / Heidelberg, 1992.

[101] John C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(03):245–285, 1991.

[102] Ugo Montanari and Marco Pistore. Structured coalgebras and minimal HD-automata for the π-calculus. *Theoretical Computer Science*, 340(3):539–576, 2005. Mathematical Foundations of Computer Science 2000.

[103] Christopher J. Mulvey. &. *Rendiconti del Circolo Matematico di Palermo*, 12:99–104, 1986.

[104] Mark Nottingham and Robert Sayre. The Atom Syndication Format. Proposed Standard rfc4287, Internet Engineering Task Force, Fremont, CA, December 2005.

[105] Chimezie Ogbuji. SPARQL 1.1 uniform HTTP protocol for managing RDF graphs. Working draft WD-sparql11-http-rdf-update-20101014, W3C, MIT, Cambridge, MA, October 2010.

[106] Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[107] Eyal Oren, Spyros Kotoulas, George Anadiotis, Ronny Siebes, Annette ten Teije, and Frank van Harmelen. Marvin: Distributed reasoning over large-scale semantic web data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(4):305 – 316, 2009. Semantic Web challenge 2008.

[108] Jeff Z. Pan and Ian Horrocks. Metamodeling architecture of Web Ontology Languages. In *In Proceedings of the Semantic Web Working Symposium*, pages 131–149, 2001.

[109] Dirk Pattinson. Coalgebraic modal logic: soundness, completeness and decidability of local consequence. *Theoretical Computer Science*, 309(1-3):177–193, 2003.

[110] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems*, 34(3):1–45, 2009.

[111] Benjamin C. Pierce and David N. Turner. *Pict: a programming language based on the π-Calculus*, pages 455–494. MIT Press, Cambridge, MA, 2000.

[112] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN19, Computer Science Department, Aarhus University, 1981.

[113] Henri Poincaré. The future of mathematics. *Revue generale des Sciences pures et appliquees*, 19(23), 1908.

[114] Henri Poincaré. *Science et Méthode*. T. Nelson and Sons, London and New York, 1914.

[115] Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF. Recommendation REC-rdf-sparql-query-20080115, W3C, MIT, Cambridge, MA, 2008.

[116] Alan L. Rector and Ian R. Horrocks. Experience building a large, re-usable medical ontology using a description logic with transitivity and concept inclusions. In *In Proceedings of the Workshop on Ontological Engineering, AAAI Spring Symposium*. AAAI Press, 1997.

[117] V. N. Redko. On defining relations for the algebra of regular events. *Ukrainskii Matematicheskii Zhurnal*, pages 120–126, 1964.

[118] John Reynolds. Toward a grainless semantics for shared-variable concurrency. In Kamal Lodaya and Meena Mahajan, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 3328, pages 11–38. Springer, Berlin/Heidelberg, 2005.

[119] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

[120] Jan J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249(1):3 – 80, 2000.

[121] Jan J. M. M. Rutten. Behavioural differential equations: a coinductive calculus of streams, automata, and power series. *Theoretical Computer Science*, 308:1 53, 2003.

[122] Davide Sangiorgi and Robin Milner. The problem of 'weak bisimulation up to'. In W.R. Cleaveland, editor, *CONCUR '92*, volume 630 of *Lecture Notes in Computer Science*, pages 32–46. Springer Berlin / Heidelberg, 1992. 10.1007/BFb0084781.

[123] Davide Sangiorgi and David Walker. *π-calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, 2001.

[124] Vijay A. Saraswat, Martin Rinard, and Prakash Panangaden. The semantic foundations of concurrent constraint programming. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–352, Orlando, Florida, January 21-23 1991. ACM, NY.

[125] Robert Sayre. Atom: the standard in syndication. *IEEE Internet computing*, 9(4):71–78, 2005.

[126] Andy Seaborne and Geetha Manjunath. SPARQL/Update: A language for updating RDF graphs. External HPL-2007-102, Hewlett–Packard Laboritories, Bristol, 2007.

[127] Claudio Gutierrez Sergio Muñoz, Jorge Pérez. Simple and efficient minimal RDFS. *Journal of Web Semantics*, 7(3):220–234, 2009.

[128] Nigel Shadbolt, Wendy Hall, and Tim Berners-Lee. The Semantic Web revisited. *IEEE intelligent systems*, 21(3):96–101, 2006.

[129] Pawel Sobocinski. A non-interleaving process calculus for multi-party synchronisation. In Filippo Bonchi, Davide Grohmann, Paola Spoletini, and Emilio Tuosto, editors, *Proceedings 2nd Interaction and Concurrency Experience: Structured Interactions*, pages 87–98, Bologna, Italy, August 31 2009. EPTCS.

[130] Marshal Stone. The theory of representations of Boolean algebras. *Transactions of the American Mathematical Society*, 40(1), 1936.

[131] Lutz Straßburger. *Linear Logic and Noncommutativity in the Calculus of Structures*. PhD thesis, Faculty of Informatics, Technical University of Dresden, 2003.

[132] Dan Suciu. Distributed query evaluation on semistructured data. *ACM Transactions on Database Systems*, 27(1):1–62, 2002.

[133] Ludwig Wittgenstein. *Philosophical Investigations*. Wiley-Blackwell, Oxford, 1973. First edition 1953.

[134] David N. Yetter. Quantales and noncommutative Linear Logic. *Journal of Symbolic Logic*, 55(1):41–64, 1990.