Context Logic and Tree Update

Uri Zarfaty

Abstract

This thesis introduces Context Logic, a novel spatial logic which was developed to allow local Hoare-style reasoning about tree update, but which also permits reasoning about more general data update. Spatial logics have previously been used to describe properties of tree-like structures (as in Ambient Logic) and to reason locally about dynamic updates of heaps (as in Separation Logic). However, simple adaptations of the Ambient Logic are not expressive enough to capture dynamic updates of trees. Instead, one must reason explicitly about tree contexts in order to capture updates throughout the tree. For example, a typical update removes a portion of data and replaces it by inserting new data in the same place. Context Logic allows us to reason about both the data and the place of insertion.

The thesis describes the general theory of Context Logic, presents a number of extensions and applications, and shows that Context Logic is a generalisation of the Logic of Bunched Implication, the underlying theory of Separation Logic. The thesis then uses Context Logic to reason locally about tree, heap and term update languages, adapting the local reasoning framework of Separation Logic and providing a generalisation of its Frame Rule. Completeness results for these program logics are provided by deriving the weakest preconditions of the update commands from the command axioms. Finally, the thesis introduces an extended imperative update language for manipulating trees with pointers, which incorporates path queries and atomic commands that act at multiple locations. Reasoning about this raises an important point regarding the link between local reasoning and local specifications.

Contents

List of Figures

Acknowledgements

- to my supervisor, Philippa Gardner, for her guidance, encouragement and inspiration;
- to my parents, יוסי and שרה, for their love and support, and my sister יעל, for being the best;
- to Cristiano, for his patience and help;
- to Θεόδωρος, ಕೆವಲ, Алексей, MARCVS and the other Imperialists, for bringing me to life and suffering me gladly;
- and finally, to future readers, who by reading this will have made it worth writing.

Chapter 1

Prologue

This chapter contains a brief outline of the motivation and objectives of the work in this thesis, as well as its key contributions and the publications wherein they appeared. A fuller introductory account is given in the next chapter, which also contains important background information.

1.1 Motivation and Objectives

The original motivation for the work in this thesis came from three overlapping areas of current research: tree update, spatial logics and local reasoning. The online presence of huge amounts of semistructured data, usually represented as trees using XML, has made research on in-place tree update an interesting and active field. In particular, very little work has yet been done on formal reasoning about tree update, making this an obvious challenge. At around the same time, a number of logics collectively known as 'spatial logics' appeared for describing spatial properties of structured data, typically using (de-)composition operators that split terms into parts and reason about them separately. The archetypal examples of these are Ambient Logic, which describes tree-like hierarchies, and the assertion language of Separation Logic, which describes memory heaps. Separation Logic also introduced a third area of research, comprising a Hoare reasoning framework for heap update based on local reasoning principles. These principles assert that program reasoning should be confined to the memory cells that a program actually accesses, which Separation Logic achieves by using its spatial assertion language to split heaps into the appropriate subheaps. Since its introduction, Separation Logic has been successfully applied to

a number of problems, and has helped make widespread program verification a more plausible goal.

Like pieces in a puzzle, these three areas of work combined to form a simple objective, which was to attempt to reason about simple tree update by applying Separation Logic-style local reasoning based on some spatial logic. To do this, it was natural to try to adapt a pre-existing spatial logic for trees such as the Ambient Logic. It turned out, however, that the Ambient Logic was not expressive enough to capture dynamic updates of trees; instead, a real change in approach was required, resulting in the introduction of Context Logic and this work. As the work developed and Context Logic proved to be increasingly successful, the objectives were extended further to exploring the logical foundations of Context Logic, and attempting to use it to reason about other forms of data update, including more realistic tree update languages.

1.2 Contributions

The key contributions of the work in this thesis can be summarised as follows:

- Context Logic — the thesis introduces a novel spatial logic in the style of Separation Logic and the Ambient Logic, based on the simple idea of context application, but general enough to describe a wide range of inductive data structures.

- Local reasoning using Context Logic — the thesis introduces a framework for local Hoare reasoning about data update using Context Logic, extending previous work of Separation Logic and successfully applying it to new domains such as tree update and term rewriting.

- Tree update reasoning — the thesis successfully applies local reasoning to an extended tree update language containing pointers, queries and atomic updates at multiple locations. This serves as a step towards reasoning about updates to semistructured data, and raises an interesting point concerning the link between local reasoning and local specifications.

1.3 Statement of Originality

I declare that this thesis was composed by myself, and that the work that it presents is my own, except where otherwise stated.

1.4 Publications

While some of the work in this thesis is still unpublished, other parts have previously appeared in the following publications:

- C. Calcagno, P. Gardner and U. Zarfaty. *Context Logic and Tree Update*. 33rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 2005 [CGZ05].

  This paper introduced Context Logic and applied it to reasoning about tree update, heap update and term rewriting. A preliminary version of this paper appeared in the Workshop on Logics for Resources, Processes, and Programs (LRPP), 2004 [CGZ04].

- U. Zarfaty and P. Gardner. *Local Reasoning about Tree Update*. 22nd Conference on the Mathematical Foundations of Programming Semantics (MFPS), 2006 [ZG06].

  This paper used Context Logic to reason about a more realistic tree update language including pointers, path queries and atomic updates at multiple locations.

- C. Calcagno, P. Gardner and U. Zarfaty. *Local Reasoning about Data Update*. Gordon Plotkin's Festschrift, 2007 [CGZ07b].

  This paper provided an overview of Context Logic and its application to reasoning about data update.

Furthermore, some of the work in this thesis was also mentioned in a lecture course by Philippa Gardner for the APPSEM summer school [Gar05]. Related work that is not included in the thesis includes a paper on the expressivity of Context Logic published in POPL 2007 [CGZ07a].

Chapter 2

Introduction

This chapter provides a general introduction to the work in the thesis, describing its background and introducing some of its basic ideas. The chapter contains a review of the key preceding work, briefly introduces Context Logic, and summarises the overall structure of the thesis.

As stated in the previous chapter, this thesis presents a novel spatial logic called Context Logic, and uses it reason locally about updates to tree structures. The work can be neatly split into two parts: the first concerns the logical foundation of Context Logic, and builds on related work on spatial logics; the second concerns the update reasoning framework, and follows recent work on tree update and local reasoning. This chapter begins with a background review of these two areas, before giving a brief introduction to Context Logic and a summary of the contents of the thesis.

2.1 Spatial Logics

Spatial logics are logics that contain operators which inspect the spatial structure of the model, as opposed to its temporal behaviour. This is usually achieved using a separation/composition operator that splits terms into parts, and reasons about them separately. While the idea of spatial logics is relatively recent, spatial logics have already been used to describe data structures from a wide variety of areas, including: memory heaps [ORY01, Rey02], stack variables [BCY06], trees [Car01, CGG03b], graphs [CGG02], bigraphs [CMS05b], permissions [BCOP05], concurrent objects [CM98] and distributed systems [CG00a, CC02].

Research on spatial logics has historically consisted of two main independent threads. The first uses spatial logics as part of a Hoare-style reasoning framework for describing and verifying properties of programs that manipulate structured data. The archetypal example of this is Separation Logic by O'Hearn, Reynolds *et al.* [Rey00, IO01, ORY01], which employs a spatial logic based on O'Hearn and Pym's Bunched Logic [OP99, POY04] to reason about heap update. The second thread uses spatial logics for model-checking processes in a way that takes into account the underlying structure of the model. This is typified by the Ambient Logic of Cardelli and Gordon [CG00a, CG07], a logic for describing structural and computational properties of distributed and mobile computation. Since computations evolve over time, Ambient Logic also includes temporal connectives. However, a static subfragment of the logic, comprising a 'pure spatial' logic, has been widely used to model types and query languages for semistructured data [Car01, CG04].

Context Logic combines ideas from both these threads of research, and in particular from their two representative logics, the heap logic of Separation Logic and Ambient Logic. These logics are discussed individually below.

Heap Logic

Separation Logic [ORY01, Rey02] is a framework for reasoning about heap update based on a spatial logic for describing heap resources. This *heap logic* combines the standard connectives of classical logic with spatial assertions describing the shape of heaps, including two key spatial connectives: a separating conjunction $P_1 * P_2$ and a separating implication $P_1 \mathrel{-\!\!*} P_2$.

The idea of the separating conjunction is implicit in early work by Burstall [Bur72], but was first explicitly described by Reynolds in lectures in 1999. An intuitionistic logic based on this idea was presented independently by Reynolds [Rey00] and by Ishtiaq and O'Hearn [IO01], with the latter also introducing the concept of the separating implication. Ishtiaq and O'Hearn also devised a classical version of the logic that is more expressive than the intuitionistic one. The separating conjunction and implication of Separation Logic are both multiplicative operators, in the sense of linear logic [Gir87]: they can be interpreted in terms of resource accumulation and consumption, meaning for example that $P * P \not\equiv P$. The integration of additive and multiplicative connectives in a single categorical structure was studied by O'Hearn

and Pym in the Logic of Bunched Implications [OP99, POY04, BBTS05]. This provides an analysis of the basic notion of resource, and forms the logical basis of the heap logic of Separation Logic.

Separation Logic is used to reason about updates to memory heaps. Heaps are viewed as finite collections of identified locations with values. The locations correspond to dynamically-allocated 'active' memory addresses, which can be acquired and released; the values consist of either static data or pointers, allowing heaps to represent recursive and cyclic data structures. The model permits 'dangling pointers', which point out of the heap, though dereferencing these is always assumed to result in an error. A key observation of Separation Logic is that it is possible to construct heaps by combining two smaller heaps with disjoint address domains. This partial operation on heaps forms the basis of the separating conjunction. Hence, $P_1 * P_2$ is satisfied by a heap h if h can be split into two disjoint subheaps, h_1 and h_2, such that P_1 holds for h_1 and P_2 holds for h_2. This connective is spatial over the heap, in the sense that its semantics depends on the spatial structure of the heap on which it is interpreted. Combining the separating conjunction with basic assertions describing singleton and empty heaps, it is possible to express all the formulæ that describe the content of the heap exactly. For example, the formula $(1 \mapsto 2 * 2 \mapsto 1)$ describes a two-cell heap containing a circular list at locations 1 and 2. In contrast, the formula $(1 \mapsto 2 * 1 \mapsto 2)$ is not satisfied by any heap, since the domains of the heaps described by the two subformulæ are not disjoint. The inclusion of classical logic connectives in the heap logic also makes it possible to express non-exact predicates, such as $(\text{true} * 1 \mapsto 2)$, which describes all heaps that contain a pointer from location 1 to 2, and which can therefore be split into the single-cell heap satisfying $1 \mapsto 2$ and some arbitrary disjoint remainder satisfying true.

In addition to the separating conjunction, the logic also includes a separating implication $P_1 \mathrel{-\!\!*} P_2$ (usually referred to as the 'magic wand'), where $P \mathrel{-\!\!*} [-]$ is the right adjoint of $P * [-]$. Hence $P_1 \mathrel{-\!\!*} P_2$ is satisfied by a heap h if whenever h is composed with a heap h_1 satisfying P_1, the resulting heap satisfies P_2. For example, the formula $(1 \mapsto 2 \mathrel{-\!\!*} P) * (1 \mapsto 1)$ describes a heap that contains a pointer from location 1 to itself but which also satisfies some formula P if that pointer is removed and replaced instead with a pointer from location 1 to location 2. Note that since composition is only possible for disjoint heaps, the separating implication says nothing

about heaps with overlapping domains. For example, the formula $(1 \mapsto 2) \mathrel{-\!\!*} \mathsf{false}$ describes heaps that contain the address 1, as it is not possible to compose with them the single-cell heap satisfying $1 \mapsto 2$.

The standard formulation of Separation Logic also contains quantification over names. For example, the formula $(\exists x.\, 1 \mapsto x)$ describes a single-cell heap at location 1 with an arbitrary value. Similarly, $(\exists x.\, 1 \mapsto x * x \mapsto 2)$ describes a two-cell linked list starting at location 1 and ending with a pointer to location 2. For reasoning about inductive structures such as lists, Separation Logic makes use of inductive heap predicates [Rey02, BCO04b]. For example, the predicate $\mathrm{list}(n, i, j)$ below describes a list of length n starting at location i and ending with a pointer to location j:

$$\mathrm{list}(0, i, j) \triangleq \mathsf{emp} \wedge (i = j)$$
$$\mathrm{list}(n+1, i, j) \triangleq \exists k.\, i \mapsto k * \mathrm{list}(n, k, j)$$

The inductive case describes a pointer from i to some location k, together with a disjoint list starting at k. The base case, meanwhile, states that the heap is empty, and 'ties up' the two variable values.

Much of the utility of the heap logic comes from its ability to easily express separation properties using the separating conjunction and implication, which can then be used in Separation Logic to reason locally about heap update. Counterintuitively, it was shown by Lozes that the separating implication can in fact be eliminated from a quantifier-free version of the heap logic and replaced with a formula that states that a given address is allocated (just like $(1 \mapsto 2) \mathrel{-\!\!*} \mathsf{false}$ did above), without losing any expressivity [Loz04b, Loz05]. Furthermore, Lozes also showed that the separating conjunction can also be removed, and replaced by a size modality. However, while these results are theoretically significant, they are less important in practice: it has since been shown that though the separating connectives can be eliminated, they are still essential for reasoning parametrically, such as when giving general weakest preconditions of update commands, or specifying behaviour using abstract predicates [CGZ07a]. In this sense, star and magic wand are indeed fundamental.

Finally, the decidability properties of the heap logic, which are important for implementing automatic reasoning tools which use it, have also been investigated. While it has been shown that the heap logic is in general undecidable, useful decidable fragments have been presented and studied [CYO01, BCO04a, CGH05].

Ambient Logic

Ambient Logic [CG00a, CG07] is an example of a dynamic spatial logic: it models temporal as well as spatial properties. Such logics first appeared in the context of process calculi, and accompanied a shift of focus from monolithic concurrent systems towards distributed systems. By incorporating both spatial and temporal connectives, dynamic spatial logics can distinguish between systems that differ on their distributed (spatial) structure but not necessarily on their behaviour— unlike previous systems, which typically equated logical equivalence with process bisimulation. A number of interesting properties of distributed systems are inherently spatial, including connectivity, resource availability, local channel bounds, and security.

Ambient Logic was proposed by Cardelli and Gordon to express properties of processes in the Ambient Calculus [CG00b, CGG00, CG03]. This process calculus was introduced to model distributed concurrent systems that include mobility, where mobility refers both to computations carried out on mobile devices (networks with a dynamic topology) and mobile code (executable code that is able to move around the network). The basic unit of the calculus is the 'ambient', which refers to a bounded place where a computation can occur. Ambients are hierarchically structured as unordered (unranked) trees, while processes, which act as mobility agents for the ambients, are located inside them. Examples of ambients include administrative domains bounded by firewalls, directories and files in a file system, and laptops or web pages in a network. As a result of the widespread interest generated by the Ambient Calculus, a number of alternative calculi exist, each describing different ambient movement capabilities [PV02, LS03, BCC04, TZH04].

The purpose of Ambient Logic is to succinctly describe the structural and behavioural properties of these ambient hierarchies and processes. While the full logic also includes temporal elements, a static fragment describing just structural properties is widely used. In contrast to Separation Logic, which describes flat heap structures, this consists of a spatial logic for analysing unordered labelled trees. Tree-shaped ambient hierarchies are freely generated from the parallel composition of ambients $\mathcal{A}_1 \mid \mathcal{A}_2$ and from locations containing ambients $l[\mathcal{A}]$. Hence, for example, $(l[0]|m[0])$ describes two empty ambients in parallel, while $l[m[0]]$ describes one ambient inside another. Mirroring this structure, the Ambient Logic contains, in addition to the standard propositional connectives, a parallel (de-)composition con-

nective $P_1|P_2$, which states that an ambient hierarchy can be expressed as the parallel composition of two hierarchies satisfying P_1 and P_2, and the location connective $l[P]$, which describes an ambient l with a subhierarchy satisfying P. Combined with a formula 0 representing the empty ambient, these spatial connectives can describe all exact ambient hierarchies, as in heap logic. The inclusion of classical connectives also allows formulæ such as $(\text{true} \mid l[\text{true}])$, which describes an arbitrary hierarchy containing the ambient l at the top level.

Like Separation Logic, Ambient Logic also includes the adjuncts of the spatial connectives. This includes the location adjunct $P@l$, which describes a hierarchy that satisfies P if placed inside an ambient l. Thus, $m[P@l]$ describes an ambient m which satisfies P if the m is replaced by an l, while $(m[\text{true}] \mathbin{@} l)$ is satisfied by any ambient as long as m equals l. The other adjoint is the composition adjunct $P_1 \rhd P_2$, which describes a hierarchy that satisfies P_2 whenever a hierarchy satisfying P_1 is composed in parallel. Thus, $(l[\text{true}] \rhd P)$ describes an ambient hierarchy that satisfies P in the presence of an additional ambient l, while $P \rhd \text{false}$ is satisfied by any hierarchy as long as P is unsatisfiable.

In addition to these core connectives, static Ambient Logic also includes three other operators: quantification over names, like in heap logic; a somewhere modality $\diamond P$ stating that a property P holds at some arbitrary sublocation within the hierarchy; and (sometimes) revelation and hiding operators for describing restricted names, which represent hidden ambients [CG01]. Extensions to the logic have also introduced a form of recursion based on the μ-calculus [Koz82], using a μ operator to describe the minimal set of trees satisfying a recursively defined property [Dal01]. Thus, for example, $\diamond P$ can be expressed as $\mu X.(P \vee \exists x.(x[X] \mid \text{true}))$. Finally, a horizontal iteration operator P^* based on the Kleene star has been introduced to describe a hierarchy made up of parallel hierarchies satisfying P [DLM04]. Expressible as $\mu X.(0 \vee P \mid X)$, this was used together with a quantifier-free fragment of the logic to present a regular expression language for trees.

Like for heap logic, the expressivity of the Ambient Logic has been investigated, with both adjunct elimination results [Loz04a, DGG04] and parametric expressivity and inexpressivity results [CGZ07a] given. Decidability has also been studied [CCG03, HLS02].

Other Spatial Logics

Following the work on Separation Logic and Ambient Logic, a number of other spatial logics have also been suggested. These include direct extensions of Separation Logic for modelling read and write permissions [BCOP05] and stack variables [BCY06], as well as ones introducing higher-order constructs [BBTS05] and abstract predicates [PB05], both of which help in modelling abstraction. Work on the Ambient Logic, meanwhile, has directly lead to spatial logics for trees with dangling pointers [CGG03b] and graphs [CGG02, DGG07], while the process calculus work has lead to the application of spatial logic ideas to the asynchronous π-calculus [CC02, CC03].

A slightly different type of tree logic was given by Biri and Galmiche [BG03, Bir05] for reasoning about 'resource trees', which are node-labelled trees where the nodes contain composable resources. The logic extends Bunched Logic with a location modality $[l]P$, and is unusual in that parallel composition merges nodes with identical labels: $[l]P_1|[l]P_2$ is equivalent to $[l](P_1|P_2)$.

Finally, an interesting thread of current work involves BiLog, a spatial logic for bigraphs. Bigraphs [Mil01, JM03] were introduced by Milner as a basis for a central theory of mobile interaction. They have been shown to model π-calculus and ambients [JM04], as well as Petri nets [Mil04] and CCS [Mil06]. Bigraphs contain two independent structures: a topograph, which consists of nested nodes representing locality; and a monograph, which consists of edges representing connectivity. The dynamic behaviour, meanwhile, comes from equipping bigraphs with reaction rules, forming bigraphical reactive systems (BRSs) in which bigraphs can reconfigure themselves. Recently, Conforti, Macedonio and Sassone showed how to capture bigraphs in a spatial logic called BiLog [CMS05b, Con05]. Partly inspired by Context Logic, this logic has been used to model semistructured data [CMS05a], as well as to describe other spatial logics.

2.2 Data Update and Local Reasoning

Data update is a pervasive part of computing: examples include memory updates in system software, document updates in web services, row updates in relational databases, and term and graph rewriting. At its most general, update refers to any change of state in data. In practice, however, one typically distinguishes between

in-place updates of mutable data structures and 'queries', which generate new data to represent the new state. The former are usually specified in an imperative programming style, using commands with 'side-effects' in programming languages such as C or Java. These describe computation directly in terms of a change in program state and provide a relatively low-level abstraction of the real update in the computer. Queries, in contrast, correspond more closely to a functional style of programming, where computation is viewed as the evaluation of a mathematical function resulting in a value. The primary focus of the work in this thesis is on the first type of update. These updates can be applied to a wide range of data types, from the highly-structured data in databases, to the more loosely-structured data found in web pages. The work here concentrates mainly on tree update, though two other forms of update, heap update and term rewriting, are also considered.

Formal reasoning about data update has been studied for over 35 years: Hoare Logic [Hoa69, Hoa71], a formal system for reasoning about imperative programs, was introduced by Hoare in 1969 under the influence of previous work by Floyd [Flo67]. Since then, research into update reasoning has encompassed both work on program verification, which concerns proof methods for showing correctness, safety, termination and other interesting properties of programs, and program analysis, which concentrates on techniques for automatic reasoning. Despite this long history, formal verification of update has not historically been used much in practice. However, recent research, together with technological improvements, has made widespread program verification a more plausible goal [JOW06]. In particular, work on local reasoning using Separation Logic [ORY01, YO02] has proved very successful, and has already lead to a number of practical applications.

Historically, formal update reasoning has mainly concentrated on low-level memory structures. In contrast, the work in this thesis focuses on tree update, and the application of Separation Logic-style local reasoning techniques to it. These two areas are discussed next.

Semistructured Data and Tree Update

Semistructured data [Bun97] refers to data with no rigidly predefined structure. It is typically 'self-describing', meaning that the structural information normally associated with a schema is contained within the data, and data is accessed by referring

to this structure, as opposed to, for example, giving its position in a file. External schemas may exist, but usually place only loose constraints on the data. Research into semistructured data has been strongly motivated by two factors: first, the huge amount of loosely structured data readily available from sources such as the World Wide Web [WWW07] and scientific databases such as ACeDB [ACe07]; and secondly, the desire for a flexible format for integrating heterogeneous data sources and exchanging information between globally distributed applications. Additionally, it is sometimes even advantageous to view structured data as semistructured, as this permits data to be browsed without full knowledge of its schema. The standard data model for semistructured data consists of labelled directed graphs. Nodes are viewed as objects containing either atomic data values or labelled links to other nodes. A number of variants of the semistructured data model exist, but with only minor differences [CGMH+94, Bun97, ABS99].

While the idea of semistructured data originated in the database community, the most commonly used representation of it comes from work in describing documents. XML [XML06] (eXtended Markup Language) originated as a simplified subset of SGML [GR91] (Standard Generalized Markup Language), a metalanguage for defining markup languages for documents, enabling the sharing of machine-readable documents in large projects. An XML document consists of text interspersed with markup, which separates the information into a hierarchy of character data, nested container-like elements, and element attributes. Thus the small XML fragment

$$\langle \texttt{a href="/"} \rangle \texttt{Some } \langle \texttt{b} \rangle \texttt{XML} \langle /\texttt{b} \rangle \langle /\texttt{a} \rangle$$

describes a top-level element `a` with the attribute `href="/"` and two subelements: a string element `"Some "` followed by a `b` element with a string subelement `"XML"`. This structure corresponds to an ordered node-labelled tree, with data at the leaves. Additionally, XML provides referencing mechanisms (such as `ID` and `IDREFS` attributes) that allows the simulation of arbitrary graphs. Thus, except for being ordered, XML documents correspond closely to the standard semistructured data model. Indeed, XML has become the most common way of representing semistructured data: XML-based data standards include XHTML for web pages [XHT02], SVG for vector graphics [SVG03], MathML for mathematical formulæ [Mat03] and OOXML for office documents [OOX07]. Moreover, XML is used as an intermediate format in data transfer between applications, and in 'native XML databases' [CRZ03],

which resemble relational databases, except that they use XML as the fundamental unit of logical storage, as opposed to rows in a table.

Right now, the most common way of manipulating semistructured data is with queries. Most often, 'XML transformation languages' are used to write programs that generate new XML or data based on the contents of an XML document. The two best-known transformation languages are XSLT [XSL99] and XQuery [XQu07]. XSLT consists of a 'template processor', and uses an XML-based stylesheet to specify a functional transformation of an XML document based on text-based pattern matching. It is primarily used to convert data between different XML Schemas [XML01] or into HTML documents for output as web pages. XQuery, meanwhile, resembles a database query language in the style of SQL. It is used both to transform and construct XML documents, as well as for more general information retrieval. Other interesting query languages for semistructured data include: XDuce [HP02, HP03], a statically typed pattern-matching language; CDuce [BCF03], an extension of XDuce to a general-purpose functional programming language; and logic-based query languages such as TQL [CG04] and the pattern-matching language in [CGG03a].

While queries are relatively well understood and easy to use, they are not always appropriate when manipulating semistructured data. For example, generating (or at least specifying) new data each time can be inappropriate for systems with a large amount of data or high storage costs, or for systems with an intrinsic sense of state, such as native XML databases or concurrent web-service applications. For these, it is often more natural to use an update language that updates the data in place.

At the moment, most in-place updates of semistructured data are specified using implementations of DOM [DOM04] (Document Object Model), a W3C specification for a programming interface (API) that allows programs to update XML pages. This interface presents documents as hierarchical node trees, and handles the navigation and modification of this object model. Though easy to learn and widely implemented, DOM's low-level approach nevertheless adds an extra layer of complication to programming and reasoning about updates. In contrast, high-level tree update languages aim to provide special-purpose tools for updating trees in a way that treats trees as trees, rather than as node collections. Until recently there had been very little work on such languages (to the point that most native XML databases use transformations rather than updates to update entries). However, this is beginning to change. There

is ongoing work to add update capabilities to XQuery [XQu06], while an XSLT-style language called XUpdate is also being developed [XUp00]. Other exploratory work has mostly followed the approach of XQuery, though little consensus on the details exists yet. Early research includes [Leh01, Rys02, TIHW01], while more recent work includes [SHS04, BBFV05a, GRS06].

Just like heap update, tree update raises many natural verification problems to do with safety, security, termination and preservation of types or schemas. Given the relative recency of high-level tree update, it is perhaps not surprising that there has been very little work on formal reasoning on such update so far. Scarce examples include work on verification [BBFV05b] and static analysis of optimisation [BBFV05a]. The aim in this work was to consider Hoare-style reasoning about tree update, a natural approach given the continually changing structure of semi-structured data. A few simple examples of this have been performed at the low-level in Separation Logic [ORY01], but otherwise this had not been previously attempted.

Separation Logic

Despite over 35 years of research, complexity and efficiency issues have frequently meant that Hoare reasoning was not used much in practice. However, recent work by O'Hearn, Reynolds, Yang *et al.* [ORY01, YO02, Yan01b] adapting Hoare logic to a local reasoning framework called Separation Logic has resulted in a proliferation of applications, and has helped make widespread program verification a more plausible goal [JOW06]. Separation Logic has been used to reason about a variety of data update algorithms, ranging from graph algorithms [Yan01a, BCO04b] to garbage collectors [BTSR04]. It has also been successfully applied to reasoning about concurrent heap update [O'H04, Bro04], and combined with the rely/guarantee method of shared variables to reason about interference [VP06, FFS07]. Most impressively, it has bred a multitude of automated tools, for tasks such as assertion checking [BCDO05, BCO06, GBC06], shape analysis [DOY06, CDOY06] and termination proving [TER07, CPR06, BCDO06].

The key to Separation Logic is its local reasoning approach, which is conceptually different from that of traditional Hoare Logic. This approach has been concisely summarised by the following statement:

> "To understand how a program works, it should be possible for reasoning

> and specification to be confined to the cells that the program actually accesses. The value of any other cell will remain unchanged." [ORY01]

This assertion is not automatically true. For the statement to hold, every command must *have* a well-specified part of heap that it accesses (a 'footprint'), and commands must behave in such a way that reasoning and specification can be confined to just that footprint. Such 'local' commands are natural in the context of memory update: due to the modular view of heap memory management, update commands for heaps typically consist of small data surgeries that do not depend on the entire memory of a system. Non-local updates, such as a command that frees every single memory cell, only occur when this view of memory breaks down, such as in embedded systems with limited resources.

One consequence of command locality is the existence of local specifications, or 'Small Axioms', which describe only the resources that a command or program accesses. The locality of the commands means that such specifications completely determine the commands' behaviour on all other heaps. To derive this global behaviour from the local specifications, Separation Logic introduces a simple rule that allows the inference of the invariant properties implied by locality. The rule is called the Frame Rule, since it avoids the need for Frame Axioms [MH69], which are used to explicitly describe invariants. To formulate the Frame Rule, Separation Logic uses the spatial heap logic described in the previous section. The separating conjunction $*$ expresses the separation of heaps, and hence the necessary disjointness condition on the invariants. The Frame Rule states:

$$\frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{R * P\}\ \mathbb{C}\ \{R * Q\}}\ \mathrm{mod}(\mathbb{C}) \cap \mathrm{free}(R) = \emptyset$$

This lets us take any specification $\{P\}\ \mathbb{C}\ \{Q\}$, which says that a terminating execution of $\mathbb{C}$ on a heap satisfying P always results in a heap satisfying Q and never faults, and add any invariant 'frame' given by an arbitrary formula R, where the non-modification of the frame is guaranteed by the locality of the commands. The side-condition of the rule, meanwhile, ensures the non-interference of variables, though this has recently been successfully removed by modelling variables as resources [BCY06].

The simplicity of this approach is best illustrated using an example. Consider the heap update command $[n] := v$, which replaces the value at a heap address n with a new value v. This command can be given a local specification, describing the

update on just the single heap cell with address n:

$$\{n \mapsto -\} \quad [n] := v \quad \{n \mapsto v\}$$

The precondition describes a single-cell heap with address n and unspecified value, while the postcondition states that the cell now has value v. Using the Frame Rule, it is possible to extend this local specification to descriptions of behaviour on larger heaps, by adding a heap frame satisfying some formula R (for any R satisfying the Frame Rule side-condition above):

$$\{R * (n \mapsto -)\} \quad [n] := v \quad \{R * (n \mapsto v)\}$$

One example use of the Frame Rule is to derive the weakest precondition axiom of the update command. Here, we can use the 'magic wand' formula of the heap logic to specify the frame $(n \mapsto v) \mathrel{-\!\!*} P$, which describes any heap that satisfies P when it is extended by a cell with address n and value v. Adding this to the update specification using the Frame Rule results in a postcondition satisfying P and the following triple:

$$\{((n \mapsto v) \mathrel{-\!\!*} P) * (n \mapsto -)\} \quad [n] := v \quad \{P\}$$

The precondition describes a heap containing a cell at address n which satisfies P whenever the value at this cell is set to v. This is exactly the weakest precondition of the update command.

2.3 Introduction to Context Logic

The original goal of this thesis was to try to adapt Separation Logic's local reasoning about heap update to reason about high-level tree updates. It quickly became apparent that such an adaptation was by no means trivial. An initial idea was to use Ambient Logic to express properties of static trees, in a similar fashion to the heap logic of Separation Logic. However, it turned out that this was not possible, since Ambient Logic could not express some of the complex properties associated with tree updates, including the weakest preconditions of even simple update commands. Instead, a new spatial logic had to be developed, requiring a real change in approach. This logic, called Context Logic [CGZ05, CGZ07b], forms the basis of this thesis.

With hindsight, it is perhaps not surprising that reasoning about tree update is not possible in Ambient Logic. A typical update of structured data proceeds by

identifying a portion of data to be replaced, deleting it, and inserting new data *in the same place*. This place of insertion therefore plays a crucial role in reasoning about the update. By reasoning about contexts, Context Logic can describe these update locations directly, and is therefore well equipped for such reasoning. Ambient Logic, which reasons just about trees, is not.

This can be best illustrated with a simple tree update example. The trees used are simple unranked trees, similar to the ambient structures described by Ambient Logic but with the additional restriction that the nodes names are all unique; this allows us to refer to individual nodes and specify update locations directly (just like with heap addresses and heaps). Consider a simple tree update command $[n] := t$ that replaces the subtree below a location n with some new subtree given by t (turning the tree $m_1[n[m_2[0] \mid m_3[0]] \mid m_4[0]]$ into $m_1[n[t] \mid m_4[0]]$, for example). Following the local reasoning approach of Separation Logic, one could try to specify the behaviour of this command locally, mentioning just the part of the tree that it affects, which in this case is the subtree at n. This can be easily expressed in Ambient Logic by the following Hoare Triple:

$$\{n[\text{true}]\} \quad [n] := t \quad \{n[t]\}$$

The precondition describes a tree with root node n and an arbitrary subtree, while the postcondition describes the tree with the updated subtree t. However, specifying and inferring more global descriptions of the command behaviour in Ambient Logic is less straightforward. For example, consider the weakest precondition of the command with respect to some postcondition P. Intuitively, this describes any tree that satisfies P when the subtree at n is removed and a new tree t is inserted in the same location. Unlike the local specification, this turns out not to be expressible in Ambient Logic; it can, however, be expressed in Context Logic.

The basic idea of Context Logic is simple: to reason about both data and contexts, linking the two using context application. Consequently, the logic contains two types of formulæ, one for contexts and one for data. Context application is expressed using a spatial connective $K \cdot P$, which describes the result of inserting data satisfying a data formula P into contexts satisfying a context formula K. For example, if the context formula $n[-]$ describes a node n with a hole underneath, then $n[-] \cdot \text{true}$ describes an arbitrary tree with root node n, while $\text{True} \cdot (n[-] \cdot \text{true})$ describes a tree containing a subtree with root node n inside some arbitrary context. Like Sep-

aration Logic and Ambient Logic, Context Logic also includes the right adjoints of its spatial connective. Unlike the $*$ connective of Separation Logic, however, context application is not commutative, which means that there are two distinct adjoints. The first, written $K \lhd P$, describes data that satisfies P when placed inside a context satisfying K (hence $K \lhd [-]$ is the right adjoint of $K \cdot [-]$). For example, $n[-] \lhd P$ describes a tree that satisfies P when placed underneath a node n, corresponding precisely to the $P @ n$ adjoint of Ambient Logic. The second adjoint, written $P_1 \rhd P_2$, describes contexts that satisfy P_2 whenever data satisfying P_1 is placed inside them (hence $P_1 \rhd [-]$ is the right adjoint of $[-] \cdot P_1$). For example, the formula $(0 \rhd P)$ is satisfied by any context that satisfies P when its hole is removed by placing the empty tree 0 inside it; in other words, it describes the result of adding a hole to a tree satisfying P. Returning to the update command $[n] := t$, it is now possible to express its weakest precondition by the Context Logic formula $(n[t] \rhd P) \cdot n[\text{true}]$. Just as required, this describes a tree that satisfies P when its subtree at n is removed and a new tree t is inserted in the same place.

Having introduced Context Logic, the next step is to incorporate it into a local reasoning framework in the style of Separation Logic. Like in Separation Logic, we would like to extend local specifications (such as the update specification given before) to specifications on larger trees (such as the weakest precondition above). This is done by defining a new Frame Rule for Context Logic that, instead of adding $*$-separated 'heap frames' as for Separation Logic, adds 'context frames' given by context formulæ K, using the context application connective:

$$\frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{K \cdot P\}\ \mathbb{C}\ \{K \cdot Q\}}\ \text{mod}(\mathbb{C}) \cap \text{free}(K) = \emptyset$$

Like in Separation Logic, this lets us take any specification $\{P\}\ \mathbb{C}\ \{Q\}$, which states that a terminating execution of $\mathbb{C}$ on a tree satisfying P always results in a tree satisfying Q and never faults, and add an invariant frame given by an arbitrary context formula K, where the non-modification of the frame is guaranteed by a locality property of the commands. As before, the side-condition of the rule ensures the non-interference of variables. For example, we can extend the local specification of $[n] := t$ given before to descriptions of its behaviour on larger trees by adding a context frame satisfying some formula K:

$$\{K \cdot n[\text{true}]\} \quad [n] := t \quad \{K \cdot n[t]\}$$

As in Separation Logic, we can use this approach to derive the weakest precondition axiom of the command. In this case, we simply select the context frame $(n[t] \rhd P)$, and simplify the postcondition using the standard Hoare rule of consequence:

$$\dfrac{\dfrac{\{n[\text{true}]\}\ [n] := t\ \{n[t]\}}{\{(n[t] \rhd P) \cdot n[\text{true}]\}\ [n] := t\ \{(n[t] \rhd P) \cdot n[t]\}}\ \textsc{Frame Rule}}{\{(n[t] \rhd P) \cdot n[\text{true}]\}\ [n] := t\ \{P\}}\ \textsc{Rule of Consequence}$$

Note the use of the second context application adjoint, $\rhd$, in the weakest precondition axiom above. It is this adjoint that provides the extra expressivity not found in Ambient Logic, and allows Context Logic to express the weakest precondition above. The formal proof of this difference in expressivity, which is outside the scope of this thesis, was given in [CGZ07a] using bisimulation techniques from modal logic. Specifically, the paper showed that (quantifier-free) Context Logic is strictly more expressive than (quantifier-free) Ambient Logic in the presence of propositional variables. Hence, parametric formulæ in Context Logic, such as the command weakest preconditions, cannot necessarily be expressed parametrically in Ambient Logic. For example, it can be shown that this is indeed the case for the Context Logic formula $(0 \rhd P) \cdot n[\text{true}]$ (parametric in P), which represents the weakest precondition of the command dispose n, which deletes the subtree at a location n. This can be illustrated informally. Expressing the weakest precondition of this command for even a simple postcondition typically requires a case by case analysis using the Ambient Logic. For example, for the postcondition $m_1[m_2[0]]$, the weakest precondition can be expressed most concisely by the axiom:

$$\{m_1[m_2[P] \mid P] \mid P \wedge \diamondsuit n[\text{true}]\} \quad \text{dispose } n \quad \{m_1[m_2[0]]\}$$

where $P \triangleq n[\text{true}] \vee 0$, and the result follows from the uniqueness of node identifiers. Meanwhile, the weakest precondition axiom for the postcondition $\diamondsuit m_2[\text{true}]$ can be expressed as:

$$\{\diamondsuit n[\neg\diamondsuit m_2[\text{true}]] \wedge \diamondsuit m_2[\text{true}]\} \quad \text{dispose } n \quad \{\diamondsuit m_2[\text{true}]\}$$

Clearly, the two preconditions are not parametric in the postcondition. Furthermore, we know that the first precondition must imply the second, since weakest preconditions are monotonic in the postcondition and $m_1[m_2[0]]$ implies $\diamondsuit m_2[\text{true}]$. Proving this implication, however, requires effort. In contrast, proving the corresponding

implication for the weakest preconditions in Context Logic is trivial:

$$(0 \rhd m_1[m_2[0]]) \cdot n[\text{true}] \quad \Rightarrow \quad (0 \rhd \text{True} \cdot m_2[\text{true}]) \cdot n[\text{true}]$$

Thus, Context Logic can be easily applied to reasoning about high-level tree update. In fact, Context Logic turned out to be much more far-reaching than this. It can be adapted to a wide range of other data types and forms of update, including heap update and term rewriting. Moreover, it contains an underlying logical structure that is interesting of itself: the idea of context application leads to an abstract notion of hierarchical resource splitting that generalises the flat resource model of Bunched Logic. These applications and properties are discussed in detail in the thesis.

2.4 Thesis Summary

The structure of the thesis is relatively straightforward. Following this introduction, the thesis is split into two parts: Part A (Chapters 3, 4 and 5) describes the logical foundation of Context Logic, while Part B (Chapters 6, 7 and 8) describes Context Logic-based reasoning about tree, and other data update. The thesis concludes with a discussion of its main achievements and some possible future work.

The content of the individual chapters is as follows:

Chapter 3 describes the basic theory of Context Logic, laying the groundwork and mentioning some interesting properties of the logic.

Chapter 4 describes extended theories of Context Logic, introducing additional structures such as context composition and empty data elements.

Chapter 5 contains adaptations of Context Logic to a number of structured data models, including sequences, heaps, trees and terms.

Chapter 6 describes Context Logic-based local reasoning about a basic tree update language, presenting the reasoning framework and proving its completeness.

Chapter 7 applies the same approach to two other forms of data update, namely heap update and term rewriting.

Chapter 8 considers an extended tree update language with path queries and atomic commands that act at multiple locations in the tree. This raises an interesting point concerning the link between local reasoning and local specifications.

Part A

Context Logic

Chapter 3

A Basic Theory of Context Logic

This chapter introduces the basic theory of Context Logic, a spatial logic for reasoning about data update. This includes giving its proof theory, models, semantics, and sketches of completeness and soundness results. The chapter also describes a number of useful derived properties and interesting formula classes.

3.1 Basic Context Logic

Context Logic is a spatial logic which analyses both data and contexts using the simple idea of 'context application'. It views data and contexts as two separate classes of objects, and treats the insertion of data into a context as a simple functional operation. The approach is abstract and general, just like in Bunched Logic, and its expressivity goes beyond the standard structured view of data and contexts. The application of Context Logic to structured data models is considered in Chapter 5, and throughout Part B.

Context Logic is based on two sorts of formulæ: one for reasoning about data and another for contexts. Both sorts contain the standard boolean connectives, while an application connective and its corresponding adjoints are used to join the two levels. Extensions to this simple representation are considered in Chapter 4.

Definition 3.1 (CL formulæ)**.** The formulæ of Context Logic (CL) consist of a set of *data* formulæ $P \in \mathcal{P}$ and *context* formulæ $K \in \mathcal{K}$, constructed from two infinite sets of propositional variables $\mathcal{V}_P = \{p, \dots\}$ and $\mathcal{V}_K = \{k, \dots\}$ and described by the

grammars:

$$
\begin{array}{lll}
P ::= p & & \text{propositional variables} \\
\quad \mid K \cdot P \mid K \lhd P & & \text{structural formulæ} \\
\quad \mid P \wedge P \mid P \Rightarrow P \mid \text{false} & & \text{logical formulæ} \\
\\
K ::= k & & \text{propositional variables} \\
\quad \mid P \rhd P \mid I & & \text{structural formulæ} \\
\quad \mid K \wedge K \mid K \Rightarrow K \mid \text{False} & & \text{logical formulæ}
\end{array}
$$

Binding precedence, from tightest to loosest, is $\cdot$, $\wedge$, $\rhd/\lhd$ and $\Rightarrow$.

The nub of Context Logic lies in its four structural formulæ, which describe context application, its two adjoints, and the identity context. The *context application* formula $K \cdot P$ represents the result of inserting data satisfying P into contexts satisfying K. Its two right adjoints, meanwhile, express the corresponding adjunct implications, which possess implicit universal quantifications over contexts and data. Thus, the *left-triangle* formula $K \lhd P$ is satisfied by some data if, whenever *any* context satisfying K is applied to it, the resulting data satisfies P. Similarly, the *right-triangle* formula $P_1 \rhd P_2$ describes contexts that, when applied to *any* subdata satisfying P_1, result in data satisfying P_2. Finally, the *identity context* formula I describes a context or set of contexts that does not affect data, acting as a left-identity for context application.

Both sorts of formulæ also include classical first-order logic. While it would be interesting to study intuitionistic Context Logic, this is unnecessary for reasoning about structured data update models, which satisfy all the laws of classical logic. The $\wedge$ connective is included explicitly despite being expressible using $\Rightarrow$ and false, as it simplifies the presentation of the proof rules and, together with $\Rightarrow$, provides a logical symmetry with the structural formulæ.

Definition 3.2 (CL proof theory)**.** The Hilbert-style proof theory of CL is given in Figure 3.1. This is separated into rules for propositional logic for both contexts and data, and rules for the structural formulæ.

The proof theory presentation is very similar to the Hilbert-style presentation given in [POY04] for Bunched Logic. The structural rules simply state the left-identity property of I and the adjunction properties of $\lhd$ and $\rhd$. By analogy, $\Rightarrow$ and $\wedge$ are presented in the same adjunctive style.

Identity	Falsity	Double Neg. Elim.
$P \vdash_{\mathcal{D}} P$	$\text{false} \vdash_{\mathcal{D}} P$	$(P \Rightarrow \text{false}) \Rightarrow \text{false} \vdash_{\mathcal{D}} P$

Contraction	Weakening	Exchange
$P \vdash_{\mathcal{D}} P \wedge P$	$P_1 \wedge P_2 \vdash_{\mathcal{D}} P_i$	$P_1 \wedge P_2 \vdash_{\mathcal{D}} P_2 \wedge P_1$

$$\frac{P_1 \wedge P_2 \vdash_{\mathcal{D}} P_3}{P_1 \vdash_{\mathcal{D}} P_2 \Rightarrow P_3}\,\wedge,\Rightarrow \qquad \frac{P_1 \vdash_{\mathcal{D}} P_2 \quad P_2 \vdash_{\mathcal{D}} P_3}{P_1 \vdash_{\mathcal{D}} P_3}\,\textsc{Cut}$$

Identity	Falsity	Double Neg. Elim.
$K \vdash_{\mathcal{C}} K$	$\text{False} \vdash_{\mathcal{C}} K$	$(K \Rightarrow \text{False}) \Rightarrow \text{False} \vdash_{\mathcal{C}} K$

Contraction	Weakening	Exchange
$K \vdash_{\mathcal{C}} K \wedge K$	$K_1 \wedge K_2 \vdash_{\mathcal{C}} K_i$	$K_1 \wedge K_2 \vdash_{\mathcal{C}} K_2 \wedge K_1$

$$\frac{K_1 \wedge K_2 \vdash_{\mathcal{C}} K_3}{K_1 \vdash_{\mathcal{C}} K_2 \Rightarrow K_3}\,\wedge,\Rightarrow \qquad \frac{K_1 \vdash_{\mathcal{C}} K_2 \quad K_2 \vdash_{\mathcal{C}} K_3}{K_1 \vdash_{\mathcal{C}} K_3}\,\textsc{Cut}$$

$\cdot$ identity		
$P \dashv\vdash_{\mathcal{D}} I \cdot P$	$\dfrac{K \cdot P_1 \vdash_{\mathcal{D}} P_2}{K \vdash_{\mathcal{C}} P_1 \rhd P_2}\,\cdot,\rhd$	$\dfrac{K \cdot P_1 \vdash_{\mathcal{D}} P_2}{P_1 \vdash_{\mathcal{D}} K \lhd P_2}\,\cdot,\lhd$

Figure 3.1: CL Proof Theory

CL formulæ are interpreted using a forcing semantics. Models consist of a set of contexts, a set of data, an application function, and a subset of the contexts acting as the 'identity contexts'. Data formulæ are interpreted on members of the data set, and context formulæ on members of the context set.

Definition 3.3 (CL models)**.** A CL *model* $\mathcal{M} = (\mathcal{C}, \mathcal{D}, \mathrm{ap}, \mathsf{I})$ consists of a set $\mathcal{C}$ of contexts, a set $\mathcal{D}$ of data, a partial application function $\mathrm{ap} : \mathcal{C} \times \mathcal{D} \rightharpoonup \mathcal{D}$ and a subset $\mathsf{I} \subseteq \mathcal{C}$ that acts as a left-identity for ap:

$$\forall \mathsf{d} \in \mathcal{D}.\, (\exists \mathsf{i} \in \mathsf{I}.\ \mathrm{ap}(\mathsf{i}, \mathsf{d}) = \mathsf{d}) \wedge (\forall \mathsf{i} \in \mathsf{I}.\ \mathrm{ap}(\mathsf{i}, \mathsf{d})\!\downarrow \Rightarrow \mathrm{ap}(\mathsf{i}, \mathsf{d}) = \mathsf{d})$$

where $\mathrm{ap}(\mathsf{c}, \mathsf{d})\!\downarrow$ signifies that $\mathrm{ap}(\mathsf{c}, \mathsf{d})$ is defined.

Definition 3.4 (CL forcing semantics)**.** Given a CL model $\mathcal{M}$ and an interpretation function $\sigma : (\mathcal{V}_P \to \mathcal{P}(\mathcal{D})) \times (\mathcal{V}_K \to \mathcal{P}(\mathcal{C}))$ mapping propositional variables to sets of data and contexts, the forcing semantics is given by two satisfaction relations $\sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathcal{D}} P$ and $\sigma, \mathcal{M}, \mathsf{c} \vDash_{\mathcal{C}} K$ defined inductively on the structure of data and context formulæ:

$$\begin{array}{ll}
\sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathcal{D}} p & \Leftrightarrow \mathsf{d} \in \sigma(p) \\
\sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathcal{D}} K \cdot P & \Leftrightarrow \exists \mathsf{c} \in \mathcal{C}, \mathsf{d}' \in \mathcal{D}.(\mathsf{d} = \mathrm{ap}(\mathsf{c}, \mathsf{d}') \wedge \sigma, \mathcal{M}, \mathsf{c} \vDash_{\mathcal{C}} K \wedge \sigma, \mathcal{M}, \mathsf{d}' \vDash_{\mathcal{D}} P) \\
\sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathcal{D}} K \lhd P & \Leftrightarrow \forall \mathsf{c} \in \mathcal{C}.((\mathrm{ap}(\mathsf{c}, \mathsf{d})\!\downarrow \wedge \sigma, \mathcal{M}, \mathsf{c} \vDash_{\mathcal{C}} K) \Rightarrow \sigma, \mathcal{M}, \mathrm{ap}(\mathsf{c}, \mathsf{d}) \vDash_{\mathcal{D}} P) \\
\sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathcal{D}} P_1 \wedge P_2 & \Leftrightarrow \sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathcal{D}} P_1 \wedge \sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathcal{D}} P_2 \\
\sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathcal{D}} P_1 \Rightarrow P_2 & \Leftrightarrow \sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathcal{D}} P_1 \Rightarrow \sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathcal{D}} P_2 \\
\sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathcal{D}} \mathrm{false} & \text{never}
\end{array}$$

$$\begin{array}{ll}
\sigma, \mathcal{M}, \mathsf{c} \vDash_{\mathcal{C}} k & \Leftrightarrow \mathsf{c} \in \sigma(k) \\
\sigma, \mathcal{M}, \mathsf{c} \vDash_{\mathcal{C}} P_1 \rhd P_2 & \Leftrightarrow \forall \mathsf{d} \in \mathcal{D}.((\mathrm{ap}(\mathsf{c}, \mathsf{d})\!\downarrow \wedge \sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathcal{D}} P_1) \Rightarrow \sigma, \mathcal{M}, \mathrm{ap}(\mathsf{c}, \mathsf{d}) \vDash_{\mathcal{D}} P_2) \\
\sigma, \mathcal{M}, \mathsf{c} \vDash_{\mathcal{C}} I & \Leftrightarrow \mathsf{c} \in \mathsf{I} \\
\sigma, \mathcal{M}, \mathsf{c} \vDash_{\mathcal{C}} K_1 \wedge K_2 & \Leftrightarrow \sigma, \mathcal{M}, \mathsf{c} \vDash_{\mathcal{C}} K_1 \wedge \sigma, \mathcal{M}, \mathsf{c} \vDash_{\mathcal{C}} K_2 \\
\sigma, \mathcal{M}, \mathsf{c} \vDash_{\mathcal{C}} K_1 \Rightarrow K_2 & \Leftrightarrow \sigma, \mathcal{M}, \mathsf{c} \vDash_{\mathcal{C}} K_1 \Rightarrow \sigma, \mathcal{M}, \mathsf{c} \vDash_{\mathcal{C}} K_2 \\
\sigma, \mathcal{M}, \mathsf{c} \vDash_{\mathcal{C}} \mathrm{False} & \text{never}
\end{array}$$

Using techniques from modal logic, it is possible to derive a soundness and completeness result for CL, as outlined below.

Definition 3.5 (Formula validity)**.** A formula P or K is *valid* for a given model $\mathcal{M}$ and interpretation function σ, written $\sigma, \mathcal{M} \vDash_{\mathcal{D}} P$ or $\sigma, \mathcal{M} \vDash_{\mathcal{C}} K$, if it is satisfied by

all data or contexts:

$$\sigma, \mathcal{M} \vDash_{\mathcal{D}} P \triangleq \forall \mathsf{d} \in \mathcal{D}.\ \sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathcal{D}} P$$
$$\sigma, \mathcal{M} \vDash_{\mathcal{C}} K \triangleq \forall \mathsf{c} \in \mathcal{C}.\ \sigma, \mathcal{M}, \mathsf{c} \vDash_{\mathcal{C}} K$$

Theorem 3.6 (CL soundness and completeness)**.** *The proof theory of* CL *is sound and complete with respect to the* CL *forcing relation:*

$$\text{true} \vdash_{\mathcal{D}} P \Leftrightarrow \forall \sigma, \mathcal{M}.(\sigma, \mathcal{M} \vDash_{\mathcal{D}} P)$$
$$\text{True} \vdash_{\mathcal{C}} K \Leftrightarrow \forall \sigma, \mathcal{M}.(\sigma, \mathcal{M} \vDash_{\mathcal{C}} K)$$

Proof. Soundness follows by simple induction on the derivations $\text{true} \vdash_{\mathcal{D}} P$ and $\text{True} \vdash_{\mathcal{C}} K$. The soundness of the $\cdot$ IDENTITY axiom follows from the left-identity property of I, while that of the adjunction rules follows directly from the semantics. The soundness of the classical axioms and inference rules is standard.

Showing completeness is more difficult and only a rudimentary sketch is given here. Completeness for Context Logic with relational models was proved in [CGZ07a]. This involves a reformulation of the logic as a modal logic, and an axiomatisation which uses a class of well-behaved formulæ due to Sahlqvist. Completeness then follows from a general theorem of modal logic [BdV01]. Completeness for functional models, meanwhile, is obtained by a construction which, for each relational model, gives a functional model which is bisimilar. Since bisimilar models satisfy the same formulæ, this transfers the completeness result from relational models to functional ones.

3.2 Examples

Detailed examples of applications of Context Logic to structured data models are given in Chapter 5. The following are some simple examples of CL models for use in this chapter.

Example 3.7 (CL models)**.** The following are all models of CL:

(a) $Emp = (\emptyset, \emptyset, \emptyset : \emptyset \to \emptyset, \emptyset)$, the *empty model*.

(b) $Fun_{\mathcal{D}} = (\mathcal{D} \to \mathcal{D}, \mathcal{D}, \text{ap}, \{i\})$, the *function model*, where $\mathcal{D}$ is an arbitrary set, contexts are total functions on $\mathcal{D}$, ap is function application, and i is the identity function on $\mathcal{D}$. Also,

$PartFun_{\mathcal{D},I} = (\mathcal{D} \rightharpoonup \mathcal{D}, \mathcal{D}, \mathrm{ap}, I)$, the *partial function* model, where $\mathcal{D}$ is an arbitrary set, contexts are partial functions on $\mathcal{D}$, ap is function application, and I is a set of partial identity functions, the union of whose domains is the whole of $\mathcal{D}$. Note that different choices of I result in different models.

(c) $Step = (\{0,1\}, \mathbb{N}, +, \{0\})$, the *step-by-step model*, where data corresponds to natural numbers and contexts correspond either to the identity or to the successor function. This somewhat contrived model is used to illustrate some of the quirks of Context Logic.

(d) $Mon_{\mathcal{D},\circ,0} = (\mathcal{D}, \mathcal{D}, \circ, \{0\})$, the *monoid model*, where $\mathcal{D}$ is a monoid with monoidal operator $\circ$ and identity 0. Also,

$PartMon_{\mathcal{D},\circ,0} = (\mathcal{D}, \mathcal{D}, \circ, \{0\})$, the *partial monoid model*, where $\mathcal{D}$ is a partial monoid with monoidal operator $\circ$ and identity 0.

Example 3.8 (CL constructions)**.** Given two CL models, $\mathcal{M}_1 = (\mathcal{C}_1, \mathcal{D}_1, \mathrm{ap}_1, \mathsf{I}_1)$ and $\mathcal{M}_2 = (\mathcal{C}_2, \mathcal{D}_2, \mathrm{ap}_2, \mathsf{I}_2)$, it is possible to construct the following derived models, for arbitrary $\mathsf{c}_i \in \mathcal{C}_i, \mathsf{d}_i \in \mathcal{D}_i$:

(a) $\mathcal{M}_1 + \mathcal{M}_2 = (\mathcal{C}_1 \cup \mathcal{C}_2, \mathcal{D}_1 \cup \mathcal{D}_2, \mathrm{ap}, \mathsf{I}_1 \cup \mathsf{I}_2)$, the *union model*, where $\mathrm{ap}(\mathsf{c}_i, \mathsf{d}_j) = \mathrm{ap}_i(\mathsf{c}_i, \mathsf{d}_j)$ if $i = j$ and is undefined otherwise;

(b) $\mathcal{M}_1 \times \mathcal{M}_2 = (\mathcal{C}_1 \times \mathcal{C}_2, \mathcal{D}_1 \times \mathcal{D}_2, \mathrm{ap}, \mathsf{I}_1 \times \mathsf{I}_2)$, the *product model*, where $\mathrm{ap}((\mathsf{c}_1, \mathsf{c}_2), (\mathsf{d}_1, \mathsf{d}_2)) = (\mathrm{ap}_1(\mathsf{c}_1, \mathsf{d}_1), \mathrm{ap}_2(\mathsf{c}_2, \mathsf{d}_2))$ if both applications are defined, and is undefined otherwise;

(c) $\mathcal{M}_1^{\mathcal{A}} = (\mathcal{A} \times \mathcal{C}_1, \mathcal{A} \times \mathcal{D}_1, \mathrm{ap}, \mathcal{A} \times \mathsf{I}_1)$, the *set-indexed model*, where $\mathcal{A}$ is an arbitrary index set, and $\mathrm{ap}((\mathsf{a}_1, \mathsf{c}_1), (\mathsf{a}_2, \mathsf{d}_1)) = (\mathsf{a}_1, \mathrm{ap}_1(\mathsf{c}_1, \mathsf{d}_1))$ if $\mathsf{a}_1 = \mathsf{a}_2$ and is undefined otherwise, for arbitrary $\mathsf{a}_1, \mathsf{a}_2 \in \mathcal{A}$.

Note that the set-indexed model $\mathcal{M}_1^{\mathcal{A}}$ corresponds to $\mathcal{M}_{\mathcal{A}} \times \mathcal{M}_1$, where $\mathcal{M}_{\mathcal{A}} \triangleq (\mathcal{A}, \mathcal{A}, \mathrm{ap}, \mathcal{A})$ and $\mathrm{ap}(\mathsf{a}_1, \mathsf{a}_2) = \mathsf{a}_1$ if $\mathsf{a}_1 = \mathsf{a}_2$ and is undefined otherwise.

3.3 Basic Properties and Definitions

This section presents some basic properties of CL, as well as some useful derived definitions. The first definitions are of the standard classical connectives.

Definition 3.9 (Classical connectives)**.** The standard classical connectives are defined for both data and contexts.

$$\begin{array}{cc} \neg P \triangleq P \Rightarrow \text{false} & \neg K \triangleq K \Rightarrow \text{False} \\ P_1 \vee P_2 \triangleq \neg P_1 \Rightarrow P_2 & K_1 \vee K_2 \triangleq \neg K_1 \Rightarrow K_2 \\ \text{true} \triangleq \neg\text{false} & \text{True} \triangleq \neg\text{False} \end{array}$$

Binding precedence, from tightest to loosest, is now $\neg$, $\cdot$, $\wedge$, $\vee$, $\triangleright/\triangleleft$ and $\Rightarrow$.

Note that the remaining work will take for granted the standard classical tautologies, as well as the standard derivable classical proof rules. Furthermore, it is also helpful to observe the following basic properties of the structural formulæ.

Lemma 3.10 (Basic properties)**.**

(a) variance rules:

$$\frac{K_1 \vdash_{\mathcal{C}} K_2 \quad P_1 \vdash_{\mathcal{D}} P_2}{K_1 \cdot P_1 \vdash_{\mathcal{D}} K_2 \cdot P_2} \qquad \frac{P_3 \vdash_{\mathcal{D}} P_1 \quad P_2 \vdash_{\mathcal{D}} P_4}{P_1 \triangleright P_2 \vdash_{\mathcal{C}} P_3 \triangleright P_4} \qquad \frac{K_2 \vdash_{\mathcal{C}} K_1 \quad P_1 \vdash_{\mathcal{D}} P_2}{K_1 \triangleleft P_1 \vdash_{\mathcal{D}} K_2 \triangleleft P_2}$$

(b) $\vee$-distributivity:

$$(K_1 \vee K_2) \cdot P \dashv\vdash_{\mathcal{D}} K_1 \cdot P \vee K_2 \cdot P$$
$$K \cdot (P_1 \vee P_2) \dashv\vdash_{\mathcal{D}} K \cdot P_1 \vee K \cdot P_2$$

(c) $\wedge$-semidistributivity:

$$(K_1 \wedge K_2) \cdot P \vdash_{\mathcal{D}} K_1 \cdot P \wedge K_2 \cdot P$$
$$K \cdot (P_1 \wedge P_2) \vdash_{\mathcal{D}} K \cdot P_1 \wedge K \cdot P_2$$

Proof. The proofs all follow immediately from the presence of right adjoints.

(a) the derivation for $\cdot$ is shown below; the other derivations are similar.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{K_2 \cdot P_2 \vdash_{\mathcal{D}} K_2 \cdot P_2}}{P_1 \vdash_{\mathcal{D}} P_2 \quad P_2 \vdash_{\mathcal{D}} K_2 \triangleleft (K_2 \cdot P_2)}}{P_1 \vdash_{\mathcal{D}} K_2 \triangleleft (K_2 \cdot P_2)}}{K_2 \cdot P_1 \vdash_{\mathcal{D}} K_2 \cdot P_2}}{K_1 \vdash_{\mathcal{C}} K_2 \qquad K_2 \vdash_{\mathcal{C}} P_1 \triangleright (K_2 \cdot P_2)}}{K_1 \vdash_{\mathcal{C}} P_1 \triangleright (K_2 \cdot P_2)}}{K_1 \cdot P_1 \vdash_{\mathcal{D}} K_2 \cdot P_2}$$

(b) the derivation for contexts is shown below; the data derivation is similar.

$$\dfrac{\dfrac{\dfrac{K_1 \cdot P \vdash_{\mathcal{D}} K_1 \cdot P \vee K_2 \cdot P}{K_1 \vdash_{\mathcal{C}} P \rhd (K_1 \cdot P \vee K_2 \cdot P)} \quad \dfrac{K_2 \cdot P \vdash_{\mathcal{D}} K_1 \cdot P \vee K_2 \cdot P}{K_2 \vdash_{\mathcal{C}} P \rhd (K_1 \cdot P \vee K_2 \cdot P)}}{K_1 \vee K_2 \vdash_{\mathcal{C}} P \rhd (K_1 \cdot P \vee K_2 \cdot P)}}{(K_1 \vee K_2) \cdot P \vdash_{\mathcal{D}} K_1 \cdot P \vee K_2 \cdot P}$$

$$\dfrac{\dfrac{K_1 \vdash_{\mathcal{C}} K_1 \vee K_2}{K_1 \cdot P \vdash_{\mathcal{D}} (K_1 \vee K_2) \cdot P} \quad \dfrac{K_2 \vdash_{\mathcal{C}} K_1 \vee K_2}{K_2 \cdot P \vdash_{\mathcal{D}} (K_1 \vee K_2) \cdot P}}{K_1 \cdot P \vee K_2 \cdot P \vdash_{\mathcal{D}} (K_1 \vee K_2) \cdot P}$$

(c) the derivation for contexts is shown below; the data derivation is similar.

$$\dfrac{\dfrac{K_1 \wedge K_2 \vdash_{\mathcal{C}} K_1 \quad P \vdash_{\mathcal{D}} P}{(K_1 \wedge K_2) \cdot P \vdash_{\mathcal{D}} K_1 \cdot P} \quad \dfrac{K_1 \wedge K_2 \vdash_{\mathcal{C}} K_2 \quad P \vdash_{\mathcal{D}} P}{(K_1 \wedge K_2) \cdot P \vdash_{\mathcal{D}} K_2 \cdot P}}{(K_1 \wedge K_2) \cdot P \vdash_{\mathcal{D}} (K_1 \cdot P) \wedge (K_2 \cdot P)}$$

It is also useful to define two pairs of derived formulæ: the existential duals of the context application adjoints, and formulæ corresponding to somewhere and everywhere modalities.

Definition 3.11 (Existential adjoint duals)**.** The existential duals of the structural adjoints are defined as follows:

$$P_1 \blacktriangleright P_2 \triangleq \neg(P_1 \rhd \neg P_2) \qquad K \blacktriangleleft P \triangleq \neg(K \lhd \neg P)$$

and have the following derived semantics:

$$\sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathcal{D}} K \blacktriangleleft P \;\Leftrightarrow \exists \mathsf{c} \in \mathcal{C}.(\sigma, \mathcal{M}, \mathsf{c} \vDash_{\mathcal{C}} K \wedge \mathrm{ap}(\mathsf{c}, \mathsf{d})\downarrow \wedge \sigma, \mathcal{M}, \mathrm{ap}(\mathsf{c}, \mathsf{d}) \vDash_{\mathcal{D}} P)$$

$$\sigma, \mathcal{M}, \mathsf{c} \vDash_{\mathcal{C}} P_1 \blacktriangleright P_2 \Leftrightarrow \exists \mathsf{d} \in \mathcal{D}.(\sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathcal{D}} P_1 \wedge \mathrm{ap}(\mathsf{c}, \mathsf{d})\downarrow \wedge \sigma, \mathcal{M}, \mathrm{ap}(\mathsf{c}, \mathsf{d}) \vDash_{\mathcal{D}} P_2)$$

As can be seen from their semantics, these structural duals describe existential, as opposed to universal, properties. Thus $K \blacktriangleleft P$ is satisfied by some data if it is *possible* to apply a context satisfying K to that data and obtain data satisfying P. Similarly, $P_1 \blacktriangleright P_2$ is satisfied by a context if it is possible to insert in that context data satisfying P_1 and obtain data satisfying P_2.

Definition 3.12 (Somewhere modality)**.** The somewhere and everywhere modalities are defined as follows:

$$\Diamond P \triangleq \mathrm{True} \cdot P \qquad \Box P \triangleq \neg(\Diamond \neg P)$$

and have the following derived semantics:

$$\sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathcal{D}} \Diamond P \Leftrightarrow \exists \mathsf{c} \in \mathcal{C}, \mathsf{d}' \in \mathcal{D}.(\mathsf{d} = \mathrm{ap}(\mathsf{c}, \mathsf{d}') \wedge \sigma, \mathcal{M}, \mathsf{d}' \vDash_{\mathcal{D}} P)$$
$$\sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathcal{D}} \Box P \Leftrightarrow \forall \mathsf{c} \in \mathcal{C}, \mathsf{d}' \in \mathcal{D}.(\mathsf{d} = \mathrm{ap}(\mathsf{c}, \mathsf{d}') \Rightarrow \sigma, \mathcal{M}, \mathsf{d}' \vDash_{\mathcal{D}} P)$$

Both the modalities concern context splitting. The somewhere modality, $\Diamond P$, states that it is possible to split data into a context and some subdata satisfying P. The everywhere modality, $\Box P$, states that however one splits some data into a context and subdata, the subdata must satisfy P.

These derived definitions lead to two more useful properties: a modus-ponens style result for both the universal and existential adjoints, and a simple modality result for somewhere and everywhere.

Theorem 3.13 (Modus ponens)**.** *The following modus-ponens style results hold:*

$$(P \rhd P') \cdot P \vdash_{\mathcal{D}} P' \wedge \mathrm{True} \cdot P \qquad P' \wedge \mathrm{True} \cdot P \vdash_{\mathcal{D}} (P \blacktriangleright P') \cdot P$$
$$K \cdot (K \lhd P) \vdash_{\mathcal{D}} P \wedge K \cdot \mathrm{true} \qquad P \wedge K \cdot \mathrm{true} \vdash_{\mathcal{D}} K \cdot (K \blacktriangleleft P)$$

Proof. The derivations for the top line are given below (using results in Lemma 3.10); the other derivations are similar.

$$\dfrac{\dfrac{P \rhd P' \vdash_{\mathcal{C}} P \rhd P' \quad P \rhd P' \vdash_{\mathcal{C}} \mathrm{True}}{(P \rhd P') \cdot P \vdash_{\mathcal{D}} P' \quad (P \rhd P') \cdot P \vdash_{\mathcal{D}} \mathrm{True} \cdot P}}{(P \rhd P') \cdot P \vdash_{\mathcal{D}} P' \wedge \mathrm{True} \cdot P}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathrm{True} \vdash_{\mathcal{C}} (P \blacktriangleright P') \vee (P \rhd \neg P')}{\mathrm{True} \cdot P \vdash_{\mathcal{D}} ((P \blacktriangleright P') \vee (P \rhd \neg P')) \cdot P}}{\mathrm{True} \cdot P \vdash_{\mathcal{D}} ((P \blacktriangleright P') \cdot P) \vee ((P \rhd \neg P') \cdot P)}}{P' \wedge \mathrm{True} \cdot P \vdash_{\mathcal{D}} (P' \wedge (P \blacktriangleright P') \cdot P') \vee (P' \wedge (P \rhd \neg P') \cdot P)}}{P' \wedge \mathrm{True} \cdot P \vdash_{\mathcal{D}} P' \wedge (P \blacktriangleright P') \cdot P}\,\star}{P' \wedge \mathrm{True} \cdot P \vdash_{\mathcal{D}} (P \blacktriangleright P') \cdot P}$$

where the step marked $\star$ depends on the simple result $P' \wedge (P \rhd \neg P') \cdot P \Rightarrow \mathrm{false}$.

Lemma 3.14 (Somewhere modality)**.** *Somewhere and everywhere for data satisfy the following result, corresponding to the T-axiom of modal logic:*

$$P \Rightarrow \Diamond P \qquad \Box P \Rightarrow P$$

Proof.

$$\dfrac{\dfrac{}{I \vdash_{\mathcal{C}} \text{True}} \quad P \vdash_{\mathcal{D}} P}{P \vdash_{\mathcal{D}} I \cdot P \qquad I \cdot P \vdash_{\mathcal{D}} \text{True} \cdot P} \qquad \dfrac{}{\neg P \vdash_{\mathcal{D}} \Diamond \neg P}$$

$$\dfrac{}{P \vdash_{\mathcal{D}} \text{True} \cdot P \triangleq \Diamond P} \qquad \dfrac{}{\Box P \triangleq \neg(\Diamond \neg P) \vdash_{\mathcal{D}} P}$$

Note that the 4-axiom of modal logic, which states that $\Diamond\Diamond P \Rightarrow \Diamond P$, does not hold in general. For example, in the *Step* model in Example 3.7, the number 2 satisfies $\Diamond\Diamond 0$ but not $\Diamond 0$. Chapter 4 discusses models with context composition, where the 4-axiom does always hold, and where somewhere corresponds slightly more closely to the spatial intuition of location.

Finally, given that formula validity with respect to the heap is expressible in Separation Logic, it is interesting to consider the expressibility of validity in Context Logic. In fact, it is easy to show that general validity is not expressible in Context Logic. This follows from a simple reachability argument based on the potential partiality of context application. However, it is possible to define a formula expressing a limited form of validity for elements related by a member of the identity set I: that is, a formula $\text{valid}_i(P)$ that holds for a given data element d if the assertion P is satisfied by every element d' that can be placed in some identity context $\mathsf{i} \in \mathsf{I}$ that d can. This definition turns out to be very useful (see Chapter 5) and, in the case of heaps with variables stores, coincides with the validity formula of Separation Logic.

Definition 3.15 (Partial validity)**.** Partial validity using the identity set I is defined as follows:

$$\text{valid}_i(P) \triangleq (I \wedge (\text{true} \rhd P)) \cdot \text{true}$$

and has the following derived semantics

$$\sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathcal{D}} \text{valid}_i(P) \Leftrightarrow \exists \mathsf{i} \in \mathsf{I}.\ \text{ap}(\mathsf{i}, \mathsf{d})\downarrow \wedge \forall \mathsf{d}' \in \mathcal{D}.\ (\text{ap}(\mathsf{i}, \mathsf{d}')\downarrow \Rightarrow \sigma, \mathcal{M}, \mathsf{d}' \vDash_{\mathcal{D}} P)$$

3.4 Formula Classes

The final step in this basic introduction to Context Logic is the description of a number of classes of formulæ with interesting logical properties. These correspond to, and extend, similar definitions given by Reynolds *et al.* for Separation Logic formulæ and heaps in [Rey03, Rey05]. There, Reynolds demonstrated that certain

semantically-defined classes of formulæ possess interesting properties that are useful for reasoning about heap update, memory leaks and shared-resource concurrency. Furthermore, these classes often contain easily defined syntactic subclasses, making them useful in automatic program reasoning.

The four classes of formulæ considered here are 'pureness', 'exactness', 'preciseness' and 'ubiquity'. The first three of these correspond to the homonymous classes given by Reynolds, though the adaptation to contexts involves some non-trivial extensions. The ubiquity class is new.

Unlike in [Rey03], the classes are defined logically, by declaring the validity of specific formulæ with respect to specific models and interpretation functions. This makes it easy to incorporate the derived properties into the proof theory, when adding extra structure into Context Logic, as in Chapter 4. However, the definitions are also reduced to direct semantic properties, which are general instances of the ones given by Reynolds. In Chapter 5, the formula classes are discussed in more depth for specific structured data models.

3.4.1 Pure Formulæ

Pureness in [Rey03] concerns the independence of formulæ from the heap, though not necessarily from the variable store. Here, pureness is presented instead as an insensitivity to context application. This can be split into two properties. The first, upward-closure, describes closure under context application: if d satisfies an upward-closed formula, then so must any $\mathrm{ap}(\mathsf{c}, \mathsf{d})$. The second, downward-closure, describes the reverse closure under the subdata relation: if $\mathrm{ap}(\mathsf{c}, \mathsf{d})$ satisfies a downward-closed formula, then so must d. Pure formulæ are then defined as being both upward- and downward-closed. The connection between this definition and the pureness property in [Rey03] will become clear later on.

The closure properties of pureness are expressed logically using the somewhere and everywhere modalities of Defn. 3.12. These definitions, which take the reverse form of the T-axiom of Lemma 3.14, are equivalent to a direct semantic expression of the closure properties. A logical expression of context insensitivity follows as a derived property.

Definition 3.16 (Pureness classes)**.** Given a model $\mathcal{M}$ and an interpretation function σ, a data formula P is:

(a) *upward-closed* iff $\sigma, \mathcal{M} \vDash_{\mathcal{D}} \Diamond P \Rightarrow P$;

(b) *downward-closed* iff $\sigma, \mathcal{M} \vDash_{\mathcal{D}} P \Rightarrow \Box P$; and

(c) *pure* iff it is both upward- and downward-closed.

Lemma 3.17 (Pureness semantics)**.** *A data formula P is*

(a) upward-closed iff $\forall \mathsf{c}, \mathsf{d}.\ \mathrm{ap}(\mathsf{c}, \mathsf{d})\downarrow \Rightarrow (\sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathcal{D}} P \Rightarrow \sigma, \mathcal{M}, \mathrm{ap}(\mathsf{c}, \mathsf{d}) \vDash_{\mathcal{D}} P)$

(b) downward-closed iff $\forall \mathsf{c}, \mathsf{d}.\ \mathrm{ap}(\mathsf{c}, \mathsf{d})\downarrow \Rightarrow (\sigma, \mathcal{M}, \mathrm{ap}(\mathsf{c}, \mathsf{d}) \vDash_{\mathcal{D}} P \Rightarrow \sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathcal{D}} P)$

(c) pure iff $\forall \mathsf{c}, \mathsf{d}.\ \mathrm{ap}(\mathsf{c}, \mathsf{d})\downarrow \Rightarrow (\sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathcal{D}} P \Leftrightarrow \sigma, \mathcal{M}, \mathrm{ap}(\mathsf{c}, \mathsf{d}) \vDash_{\mathcal{D}} P)$

Proof.

(a) To show that the lemma property implies upward-closure, it is sufficient to assume that $\sigma, \mathcal{M}, \mathsf{d}' \vDash_{\mathcal{D}} \Diamond P$ for some arbitrary d' and show that $\sigma, \mathcal{M}, \mathsf{d}' \vDash_{\mathcal{D}} P$. By the forcing semantics, $\mathsf{d}' = \mathrm{ap}(\mathsf{c}, \mathsf{d})$ for some c, d, where $\sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathcal{D}} P$. Therefore, by the lemma property, $\sigma, \mathcal{M}, \mathsf{d}' \vDash_{\mathcal{D}} P$.

To show that upward-closure implies the lemma property, it is sufficient to assume that $\sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathcal{D}} P$ and $\mathrm{ap}(\mathsf{c}, \mathsf{d})\downarrow$ for arbitrary c, d and show that $\sigma, \mathcal{M}, \mathrm{ap}(\mathsf{c}, \mathsf{d}) \vDash_{\mathcal{D}} P$. This follows immediately from $\mathrm{ap}(\mathsf{c}, \mathsf{d}) \vDash_{\mathcal{D}} \Diamond P$ and the upward closure property.

(b) Follows directly from (a) and Lemma 3.18a below.

(c) Follows from (a) and (b).

Lemma 3.18 (Pureness properties)**.**

(a) P is upward-closed iff $\neg P$ is downward-closed.

(b) if P is upward-closed then $\sigma, \mathcal{M} \vDash_{\mathcal{D}} K \cdot (P \wedge P') \Rightarrow P \wedge K \cdot P'$.

(c) if P is downward-closed then $\sigma, \mathcal{M} \vDash_{\mathcal{D}} P \wedge K \cdot P' \Rightarrow K \cdot (P \wedge P')$.

Proof.

(a) By contraposition, $\Diamond \neg P \Rightarrow \neg P$ iff $P \Rightarrow \neg(\Diamond \neg P) \triangleq \Box P$.

(b) $K \cdot (P \wedge P') \Rightarrow K \cdot P \wedge K \cdot P' \Rightarrow \Diamond P \wedge K \cdot P' \xRightarrow{P \text{ upward-closed}} P \wedge K \cdot P'$

(c) $P \wedge K \cdot P' \xRightarrow{P \text{ downward-closed}} \Box P \wedge K \cdot P' \Rightarrow K \cdot (P \wedge P')$

Corollary 3.19. *A pure formula P is insensitive to context application:*

$$\sigma, \mathcal{M} \vDash_{\mathcal{D}} K \cdot (P \wedge P') \Leftrightarrow P \wedge K \cdot P'$$

Examples of pure formulæ include true and false, which are pure for all models and interpretation functions, since both are trivially closed under context application and the subdata relation. For specific models, it can be useful to view the pureness closure properties as dividing the data into equivalence classes linked by context application: that is, if d is in an equivalence class then so is $\mathrm{ap}(\mathsf{c}, \mathsf{d})$, and vice versa. Thus, a pure formula is one which does not distinguish between two elements of the same class. For models that contain at least one context that can be applied to any data (which includes all the models in Example 3.7), there is just one equivalence class, and hence there are no non-trivial pure formulæ.

For a union model $\mathcal{M}_1 + \mathcal{M}_2$, however, the equivalence classes comprise the disjoint union of the equivalence classes in $\mathcal{M}_1$ and $\mathcal{M}_2$. Hence, for example, a formula satisfied by all the data elements in $\mathcal{M}_1$ would be pure. Similarly, in a set-indexed model $\mathcal{M}^{\mathcal{A}}$, data elements with different indices belong to different equivalence classes. Hence, formulæ that depend on the index, but are otherwise independent of the data value in $\mathcal{M}$, are also pure. This is exactly the case for the pure heap formulæ described by Reynolds: the base model (on which the formulæ cannot depend) is heaps, while the index (on which they can) is the variable store (see Chapter 5 for fuller details).

Examples of formulæ that are upward-closed but not pure include the formula that describes all the numbers greater than 73 in the monoidal model $Mon_{\mathbb{N},+,0}$ (which is expressible as 74^+, where $0^+ \triangleq \text{true}$ and $(n+1)^+ \triangleq \neg I \cdot n^+$). Similarly, the formula describing all numbers less than 73 (expressible as $\neg 73^+$) is downward-closed. In models where the 4-axiom of modal logic holds for the somewhere modality (that is, where $\Diamond\Diamond P \Rightarrow \Diamond P$) it is also easy to show that both $\Diamond P$ and $\text{True} \lhd P$ are upward-closed for arbitrary P, while $\Box P$ and $\text{True} \blacktriangleleft P$ are downward-closed. This includes all models containing context composition (see Chapter 4).

3.4.2 Exact Formulæ

Exactness (or strict exactness) in [Rey03] concerns the satisfaction of formulæ by at most one heap for any given store. Here, exactness is defined for both data and contexts. For the former, it corresponds to formulæ that describe data elements which

cannot be distinguished from each other using context application; for the latter, it corresponds to formulæ that describe contexts which cannot be distinguished from each other using subdata insertion. As in the case of preciseness, the data exactness definition collapses to Reynolds' notion of exactness for the appropriate heap model.

Exactness is expressed logically using implications from the existential adjoints to their universal duals. This implies that if two elements satisfying an exact formula can be placed in the same context or applied to the same data, then the results must be logically equivalent. An interesting derived property of exactness is a reverse form of the modus ponens results given in Thm. 3.13.

Definition 3.20 (Exactness classes)**.** Given a model $\mathcal{M}$ and interpretation σ,

(a) a data formula P is *exact* iff $\sigma, \mathcal{M} \vDash_{\mathcal{C}} (P \blacktriangleright P') \Rightarrow (P \rhd P')$ for all P';

(b) a context formula K is *exact* iff $\sigma, \mathcal{M} \vDash_{\mathcal{D}} (K \blacktriangleleft P') \Rightarrow (K \lhd P')$ for all P'.

Definition 3.21 (Logical equivalence)**.** Given a model $\mathcal{M}$ and interpretation function σ, two items of data are said to be logically equivalent, written $\mathsf{d} \sim \mathsf{d}'$, if there is no formula P that differentiates them: that is, there is no P where $\sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathcal{D}} P$ but $\sigma, \mathcal{M}, \mathsf{d}' \nvDash_{\mathcal{D}} P$.

Lemma 3.22 (Exactness semantics)**.** *A formula P or K is exact iff:*

(a) $\forall \mathsf{c}, \mathsf{d}_1, \mathsf{d}_2.\ \left(\bigwedge_{i=1}^{2} \mathrm{ap}(\mathsf{c}, \mathsf{d_i})\downarrow \wedge \sigma, \mathcal{M}, \mathsf{d_i} \vDash_{\mathcal{D}} P\right) \Rightarrow (\mathrm{ap}(\mathsf{c}, \mathsf{d}_1) \sim \mathrm{ap}(\mathsf{c}, \mathsf{d}_2))$

(b) $\forall \mathsf{d}, \mathsf{c}_1, \mathsf{c}_2.\ \left(\bigwedge_{i=1}^{2} \mathrm{ap}(\mathsf{c_i}, \mathsf{d})\downarrow \wedge \sigma, \mathcal{M}, \mathsf{c_i} \vDash_{\mathcal{C}} K\right) \Rightarrow (\mathrm{ap}(\mathsf{c}_1, \mathsf{d}) \sim \mathrm{ap}(\mathsf{c}_2, \mathsf{d}))$

Proof.

(a) To show that the lemma property implies exactness, it is sufficient to assume that $\sigma, \mathcal{M}, \mathsf{c} \vDash_{\mathcal{C}} (P \blacktriangleright P')$ for some P' and show that $\sigma, \mathcal{M} \vDash_{\mathcal{C}} (P \rhd P')$. The assumption implies that there is some d such that $\sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathcal{D}} P$ and $\mathrm{ap}(\mathsf{c}, \mathsf{d})$ is defined and satisfies P'. By the lemma property, if there is a d' such that $\sigma, \mathcal{M}, \mathsf{d}' \vDash_{\mathcal{D}} P$ and $\mathrm{ap}(\mathsf{c}, \mathsf{d}')\downarrow$, then $\mathrm{ap}(\mathsf{c}, \mathsf{d}')$ is logically equivalent to $\mathrm{ap}(\mathsf{c}, \mathsf{d})$, and must therefore satisfy P'. Thus, $\sigma, \mathcal{M} \vDash_{\mathcal{C}} (P \rhd P')$.

To show that exactness implies the lemma property, it is sufficient to assume that $\mathrm{ap}(\mathsf{c}, \mathsf{d_i})\downarrow$ and $\sigma, \mathcal{M}, \mathsf{d_i} \vDash_{\mathcal{D}} P$ for some arbitrary context c and data elements d_1, d_2 and show that $\mathrm{ap}(\mathsf{c}, \mathsf{d}_1) \sim \mathrm{ap}(\mathsf{c}, \mathsf{d}_2)$. For a proof by contradiction, assume there exists a P' such that $\sigma, \mathcal{M}, \mathrm{ap}(\mathsf{c}, \mathsf{d}_1) \vDash_{\mathcal{D}} P$ but $\sigma, \mathcal{M}, \mathrm{ap}(\mathsf{c}, \mathsf{d}_2) \nvDash_{\mathcal{D}} P$. Then $\sigma, \mathcal{M}, \mathsf{c} \vDash_{\mathcal{C}} (P \blacktriangleright P')$ but $\sigma, \mathcal{M}, \mathsf{c} \nvDash_{\mathcal{C}} (P \rhd P')$, contradicting exactness.

(b) Similar to (a).

Lemma 3.23 (Exactness properties)**.** *For P or K exact, the modus-ponens style implications in Thm. 3.13 hold in reverse: that is, for all P'*

(a) $\sigma, \mathcal{M} \vDash_{\mathcal{D}} (P \blacktriangleright P') \cdot P \Rightarrow P' \wedge \mathrm{True} \cdot P \quad \sigma, \mathcal{M} \vDash_{\mathcal{D}} P' \wedge \mathrm{True} \cdot P \Rightarrow (P \triangleright P') \cdot P$

(b) $\sigma, \mathcal{M} \vDash_{\mathcal{D}} K \cdot (K \blacktriangleleft P') \Rightarrow P' \wedge K \cdot \mathrm{true} \quad \sigma, \mathcal{M} \vDash_{\mathcal{D}} P' \wedge K \cdot \mathrm{true} \Rightarrow K \cdot (K \triangleleft P')$

Proof. Follows immediately from Thm. 3.13 and the functoriality of application.

(a) $(P \blacktriangleright P') \cdot P \Rightarrow (P \triangleright P') \cdot P \Rightarrow P' \wedge \mathrm{True} \cdot P \Rightarrow (P \blacktriangleright P') \cdot P \Rightarrow (P \triangleright P') \cdot P$

(b) $K \cdot (K \blacktriangleleft P') \Rightarrow K \cdot (K \triangleleft P') \Rightarrow P' \wedge K \cdot \mathrm{true} \Rightarrow K \cdot (K \blacktriangleleft P') \Rightarrow K \cdot (K \triangleleft P')$

Despite the symmetry of the definitions, exact data formulæ turn out to be much more useful in modelling data update than exact context formulæ, and are explored in more detail. This is mainly due to the emphasis on data in reasoning about update.

In models with a singleton identity context that can be applied to everything, exact data formulæ are only satisfied by data elements that are logically equivalent (by Lemma 3.22). In the case when logical equivalence corresponds to equality, exactness corresponds to satisfaction by at most one element: for example in the model $Mon_{\mathbb{N},+,0}$ considered above. In the cases where logical equivalence isn't equality, the exactness condition can sometimes be stronger than just logical equivalence. Consider, for example, the model $Fun_{\{1,2,3,4\}}$ with propositional variables p, q, and an interpretation $\sigma(p) = \{3\}$, $\sigma(q) = \{4\}$. Then $P \triangleq \neg(p \vee q)$ is satisfied by elements 1 and 2, which are logically equivalent. However, P is not exact as the context $\{1 \mapsto 3, 2 \mapsto 4, 3 \mapsto 3, 4 \mapsto 4\}$ satisfies $P \blacktriangleright p$ but not $P \triangleright p$.

In general, two data elements that satisfy an exact data formula are not necessarily logically equivalent if they cannot be put in the same member of the identity context. For example, in a set-indexed model $\mathcal{M}^{\mathcal{A}}$, a formula describing at most one data element in $\mathcal{M}$ per index is exact, since context application is only defined for pairs with identical indices. This is precisely the situation for the exact heap formulæ described by Reynolds, which are satisfied by at most one heap for any given store.

3.4.3 Precise Formulæ

Preciseness in [Rey05] concerns the satisfaction of a formula by at most one subheap for any given heap, and is a useful concept in modelling update, and reasoning about

memory leaks and concurrency. Here, preciseness is extended to a property called 'pair-preciseness'. This is defined for a pair of formulæ, a context formula K and data formula P, and holds if there is at most one way, up to logical equivalence, to split any given data into a context satisfying K and subdata satisfying P. This modelling of unique splittings closely resembles the specification of unique locations in data update, and is therefore particularly interesting for structured data models (see Chapter 5). It is possible to define data preciseness and context preciseness in terms of pair-preciseness, by removing the restriction on contexts (for data preciseness) or data (for context preciseness). Data preciseness corresponds to the view of preciseness given by Reynolds.

Pair-preciseness is expressed logically using an $\wedge$-distributivity rule for context application. This specifies that any two splittings into contexts satisfying K and subdata satisfying P imply a single splitting that is logically equivalent. In the case of data and context preciseness, the rule collapses to smaller $\wedge$-distributivity rules. An interesting derived property of data and context preciseness is an elimination property given below.

Definition 3.24 (Preciseness classes)**.** Given a model $\mathcal{M}$ and interpretation σ,

(a) a pair of context-data formulæ (K, P) is *pair-precise* iff for all P_i, K_i

$$\begin{aligned}\sigma, \mathcal{M} \vDash_{\mathcal{D}} &(K \wedge K_1) \cdot (P \wedge P_1) \wedge (K \wedge K_2) \cdot (P \wedge P_2) \\ &\Rightarrow (K \wedge K_1 \wedge K_2) \cdot (P \wedge P_1 \wedge P_2)\end{aligned}$$

(b) a data formula P is *precise* iff (True, P) is pair-precise;

(c) a context formula K is precise iff (K, true) is pair-precise.

Corollary 3.25. *For K or P precise, the $\wedge$-semidistributivity implications in Lemma 3.10 hold in reverse:*

$$\sigma, \mathcal{M} \vDash_{\mathcal{D}} K_1 \cdot P \wedge K_2 \cdot P \Rightarrow (K_1 \wedge K_2) \cdot P$$
$$\sigma, \mathcal{M} \vDash_{\mathcal{D}} K \cdot P_1 \wedge K \cdot P_2 \Rightarrow K \cdot (P_1 \wedge P_2)$$

Lemma 3.26 (Preciseness semantics)**.** (K, P) *is pair-precise iff:*

$$\begin{aligned}\forall \mathsf{c_1}, \mathsf{c_2}, \mathsf{d_1}, \mathsf{d_2}.\, &\left(\textstyle\bigwedge_{i=1}^{2} \mathrm{ap}(\mathsf{c_i}, \mathsf{d_i})\downarrow \wedge\, \sigma, \mathcal{M}, \mathsf{d_i} \vDash_{\mathcal{D}} P \wedge \sigma, \mathcal{M}, \mathsf{c_i} \vdash_{\mathcal{C}} K\right) \\ &\wedge (\mathrm{ap}(\mathsf{c_1}, \mathsf{d_1}) = \mathrm{ap}(\mathsf{c_2}, \mathsf{d_2})) \Rightarrow (\mathsf{c_1} \sim \mathsf{c_2}) \wedge (\mathsf{d_1} \sim \mathsf{d_2})\end{aligned}$$

where $\sim$ is a logical equivalence, defined in the previous subsection.

Proof. To show that the lemma condition above implies pair-preciseness, assume that $\sigma, \mathcal{M}, \mathsf{c_i} \vDash_{\mathcal{C}} K \wedge K_i$, $\sigma, \mathcal{M}, \mathsf{d_i} \vDash_{\mathcal{D}} P \wedge P_i$, $\mathrm{ap}(\mathsf{c_i}, \mathsf{d_i})\downarrow$ and $\mathrm{ap}(\mathsf{c_1}, \mathsf{d_2}) = \mathrm{ap}(\mathsf{c_2}, \mathsf{d_2})$ for some arbitrary $\mathsf{c_1}, \mathsf{c_2}, \mathsf{d_1}, \mathsf{d_2}$ and show that $\sigma, \mathcal{M}, \mathrm{ap}(\mathsf{c_1}, \mathsf{d_1}) \vDash_{\mathcal{D}} (K \wedge K_1 \wedge K_2) \cdot (P \wedge P_1 \wedge P_2)$. By the lemma property, $\mathsf{c_1} \sim \mathsf{c_2}$ and $\mathsf{d_1} \sim \mathsf{d_2}$. Hence $\sigma, \mathcal{M}, \mathsf{c_1} \vDash_{\mathcal{C}} K_2$ and $\sigma, \mathcal{M}, \mathsf{d_1} \vDash_{\mathcal{D}} P_2$ since $\sigma, \mathcal{M}, \mathsf{c_2} \vDash_{\mathcal{C}} K_2$ and $\sigma, \mathcal{M}, \mathsf{d_2} \vDash_{\mathcal{D}} P_2$. Therefore, $\sigma, \mathcal{M}, \mathrm{ap}(\mathsf{c_1}, \mathsf{d_1}) \vDash_{\mathcal{D}} (K \wedge K_1 \wedge K_2) \cdot (P \wedge P_1 \wedge P_2)$.

To show that pair-preciseness implies the lemma condition, assume that $\mathsf{c_i} \vDash_{\mathcal{C}} K$, $\mathsf{d_i} \vDash_{\mathcal{D}} P$, $\mathrm{ap}(\mathsf{c_i}, \mathsf{d_i})\downarrow$ and $\mathrm{ap}(\mathsf{c_1}, \mathsf{d_1}) = \mathrm{ap}(\mathsf{c_2}, \mathsf{d_2})$ and show that $\mathsf{c_1} \sim \mathsf{c_2}$ and $\mathsf{d_1} \sim \mathsf{d_2}$. For a proof by contradiction, assume first that $\mathsf{c_1} \vDash_{\mathcal{C}} K'$ but $\mathsf{c_2} \nvDash_{\mathcal{C}} K'$ for some K'. Then $\mathrm{ap}(\mathsf{c_1}, \mathsf{d_1}) \vDash_{\mathcal{D}} (K \wedge K') \cdot (P \wedge P)$ and $\mathrm{ap}(\mathsf{c_2}, \mathsf{d_2}) \vDash_{\mathcal{D}} (K \wedge \neg K') \cdot (P \wedge P)$. But since $\mathrm{ap}(\mathsf{c_1}, \mathsf{d_1}) = \mathrm{ap}(\mathsf{c_2}, \mathsf{d_2})$, pair-preciseness implies $\mathrm{ap}(\mathsf{c_1}, \mathsf{d_1}) \vDash_{\mathcal{D}} (K \wedge K' \wedge \neg K') \cdot (P \wedge P \wedge P)$, a contradiction. Hence $\mathsf{c_1} \sim \mathsf{c_2}$. The case for $\mathsf{d_1} \sim \mathsf{d_2}$ is similar.

Lemma 3.27 (Preciseness properties)**.** *If P or K are precise, then the following elimination properties hold for all K' and P':*

$$\sigma, \mathcal{M} \vDash_{\mathcal{C}} P \blacktriangleright (K' \cdot P) \Rightarrow K' \qquad \sigma, \mathcal{M} \vDash_{\mathcal{D}} K \blacktriangleleft (K \cdot P') \Rightarrow P'$$

Proof. The proof for P is as follows; the proof for K is similar. The premise of the proof is an immediate corollary of the definition of preciseness.

$$\dfrac{\dfrac{\dfrac{((\neg K') \cdot P) \wedge (K' \cdot P) \vdash_{\mathcal{D}} \text{false}}{(\neg K') \cdot P \vdash_{\mathcal{D}} \neg (K' \cdot P)}}{\neg K' \vdash_{\mathcal{C}} P \rhd \neg (K' \cdot P)}}{P \blacktriangleright (K' \cdot P) \vdash_{\mathcal{C}} K'}$$

Note that in general, the pair-preciseness of a pair of formulæ does not imply the preciseness of either formula. For example, in the monoidal model $Mon_{\mathbb{N},+,0}$ containing propositional variables k, p, and an interpretation $\sigma(k) = \{0, 1\}$, $\sigma(p) = \{n \in \mathbb{N} \mid n \text{ is even}\}$, the formula pair (k, p) is pair-precise: there is only one way of expressing an arbitrary natural number as the sum of an even number and a one or zero. However, clearly neither k nor p are precise by themselves. Further examples of pair-precise pairs are given in Chapter 5.

In [Rey03], exactness implied preciseness, but this is also not true in general. For example, the formula p expressing any $\mathsf{d} \in \mathcal{D}$ in the function model $Fun_{\mathcal{D}}$ is

exact but not precise: there are many ways of splitting a given value into d and a function that maps d to that value. Conversely, there are precise formulæ that are not exact. For example, the data formula p in the *Step* model with an interpretation $\sigma(p) = \{n \in \mathbb{N} \mid n \text{ is even}\}$ is not exact but is precise, for the same reasons as in the $Mon_{\mathbb{N},+,0}$ example above.

3.4.4 Ubiquitous Formulæ

The final formula class, ubiquity, was not considered in [Rey03], but is used in Chapter 4 to help characterise empty data elements. Like pureness, ubiquity is made up of two subproperties. The first is omnipresence, which describes data and contexts that can be found in at least one splitting of any data element. The second is omniplaceability, which describes data and contexts that can be combined with any context or data. Both properties entail additional derived properties: omnipresence implies a weaker form of the existential modus ponens rule, while omniplaceability implies an implication from the universal adjoints to the existential ones.

Definition 3.28 (Ubiquity classes)**.** Given a model $\mathcal{M}$ and an interpretation function σ, a formula P or K is:

(a) *omnipresent* iff $\sigma, \mathcal{M} \vDash_{\mathcal{D}} \text{True} \cdot P$ *or* $\sigma, \mathcal{M} \vDash_{\mathcal{C}} K \cdot \text{true}$;

(b) *omniplaceable* iff $\sigma, \mathcal{M} \vDash_{\mathcal{C}} P \blacktriangleright \text{true}$ *or* $\sigma, \mathcal{M} \vDash_{\mathcal{D}} K \blacktriangleleft \text{true}$;

(c) *ubiquitous* iff it is omnipresent and omniplaceable.

Lemma 3.29 (Ubiquity semantics)**.**

(a) P *is omnipresent iff* $\forall \mathsf{d}.\exists \mathsf{c}, \mathsf{d}'.\, (\mathsf{d} = \text{ap}(\mathsf{c}, \mathsf{d}') \wedge \sigma, \mathcal{M}, \mathsf{d}' \vDash_{\mathcal{D}} P)$
K *is omnipresent iff* $\forall \mathsf{d}.\exists \mathsf{c}, \mathsf{d}'.\, (\mathsf{d} = \text{ap}(\mathsf{c}, \mathsf{d}') \wedge \sigma, \mathcal{M}, \mathsf{c} \vDash_{\mathcal{C}} K)$

(b) P *is omniplaceable iff* $\forall \mathsf{c}.\exists \mathsf{d}.\, (\text{ap}(\mathsf{c}, \mathsf{d})\downarrow \wedge \sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathcal{D}} P)$
K *is omniplaceable iff* $\forall \mathsf{d}.\exists \mathsf{c}.\, (\text{ap}(\mathsf{c}, \mathsf{d})\downarrow \wedge \sigma, \mathcal{M}, \mathsf{c} \vDash_{\mathcal{C}} K)$

Proof. Follows directly from the definition of the forcing semantics.

Lemma 3.30 (Ubiquity properties)**.** *Omnipresence implies a weaker existential modus ponens result, while omniplaceability implies the reverse implication of the exactness conditions:*

(a) *for P omnipresent:* $\sigma, \mathcal{M} \models_{\mathcal{D}} P' \Rightarrow (P \blacktriangleright P') \cdot P$ *for all* P'
for K omnipresent: $\sigma, \mathcal{M} \models_{\mathcal{D}} P' \Rightarrow K \cdot (K \blacktriangleleft P')$ *for all* P'

(b) *for P omniplaceable:* $\sigma, \mathcal{M} \models_{\mathcal{C}} P \rhd P' \Rightarrow P \blacktriangleright P'$ *for all* P'
for K omniplaceable: $\sigma, \mathcal{M} \models_{\mathcal{C}} K \lhd P \Rightarrow K \blacktriangleleft P$ *for all* P'

Proof.

(a) Follows directly from Thm. 3.13 and the definition of omnipresence.

(b) For P omniplaceable, $P \blacktriangleright$ true implies $P \blacktriangleright (P' \vee \neg P')$, which in turn implies $(P \blacktriangleright P') \vee (P \blacktriangleright \neg P')$. From this it is possible to derive $P \rhd P' \Rightarrow P \blacktriangleright P'$ by elimination, since $(P \rhd P') \wedge (P \blacktriangleright \neg P') \Rightarrow$ false. The proof for K is similar.

Examples of ubiquitous formulæ include I, which is trivially ubiquitous in all models. Furthermore, if $P \Rightarrow Q$ and P are both ubiquitous then Q is ubiquitous. Hence, the context formula True is always ubiquitous, since $I \Rightarrow$ True. In contrast, Lemma 3.29 shows that the data formula true is ubiquitous only when context application is total (implying omniplaceability) and surjective (implying omnipresence).

Examples of omnipresent formulæ that are not omniplaceable can be found in the partial function models $PartFun_{\mathcal{D},I}$ in Example 3.7. All non-false data formulæ are omnipresent but none are omniplaceable, due to the presence of the everywhere-undefined context. Similarly, examples of omniplaceable formulæ that are not omnipresent can be found in the *Step* model, where all formulæ denoting singleton data sets are omniplaceable but none are omnipresent.

Chapter 4

Extended Theories of Context Logic

This chapter presents extended theories of Context Logic, which model additional structures such as context composition and empty data elements. It defines a spatial connective on data and data formulæ that can be used to embed Bunched Logic into Context Logic. Finally, it gives an alternative presentation of Context Logic, using only a projection/embedding relation between contexts and data and context composition.

4.1 Context Composition

Given the functional nature of context application used in Context Logic, it is natural to consider the composition of contexts. While not all CL models can be extended to include context composition, the interesting structural ones can: in trees, for example, context composition corresponds to placing one tree context inside another. For some of these models it is already possible to express context composition using CL, but it is not possible to do this in general. This section describes the result of adding context composition, together with its two right adjoints, directly to the logic.

Definition 4.1 ($\mathrm{CL}_\circ$ formulæ)**.** The formulæ of Context Logic with Composition ($\mathrm{CL}_\circ$) consist of the same data and context assertions as CL (Defn. 3.1), with addi-

tional assertions for context composition and its right adjoints.

$$
\begin{array}{lll}
P ::= p & & \text{propositional variables} \\
\quad \mid K \cdot P \mid K \lhd P & & \text{structural formulæ} \\
\quad \mid P \wedge P \mid P \Rightarrow P \mid \text{false} & & \text{logical formulæ} \\
\\
K ::= k & & \text{propositional variables} \\
\quad \mid I \mid P \rhd P \mid \boldsymbol{K \circ K} \mid \boldsymbol{K \multimap K} \mid \boldsymbol{K \circ\!\!- K} & & \textbf{structural formulæ} \\
\quad \mid K \wedge K \mid K \Rightarrow K \mid \text{False} & & \text{logical formulæ}
\end{array}
$$

The additional formulæ describe context composition and its right adjoints. The *context composition* formula $K_1 \circ K_2$ describes the contexts obtained from functionally composing a context satisfying K_1 with one satisfying K_2. Like for context application, the logic also includes the two right adjoints of composition, $K_1 \multimap K_2$ and $K_1 \circ\!\!- K_2$: the former is satisfied by a context if whenever one composes a context satisfying K_1 on the right then the result satisfies K_2; the latter is satisfied if whenever one composes a context satisfying K_1 on the left then the result satisfies K_2.

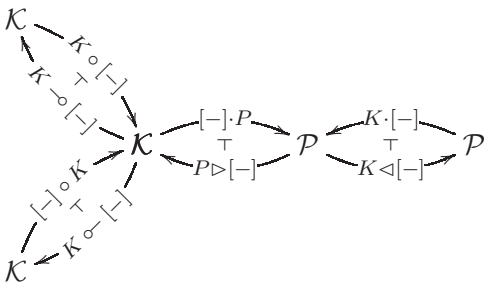
Definition 4.2 ($\mathrm{CL}_\circ$ proof theory)**.** The proof theory of $\mathrm{CL}_\circ$ is the same as that of CL (Defn. 3.2), extended by the following rules for composition:

$$
\begin{array}{ll}
\circ \text{ ASSOCIATIVITY} & \circ \text{ IDENTITY} \\
(K_1 \circ K_2) \circ K_3 \dashv\vdash_{\mathcal{C}} K_1 \circ (K_2 \circ K_2) & K \circ I \dashv\vdash_{\mathcal{C}} K \dashv\vdash_{\mathcal{C}} I \circ K
\end{array}
$$

$$
\frac{K_1 \circ K_2 \vdash_{\mathcal{C}} K_3}{K_1 \vdash_{\mathcal{C}} K_2 \multimap K_3} \circ, \multimap \qquad \frac{K_1 \circ K_2 \vdash_{\mathcal{C}} K_3}{K_2 \vdash_{\mathcal{C}} K_1 \circ\!\!- K_3} \circ, \circ\!\!-
$$

$$
\begin{array}{l}
\text{COMPOSITION} \\
(K_1 \circ K_2) \cdot P \dashv\vdash_{\mathcal{D}} K_1 \cdot (K_2 \cdot P)
\end{array}
$$

The additional rules simply state the associativity and identity properties of composition, the adjunction properties of $\multimap$ and $\circ\!\!-$, and the basic composition property that links context composition to context application. The expanded adjunctive structure can be illustrated by the following diagram:



Models for $\mathrm{CL}_\circ$ consist of CL models with the addition of a possibly partial composition operator on contexts.

Definition 4.3 ($\mathrm{CL}_\circ$ models)**.** A *model* $\mathcal{M} = (\mathcal{C}, \mathcal{D}, \mathrm{cp}, \mathrm{ap}, \mathsf{I})$ of $\mathrm{CL}_\circ$ is a model of CL (Defn. 3.3) with the addition of a partial composition function $\mathrm{cp} : \mathcal{C} \times \mathcal{C} \rightharpoonup \mathcal{C}$, which is associative with an identity set I:

$$\forall \mathsf{c}_1, \mathsf{c}_2, \mathsf{c}_3 \in \mathcal{C}.\ \mathrm{cp}(\mathsf{c}_1, \mathrm{cp}(\mathsf{c}_2, \mathsf{c}_3)) = \mathrm{cp}(\mathrm{cp}(\mathsf{c}_1, \mathsf{c}_2), \mathsf{c}_3)$$

$$\forall \mathsf{c} \in \mathcal{C}.\ (\exists \mathsf{i} \in \mathsf{I}.\ \mathrm{cp}(\mathsf{i}, \mathsf{c}) = \mathsf{c}) \wedge (\forall \mathsf{i} \in \mathsf{I}.\ \mathrm{cp}(\mathsf{i}, \mathsf{c})\downarrow \Rightarrow \mathrm{cp}(\mathsf{i}, \mathsf{c}) = \mathsf{c})$$
$$\forall \mathsf{c} \in \mathcal{C}.\ (\exists \mathsf{i} \in \mathsf{I}.\ \mathrm{cp}(\mathsf{c}, \mathsf{i}) = \mathsf{c}) \wedge (\forall \mathsf{i} \in \mathsf{I}.\ \mathrm{cp}(\mathsf{c}, \mathsf{i})\downarrow \Rightarrow \mathrm{cp}(\mathsf{c}, \mathsf{i}) = \mathsf{c})$$

and which satisfies the standard function composition property:

$$\forall \mathsf{c}_1, \mathsf{c}_2 \in \mathcal{C}, \mathsf{d} \in \mathcal{D}.(\mathrm{ap}(\mathrm{cp}(\mathsf{c}_1, \mathsf{c}_2), \mathsf{d}) = \mathrm{ap}(\mathsf{c}_1, \mathrm{ap}(\mathsf{c}_2, \mathsf{d})))$$

Definition 4.4 ($\mathrm{CL}_\circ$ forcing semantics)**.** Given a model $\mathcal{M}$ of $\mathrm{CL}_\circ$ and an interpretation function $\sigma : (\mathcal{V}_\mathcal{D} \to \mathcal{P}(\mathcal{D})) \times (\mathcal{V}_\mathcal{C} \to \mathcal{P}(\mathcal{C}))$ mapping propositional variables to data and context sets, the forcing semantics is given by two satisfaction relations $\sigma, \mathcal{M}, \mathsf{d} \vDash_\mathcal{D} P$ and $\sigma, \mathcal{M}, \mathsf{c} \vDash_\mathcal{C} K$ defined as in Defn. 3.4 and extended as follows:

$$\sigma, \mathcal{M}, \mathsf{c} \vDash_\mathcal{C} K_1 \circ K_2 \quad \Leftrightarrow \exists \mathsf{c}_1, \mathsf{c}_2 \in \mathcal{C}.(\mathsf{c} = \mathsf{c}_1 \circ \mathsf{c}_2 \wedge \sigma, \mathcal{M}, \mathsf{c}_1 \vDash_\mathcal{C} K_1 \wedge \sigma, \mathcal{M}, \mathsf{c}_2 \vDash_\mathcal{C} K_2)$$
$$\sigma, \mathcal{M}, \mathsf{c} \vDash_\mathcal{C} K_1 \multimap K_2 \Leftrightarrow \forall \mathsf{c}_1 \in \mathcal{C}.((\mathsf{c} \circ \mathsf{c}_1\downarrow \wedge \sigma, \mathcal{M}, \mathsf{c}_1 \vDash_\mathcal{C} K_1) \Rightarrow \sigma, \mathcal{M}, \mathsf{c} \circ \mathsf{c}_1 \vDash_\mathcal{D} K_2)$$
$$\sigma, \mathcal{M}, \mathsf{c} \vDash_\mathcal{C} K_1 \circ\!\!- K_2 \Leftrightarrow \forall \mathsf{c}_1 \in \mathcal{C}.((\mathsf{c}_1 \circ \mathsf{c}\downarrow \wedge \sigma, \mathcal{M}, \mathsf{c}_1 \vDash_\mathcal{C} K_1) \Rightarrow \sigma, \mathcal{M}, \mathsf{c}_1 \circ \mathsf{c} \vDash_\mathcal{D} K_2)$$

As mentioned before, not all models of CL allow context composition. For example, the model *Step* in Example 3.7 does not: there is no context $1 \circ 1$ satisfying $(1 \circ 1) + n = 1 + (1 + n)$. Below are examples of some CL models that do allow context composition.

Example 4.5 ($\mathrm{CL}_\circ$ models)**.** The following extensions to CL models from Example 3.7 are all models of $\mathrm{CL}_\circ$:

(a) $Emp = (\emptyset, \emptyset, \emptyset : \emptyset \to \emptyset, \emptyset : \emptyset \to \emptyset, \emptyset)$, the *empty model*, where the composition function is $\emptyset$.

(b) $Fun_\mathcal{D} = (\mathcal{D} \to \mathcal{D}, \mathcal{D}, \mathrm{cp}, \mathrm{ap}, \{i\})$, the *function model*, where cp is functional composition; also

$PartFun_{\mathcal{D},I} = (\mathcal{D} \rightharpoonup \mathcal{D}, \mathcal{D}, \mathrm{cp}, \mathrm{ap}, I)$, the *partial function model*, where cp is partial composition, with the additional restriction that I must contain the total identity function. This is necessary to ensure that I is an identity for context composition, as well as for application.

(d) $Mon_{\mathcal{D},\circ,0} = (\mathcal{D}, \mathcal{D}, \circ, \circ, \{0\})$, the *monoid model*, where the composition function and the application function are the same;

$PartMon_{\mathcal{D},\circ,0} = (\mathcal{D}, \mathcal{D}, \circ, \circ, \{0\})$, the *partial monoid model*.

Example 4.6 ($\mathrm{CL}_\circ$ constructions)**.** Given two $\mathrm{CL}_\circ$ models, $\mathcal{M}_1 = (\mathcal{C}_1, \mathcal{D}_1, \mathrm{cp}_1, \mathrm{ap}_1, \mathsf{I}_1)$ and $\mathcal{M}_2 = (\mathcal{C}_2, \mathcal{D}_2, \mathrm{cp}_2, \mathrm{ap}_2, \mathsf{I}_2)$, it is possible to extend all the constructions given in Example 3.8 as follows: for arbitrary $\mathsf{c}_i, \mathsf{c}'_\mathsf{i} \in \mathcal{C}_i$,

(a) $\mathcal{M}_1 + \mathcal{M}_2 = (\mathcal{C}_1 \cup \mathcal{C}_2, \mathcal{D}_1 \cup \mathcal{D}_2, \mathbf{cp}, \mathrm{ap}, \mathsf{I}_1 \cup \mathsf{I}_2)$, the *union model*, where $\mathrm{cp}(\mathsf{c}_i, \mathsf{c}'_\mathsf{j}) = \mathrm{cp}_i(\mathsf{c}_i, \mathsf{c}'_\mathsf{j})$ if $i = j$ and is undefined otherwise;

(b) $\mathcal{M}_1 \times \mathcal{M}_2 = (\mathcal{C}_1 \times \mathcal{C}_2, \mathcal{D}_1 \times \mathcal{D}_2, \mathbf{cp}, \mathrm{ap}, \mathsf{I}_1 \times \mathsf{I}_2)$, the *product model*, where $\mathrm{cp}((\mathsf{c}_1, \mathsf{c}_2), (\mathsf{c}'_1, \mathsf{c}'_2)) = (\mathrm{cp}_1(\mathsf{c}_1, \mathsf{c}'_1), \mathrm{cp}_2(\mathsf{c}_2, \mathsf{c}'_2))$ if both compositions are defined, and is undefined otherwise;

(c) $\mathcal{M}_1^\mathcal{A} = (\mathcal{A} \times \mathcal{C}_1, \mathcal{A} \times \mathcal{D}_1, \mathbf{cp}, \mathrm{ap}, \mathcal{A} \times \mathsf{I}_1)$, the *set-indexed model*, where $\mathrm{cp}((\mathsf{a}_1, \mathsf{c}_1), (\mathsf{a}_2, \mathsf{c}'_1)) = (\mathsf{a}_1, \mathrm{cp}_1(\mathsf{c}_1, \mathsf{c}'_1))$ if $\mathsf{a}_1 = \mathsf{a}_2$ and is undefined otherwise, for arbitrary $\mathsf{a}_1, \mathsf{a}_2 \in \mathcal{A}$.

As for CL, it is possible to derive additional properties for $\mathrm{CL}_\circ$, such as the distributivity rules for composition. The only property considered here is an interesting result for the somewhere modality.

Lemma 4.7 (Somewhere modality)**.** *In the presence of composition, the somewhere and everywhere modalities (Defn. 3.12) satisfy the* 4*-axiom of modal logic:*

$$\Diamond\Diamond P \Rightarrow \Diamond P \qquad \Box P \Rightarrow \Box\Box P$$

Proof. By the COMPOSITION rule, $\mathrm{True} \cdot (\mathrm{True} \cdot P) \Rightarrow (\mathrm{True} \circ \mathrm{True}) \cdot P \Rightarrow \mathrm{True} \cdot P$. Conversely, by contraposition, $\Box P = \neg\Diamond\neg P \Rightarrow \neg\Diamond\Diamond\neg P = \neg\Diamond\neg(\neg\Diamond\neg P) = \Box\Box P$.

As noted before, it is sometimes possible to express composition parametrically in the logic, using only application and its adjoints. However, this is not always the case. Consider, for example, the functional model $Fun_\mathbb{N}$ together with a propositional context variable k and an interpretation $\sigma(k) = \lambda n.\, n + 1$. Using a transitivity argument, it is easy to show that the only data formulæ expressible are true and false, and that therefore it is not possible to express $k \circ k = \lambda n.\, n + 2$. In the presence of a precise, omniplaceable formula, however, composition *can* be expressed. The

two requirements are quite strong, though Chapter 5 demonstrates how they can be weakened in the presence of quantification. Note also that the reverse implication, whereby composition implies the existence of a precise omniplaceable formula, clearly does not hold: for example, there are no omniplaceable formulæ in $PartFun_{\mathcal{D},I}$.

Lemma 4.8 (Expressibility of composition)**.** *If there exists a data formula P that is both precise (Defn. 3.24) and omniplaceable (Defn. 3.28), then context composition and its adjoints can be expressed parametrically as follows:*

$$K_1 \circ K_2 \dashv\vdash_{\mathcal{C}} P \rhd (K_1 \cdot (K_2 \cdot P))$$
$$K_1 \multimap K_2 \dashv\vdash_{\mathcal{C}} (K_1 \cdot P) \rhd (K_2 \cdot P)$$
$$K_1 \circ\!\!- K_2 \dashv\vdash_{\mathcal{C}} P \rhd (K_1 \lhd (K_2 \cdot P))$$

Proof.

(a) The left-to-right implications always hold, and follow immediately from the following entailments by applying the adjunction rules for $\rhd$ and $\lhd$.

$$(K_1 \circ K_2) \cdot P \vdash_{\mathcal{D}} K_1 \cdot (K_2 \cdot P)$$
$$(K_1 \multimap K_2) \cdot (K_1 \cdot P) \vdash_{\mathcal{D}} ((K_1 \multimap K_2) \circ K_1) \cdot P \vdash_{\mathcal{D}} K_2 \cdot P$$
$$K_1 \cdot ((K_1 \circ\!\!- K_2) \cdot P) \vdash_{\mathcal{D}} (K_1 \circ (K_1 \circ\!\!- K_2)) \cdot P \vdash_{\mathcal{D}} K_2 \cdot P$$

(b) The right-to-left implications depend on the preciseness and omniplaceability of P. By Lemmas 3.30 and 3.27, $P \rhd (K \cdot P) \vdash_{\mathcal{C}} P \blacktriangleright (K \cdot P) \vdash_{\mathcal{C}} K$ for arbitrary K. The proofs follow, making use of the results in part (a):

$$P \rhd (K_1 \cdot (K_2 \cdot P)) \vdash_{\mathcal{D}} P \rhd ((K_1 \circ K_2) \cdot P) \vdash_{\mathcal{D}} K_1 \circ K_2$$
$$\begin{aligned}((K_1 \cdot P) \rhd (K_2 \cdot P)) \circ K_1 &\vdash_{\mathcal{C}} P \rhd (((K_1 \cdot P) \rhd (K_2 \cdot P)) \cdot (K_1 \cdot P)) \\ &\vdash_{\mathcal{C}} P \rhd (K_2 \cdot P) \vdash_{\mathcal{C}} K_2\end{aligned}$$
$$\begin{aligned}K_1 \circ (P \rhd (K_1 \lhd (K_2 \cdot P))) &\vdash_{\mathcal{C}} P \rhd (K_1 \cdot ((P \rhd (K_1 \lhd (K_2 \cdot P))) \cdot P)) \\ &\vdash_{\mathcal{C}} P \rhd (K_1 \cdot (K_1 \lhd (K_2 \cdot P))) \\ &\vdash_{\mathcal{C}} P \rhd (K_2 \cdot P) \vdash_{\mathcal{C}} K_2\end{aligned}$$

4.2 Context Logic with Zero

In addition to context composition, many structured data models have an element, or set of elements, that correspond to empty data. For example, heap models have the empty heap, while tree models have the empty tree. This section describes adding a formula representing zero to the logic, and shows that this gives rise to interesting logical structure.

Definition 4.9 ($\mathrm{CL}_\emptyset$ formulæ)**.** The formulæ of Context Logic with Zero ($\mathrm{CL}_\emptyset$) consist of the same data and context assertions as CL (Defn. 3.1), with an additional assertion for zero.

$P ::= p$	propositional variables
$\mid K \cdot P \mid K \lhd P \mid \mathbf{0}$	**structural formulæ**
$\mid P \wedge P \mid P \Rightarrow P \mid \mathrm{false}$	logical formulæ
$K ::= k$	propositional variables
$\mid I \mid P \rhd P$	structural formulæ
$\mid K \wedge K \mid K \Rightarrow K \mid \mathrm{False}$	logical formulæ

Zero elements in structural data typically satisfy a number of common properties. Two such properties are used here to model zero. The first is ubiquity (Defn. 3.28): a zero element can always be placed in a context, and a data element can always be split into a context and a zero. The second is exactness (Defn. 3.20): placing a zero in a context can only ever give one result. Other properties may also hold (though most are not universal), but ubiquity and exactness are enough to provide some interesting derived properties and models.

Definition 4.10 ($\mathrm{CL}_\emptyset$ proof theory)**.** The proof theory of $\mathrm{CL}_\circ$ is the same as that of CL (Defn. 3.2), extended by the following rules specifying the ubiquity and exactness of zero (Defns. 3.28 and 3.20):

0 OMNIPRESENCE	0 OMNIPLACEABILITY	0 EXACTNESS
$\mathrm{true} \vdash_{\mathcal{D}} \mathrm{True} \cdot 0$	$\mathrm{True} \vdash_{\mathcal{C}} (0 \blacktriangleright \mathrm{true})$	$0 \blacktriangleright P \vdash_{\mathcal{C}} 0 \rhd P$

Models for $\mathrm{CL}_\emptyset$ extend CL models by adding a set of data elements corresponding to zero, together with an additional condition corresponding to exactness and ubiquity, which states that collectively applying the zero set to individual contexts defines a total surjective function from contexts to data. In general, applying a set of

data elements to a context defines a relation between the context and the results. Exactness, however, implies that the zero elements cannot be distinguished by applying a context, and that the relation must therefore be a partial function, relating each context to at most one data element. Similarly, omniplaceability implies totality, as at least one zero element can be applied to any context, while omnipresence implies surjectivity, as any data element can be split into a context and a member of zero.

Definition 4.11 ($\mathrm{CL}_\emptyset$ models)**.** A $\mathrm{CL}_\emptyset$ *model* $\mathcal{M} = (\mathcal{C}, \mathcal{D}, \mathrm{ap}, \mathsf{I}, \mathsf{0})$ is a CL model (Defn. 3.3) with the addition of a zero set $\mathsf{0} \subseteq \mathcal{D}$ for which the relation $p \subseteq \mathcal{C} \times \mathcal{D}$ defined by $(\mathsf{c}, \mathsf{d}) \in p \Leftrightarrow \exists \mathsf{o} \in \mathsf{0}.\ \mathrm{ap}(\mathsf{c}, \mathsf{o}) = \mathsf{d}$ is a total, surjective function.

Definition 4.12 ($\mathrm{CL}_\emptyset$ semantics)**.** Given a $\mathrm{CL}_\emptyset$ model $\mathcal{M}$ and an interpretation function $\sigma : (\mathcal{V}_\mathcal{D} \to \mathcal{P}(\mathcal{D})) \times (\mathcal{V}_\mathcal{C} \to \mathcal{P}(\mathcal{C}))$ mapping propositional variables to data and context sets, the forcing semantics is given by two satisfaction relations $\sigma, \mathcal{M}, \mathsf{d} \vDash_\mathcal{D} P$ and $\sigma, \mathcal{M}, \mathsf{c} \vDash_\mathcal{C} K$ defined as in Defn. 3.4 and extended as follows:

$$\sigma, \mathcal{M}, \mathsf{d} \vDash_\mathcal{D} 0 \Leftrightarrow \mathsf{d} \in \mathsf{0}$$

The soundness of $\mathrm{CL}_\emptyset$ follows by simple induction, using the semantic properties in Lemmas 3.22 and 3.29.

Like context composition, not all CL models have a zero. For example, the model *Step* in Example 3.7 clearly has no zero: there are no surjective functions from contexts (0 and 1) to data ($\mathbb{N}$). Similarly, the model $PartFun_{\mathcal{D},I}$ has no zero since there is no set of data elements that is omniplaceable: nothing can be placed in the everywhere undefined context. In Chapter 5 it will be shown that term models also do not have a zero. Below are examples of some CL models that do admit zeros.

Example 4.13 ($\mathrm{CL}_\emptyset$ models)**.** The following extensions to CL models from Example 3.7 are all models of $\mathrm{CL}_\emptyset$:

(a) $Emp = (\emptyset, \emptyset, \emptyset : \emptyset \to \emptyset, \emptyset, \emptyset)$, the *empty model*.

(b) $Fun_\mathcal{D} = (\mathcal{D} \to \mathcal{D}, \mathcal{D}, \mathrm{ap}, \{i\}, \{\mathsf{d}\})$, the *function model*, where the zero set contains an arbitrary data element $\mathsf{d} \in \mathcal{D}$.

(c) $Mon_{\mathcal{D},\circ,0} = (\mathcal{D}, \mathcal{D}, \circ, \{0\}, \{0\})$, the *monoid model*;

$PartMon_{\mathcal{D},\circ,0} = (\mathcal{D}, \mathcal{D}, \circ, \{0\}, \{0\})$, the *partial monoid model*.

Example 4.14 ($\mathrm{CL}_\emptyset$ constructions)**.** Given two $\mathrm{CL}_\emptyset$ models, $\mathcal{M}_1 = (\mathcal{C}_1, \mathcal{D}_1, \mathrm{ap}_1, \mathsf{I}_1, \mathsf{0}_1)$ and $\mathcal{M}_2 = (\mathcal{C}_2, \mathcal{D}_2, \mathrm{ap}_2, \mathsf{I}_2, \mathsf{0}_2)$, it is possible to extend all the constructions given in Example 3.8 as follows:

(a) $\mathcal{M}_1 + \mathcal{M}_2 = (\mathcal{C}_1 \cup \mathcal{C}_2, \mathcal{D}_1 \cup \mathcal{D}_2, \mathrm{ap}, \mathsf{I}_1 \cup \mathsf{I}_2, \mathsf{0}_1 \cup \mathsf{0}_2)$, the *union model*;

(b) $\mathcal{M}_1 \times \mathcal{M}_2 = (\mathcal{C}_1 \times \mathcal{C}_2, \mathcal{D}_1 \times \mathcal{D}_2, \mathrm{ap}, \mathsf{I}_1 \times \mathsf{I}_2, \mathsf{0}_1 \times \mathsf{0}_2)$, the *product model*;

(c) $\mathcal{M}_1^{\mathcal{A}} = (\mathcal{A} \times \mathcal{C}_1, \mathcal{A} \times \mathcal{D}_1, \mathrm{ap}, \mathcal{A} \times \mathsf{I}_1, \mathcal{A} \times \mathsf{0}_1)$, the *set-indexed model*.

Like context composition, zero is not generally expressible in CL. In the model $Fun_{\mathcal{D}}$, for example, any singleton data element can function as a zero, yet it is not possible to differentiate between individual data elements in the absence of an appropriate interpretation of propositional variables. However, for some structured data models, such as heaps and trees, zero can be expressed as the set of undecomposable elements: that is, those elements that can only be split into an identity context and themselves, which is expressible as $\neg(\neg I \cdot \mathrm{true})$. This property can be used in modelling zero, but the more general model considered here already displays some important properties.

4.3 Derived Connective on Data

Since the application of zero is a total surjective function, it can be regarded as a projection from contexts to data. Furthermore, it is also possible to use zero to embed data back into contexts, mapping a data element to all the contexts that project to it. This projection/embedding pair can be used to derive a useful connective on data and data formulæ, with interesting properties and an intuitive interpretation on structured data. The connective is called $*$, as for models that derive from Bunched Logic (BI) it coincides with the $*$ of BI.

A logical treatment of $*$ is considered first, followed by an interpretation of it as a connective on data. Projection and embedding with zero can easily be expressed logically: $K \cdot 0$ describes the projection of contexts satisfying K, while $0 \rhd P$ describes the embedding into contexts of data satisfying P. Using this, it is possible to define a connective on data formulæ P_1 and P_2, which embeds the data satisfying P_1 into contexts, and applies to these contexts the data satisfying P_2: namely, $P_1 * P_2 \triangleq (0 \rhd P_1) \cdot P_2$. It is also possible to derive the two right adjoints of this connective, $*\!\!-$ and $-\!\!*$.

The result is a BI-type structural connective on data formulæ. However, unlike in BI, this connective is neither necessarily commutative, nor associative. Furthermore, while 0 is a right identity for $*$, it is not necessarily a left-identity.

Definition 4.15 (Star connective)**.** The logical data connective $*$ and its two right adjoints $*\!\!\leftarrow$ and $\twoheadrightarrow$ are defined as follows:

$$P_1 * P_2 \triangleq (0 \rhd P_1) \cdot P_2$$
$$P_1 *\!\!\leftarrow P \triangleq (0 \rhd P_1) \lhd P$$
$$P_2 \twoheadrightarrow P \triangleq \neg(\neg(P_2 \rhd P) \cdot 0)$$

Lemma 4.16 (Star properties)**.**

(a) $\twoheadrightarrow$ *and* $*\!\!\leftarrow$ *are right adjoints of* $*$*: that is,*

$$\frac{P_1 * P_2 \vdash_{\mathcal{D}} P_3}{P_1 \vdash_{\mathcal{D}} P_2 \twoheadrightarrow P_3} \qquad \frac{P_1 * P_2 \vdash_{\mathcal{D}} P_3}{P_2 \vdash_{\mathcal{D}} P_1 *\!\!\leftarrow P_3}$$

(b) 0 *is a right-identity for* $*$*, though not necessarily a left-identity. In the case where* $0 \rhd 0 \vdash_{\mathcal{C}} I$*,* 0 *is also a left-identity.*

Proof.

(a) The adjunction properties follow from the derivations:

$$\frac{(0 \rhd P_1) \cdot P_2 \vdash_{\mathcal{D}} P_3}{P_2 \vdash_{\mathcal{D}} (0 \rhd P_1) \lhd P_3} \qquad \frac{\dfrac{\dfrac{\dfrac{\dfrac{(0 \rhd P_1) \cdot P_2 \vdash_{\mathcal{D}} P}{0 \rhd P_1 \vdash_{\mathcal{D}} P_2 \rhd P}\star}{0 \blacktriangleright P_1 \vdash_{\mathcal{D}} P_2 \rhd P}}{\neg(P_2 \rhd P) \vdash_{\mathcal{D}} 0 \rhd \neg P_1}}{\neg(P_2 \rhd P) \cdot 0 \vdash_{\mathcal{D}} \neg P_1}}{P_1 \vdash_{\mathcal{D}} \neg(\neg(P_2 \rhd P) \cdot 0)}$$

where the step marked $\star$ follows from the exactness (Defn. 3.20) and omniplaceability properties (Lemma 3.30) of 0.

(b) The right identity property $P * 0 = (0 \rhd P) \cdot 0 \dashv\vdash_{\mathcal{D}} 0$ follows from the omnipresence (Defn. 3.28a) and exactness (Defn. 3.20) of 0. In cases where $0 \rhd 0 \vdash_{\mathcal{C}} I$, then $0 \rhd 0 \dashv\vdash_{\mathcal{C}} I$ follows by applying the $\rhd$ adjunction rule to the $\cdot$ IDENTITY axiom, implying the left-identity property $0 * P = (0 \rhd 0) \cdot P \dashv\vdash_{\mathcal{D}} P$. For an

example where zero is not a left-identity, consider the functional model $Fun_{\mathcal{D}}$ with 0 satisfied by $\{\mathsf{d}\}$ (Example 4.13), and a propositional variable p with interpretation $\{\mathsf{d}'\}$ for $\mathsf{d} \neq \mathsf{d}'$. Then $0 \rhd 0$ is satisfied by any function that maps d to itself, and $0 * p = (0 \rhd 0) \cdot p$ is therefore logically equivalent to true.

In addition to the logical construction, there is also a counterpart to star defined as a connective on data in the underlying models. This is similar to the $*$ operator on heaps used in Separation Logic. The essence of the definition is the same as for the logical connective: $\mathsf{d}_1 * \mathsf{d}_2$ involves embedding d_1 and applying d_2. Whereas $*$ in Separation Logic is a partial function on heaps, here $*$ generally defines a relation on data, since the projection from contexts to data is not necessarily a bijection, and a data element can embed to many contexts.

Definition 4.17 (Star on data)**.** The relation $* \subseteq (\mathcal{D} \times \mathcal{D}) \times \mathcal{D}$ is defined as follows, writing for convenience $\mathsf{d}_1 * \mathsf{d}_2$ for $\{\mathsf{d} \mid ((\mathsf{d}_1, \mathsf{d}_2), \mathsf{d}) \in *\}$.

$$\mathsf{d}_1 * \mathsf{d}_2 = \{\mathsf{d} \mid \exists \mathsf{c} \in \mathcal{C}, \mathsf{o} \in \mathsf{0}.\, \mathsf{d}_1 = \mathrm{ap}(\mathsf{c}, \mathsf{o}) \wedge \mathsf{d} = \mathrm{ap}(\mathsf{c}, \mathsf{d}_2)\}$$

Lemma 4.18 (Star semantics)**.** *The semantics of the logical connectives $*$, $-\!\!*$ and $*\!\!-$ can be expressed in terms of the data connective $*$ as follows:*

$$\sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathcal{D}} P_1 * P_2 \;\Leftrightarrow \exists \mathsf{d}_1, \mathsf{d}_2 \in \mathcal{D}.\, (\mathsf{d} \in \mathsf{d}_1 * \mathsf{d}_2 \wedge \sigma, \mathcal{M}, \mathsf{d}_1 \vDash_{\mathcal{D}} P_1 \wedge \sigma, \mathcal{M}, \mathsf{d}_2 \vDash_{\mathcal{D}} P_2)$$
$$\sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathcal{D}} P_1 *\!\!- P_2 \Leftrightarrow \forall \mathsf{d}_1, \mathsf{d}_2 \in \mathcal{D}.\, (\mathsf{d}_2 \in \mathsf{d} * \mathsf{d}_1 \wedge \sigma, \mathcal{M}, \mathsf{d}_1 \vDash_{\mathcal{D}} P_1 \Rightarrow \sigma, \mathcal{M}, \mathsf{d}_2 \vDash_{\mathcal{D}} P_2)$$
$$\sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathcal{D}} P_1 -\!\!* P_2 \Leftrightarrow \forall \mathsf{d}_1, \mathsf{d}_2 \in \mathcal{D}.\, (\mathsf{d}_2 \in \mathsf{d}_1 * \mathsf{d} \wedge \sigma, \mathcal{M}, \mathsf{d}_1 \vDash_{\mathcal{D}} P_1 \Rightarrow \sigma, \mathcal{M}, \mathsf{d}_2 \vDash_{\mathcal{D}} P_2)$$

Proof. Follows directly from the definitions and derived semantics.

The following are examples of the star connective for the $\mathrm{CL}_{\emptyset}$ models given in Example 4.13. More interesting examples are considered for the sequence and tree models in Chapter 5.

Example 4.19 (Star connective)**.** Star has the following behaviour on the example models from Example 4.13:

(a) for *Emp*, the *empty model*, $*$ is the empty relation (unsurprisingly);

(b) for $Fun_{\mathcal{D}}$, the *function model* with zero $\{\mathsf{d}_0\}$, $\mathsf{d}_1 * \mathsf{d}_2$ equals $\{\mathsf{d}_1\}$ if $\mathsf{d}_2 = \mathsf{d}_0$ and all of $\mathcal{D}$ otherwise;

(c) for $Mon_{\mathcal{D},\circ,0}$ and $PartMon_{\mathcal{D},\circ,0}$, the *monoid* and *partial monoid models*, $\mathsf{d}_1 * \mathsf{d}_2 = \mathsf{d}_1 \circ \mathsf{d}_2$.

The monoid examples above shows that star is not necessarily commutative, while the tree model example in Chapter 5 will demonstrate that it is not even necessarily associative.

As mentioned before, the star connective here coincides with the $*$ connective of BI for appropriate models. As described in [POY04], boolean BI models can be expressed as partial commutative monoids $(\mathcal{D}, *, 0)$, which can be modelled in $\mathrm{CL}_\emptyset$ by $PartMon_{\mathcal{D},*,0} = (\mathcal{D}, \mathcal{D}, *, \{0\}, \{0\})$. For these models, $\mathrm{CL}_\emptyset$ turns out to be just as expressive as BI, with the derived star coinciding with BI's $*$ connective. This is demonstrated below, by showing a mutual embedding between models in $\mathrm{CL}_\emptyset$ and models in an appropriately-presented version of BI.

Definition 4.20 (BI formulæ)**.** The formulæ of Bunched Logic (BI) consist of a set of formulæ $P \in \mathcal{P}_{\mathrm{BI}}$ constructed from a set of propositional variables $\mathcal{V}_{\mathrm{BI}} = \{p, \dots\}$ and described by the grammar:

$$\begin{array}{lll} P ::= p & & \text{propositional variables} \\ \quad \mid P * P \mid P \mathrel{-\!\!*} P \mid 0 & & \text{structural formulæ} \\ \quad \mid P \wedge P \mid P \Rightarrow P \mid \mathrm{false} & & \text{logical formulæ} \end{array}$$

Definition 4.21 (BI models and forcing semantics)**.** A BI *model* $\mathcal{M} = (\mathcal{D}, *, \mathsf{0})$ consists of a partial commutative monoid $\mathcal{D}$ with monoidal operator $*$ and identity $\mathsf{0}$. Given a BI model $\mathcal{M}$ and an interpretation function $\sigma : (\mathcal{V}_{\mathrm{BI}} \to \mathcal{P}(\mathcal{D}))$ mapping propositional variables to sets of data, the forcing semantics is given by a satisfaction relation $\sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathrm{BI}} P$ defined inductively on the structure of data formulæ:

$$\begin{array}{lll} \sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathrm{BI}} p & \Leftrightarrow \mathsf{d} \in \sigma(p) \\ \sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathrm{BI}} P_1 * P_2 & \Leftrightarrow \exists \mathsf{d}_1, \mathsf{d}_2 \in \mathcal{D}. (\mathsf{d} = \mathsf{d}_1 * \mathsf{d}_2 \wedge \sigma, \mathcal{M}, \mathsf{d}_1 \vDash_{\mathrm{BI}} P_1 \wedge \sigma, \mathcal{M}, \mathsf{d}_2 \vDash_{\mathrm{BI}} P_2) \\ \sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathrm{BI}} P_1 \mathrel{-\!\!*} P_2 & \Leftrightarrow \forall \mathsf{d}' \in \mathcal{D}. ((\mathsf{d}' * \mathsf{d}{\downarrow} \wedge \sigma, \mathcal{M}, \mathsf{d}' \vDash_{\mathrm{BI}} P_1) \Rightarrow \sigma, \mathcal{M}, \mathrm{ap}(\mathsf{c}, \mathsf{d}) \vDash_{\mathrm{BI}} P_2) \\ \sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathrm{BI}} 0 & \Leftrightarrow \mathsf{d} = 0 \\ \sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathrm{BI}} P_1 \wedge P_2 & \Leftrightarrow \sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathrm{BI}} P_1 \wedge \sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathrm{BI}} P_2 \\ \sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathrm{BI}} P_1 \Rightarrow P_2 & \Leftrightarrow \sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathrm{BI}} P_1 \Rightarrow \sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathrm{BI}} P_2 \\ \sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathrm{BI}} \mathrm{false} & \text{never} \end{array}$$

Theorem 4.22 (BI Embedding Result)**.** *There exists a translation* $[\![-]\!]_{\mathrm{CL}_\emptyset}$ *of* BI *formulæ into* $\mathrm{CL}_\emptyset$ *data formulæ, and of* BI *interpretation functions into* $\mathrm{CL}_\emptyset$ *interpretation functions, such that any formula* P *on a* BI *model* $\mathcal{M}_{\mathrm{BI}} = (\mathcal{D}, *, 0)$ *agrees with the formula* $[\![P]\!]_{\mathrm{CL}_\emptyset}$ *on the corresponding* $\mathrm{CL}_\emptyset$ *model* $\mathcal{M}_{\mathrm{CL}_\emptyset} = (\mathcal{D}, \mathcal{D}, *, \{0\}, \{0\})$*:*

$$\forall \sigma, \mathsf{d}.\, (\sigma, \mathsf{d}, \mathcal{M}_{\mathrm{BI}} \vDash_{\mathrm{BI}} P \Leftrightarrow [\![\sigma]\!]_{\mathrm{CL}_\emptyset}, \mathsf{d}, \mathcal{M}_{\mathrm{CL}_\emptyset} \vDash_{\mathcal{D}} [\![P]\!]_{\mathrm{CL}_\emptyset})$$

Conversely, there exists a reverse translation $[\![-]\!]_{\mathrm{BI}}$ *of* $\mathrm{CL}_\emptyset$ *context and data formulæ into* BI *formulæ, and of* $\mathrm{CL}_\emptyset$ *interpretation functions into* BI *interpretation functions, such that any formula* P *or* K *on a* $\mathrm{CL}_\emptyset$ *model* $\mathcal{M}_{\mathrm{CL}_\emptyset} = PartMon_{\mathcal{D},*,0}$*, where* $*$ *is commutative, agrees with the formula* $[\![P]\!]_{\mathrm{BI}}$ *on the corresponding* BI *model* $\mathcal{M}_{\mathrm{BI}} = (\mathcal{D}, *, 0)$*:*

$$\forall \sigma, \mathsf{d}.\, (\sigma, \mathsf{d}, \mathcal{M}_{\mathrm{CL}_\emptyset} \vDash_{\mathcal{D}} P \Leftrightarrow [\![\sigma]\!]_{\mathrm{BI}}, \mathsf{d}, \mathcal{M}_{\mathrm{BI}} \vdash_{\mathrm{BI}} [\![P]\!]_{\mathrm{BI}})$$
$$\forall \sigma, \mathsf{d}.\, (\sigma, \mathsf{d}, \mathcal{M}_{\mathrm{CL}_\emptyset} \vDash_{\mathcal{C}} K \Leftrightarrow [\![\sigma]\!]_{\mathrm{BI}}, \mathsf{d}, \mathcal{M}_{\mathrm{BI}} \vdash_{\mathrm{BI}} [\![K]\!]_{\mathrm{BI}})$$

Proof. Let $\pi_{\mathrm{CL}_\emptyset} : \mathcal{V}_{\mathrm{BI}} \to \mathcal{V}_P$ be a bijective function from propositional variables in BI to propositional data variables in $\mathrm{CL}_\emptyset$, and let $\pi_{\mathrm{BI}} : (\mathcal{V}_P \uplus \mathcal{V}_K) \to \mathcal{V}_{\mathrm{BI}}$ be a bijective function from propositional context and data variables in $\mathrm{CL}_\emptyset$ to propositional variables in BI. Then the translations of the interpretation functions are given by:

$$[\![\sigma]\!]_{\mathrm{CL}_\emptyset}(p) = \sigma({\pi_{\mathrm{CL}_\emptyset}}^{-1}(p)) \qquad [\![\sigma]\!]_{\mathrm{BI}}(p) = \sigma({\pi_{\mathrm{BI}}}^{-1}(p))$$
$$[\![\sigma]\!]_{\mathrm{CL}_\emptyset}(k) = \emptyset$$

The formula translations, meanwhile, map the classical connectives to their direct correspondents, while the structural formulæ and propositional variables are translated as follows, with the $[\![-]\!]_{\mathrm{CL}_\emptyset}$ translation using the derived star connectives from Defn. 4.15:

$$[\![p]\!]_{\mathrm{CL}_\emptyset} \triangleq \pi_{\mathrm{CL}_\emptyset}(p) \qquad [\![P * P']\!]_{\mathrm{CL}_\emptyset} \triangleq [\![P]\!]_{\mathrm{CL}_\emptyset} * [\![P']\!]_{\mathrm{CL}_\emptyset}$$
$$[\![0]\!]_{\mathrm{CL}_\emptyset} \triangleq 0 \qquad [\![P \mathrel{-\!\!*} P']\!]_{\mathrm{CL}_\emptyset} \triangleq [\![P]\!]_{\mathrm{CL}_\emptyset} \mathrel{-\!\!*} [\![P']\!]_{\mathrm{CL}_\emptyset}$$

$$[\![p]\!]_{\mathrm{BI}} \triangleq \pi_{\mathrm{BI}}(p) \qquad [\![K \cdot P]\!]_{\mathrm{BI}} \triangleq [\![K]\!]_{\mathrm{BI}} * [\![P]\!]_{\mathrm{BI}}$$
$$[\![k]\!]_{\mathrm{BI}} \triangleq \pi_{\mathrm{BI}}(k) \qquad [\![K \lhd P]\!]_{\mathrm{BI}} \triangleq [\![K]\!]_{\mathrm{BI}} \mathrel{-\!\!*} [\![P]\!]_{\mathrm{BI}}$$
$$[\![I]\!]_{\mathrm{BI}} \triangleq 0 \qquad [\![P \rhd P']\!]_{\mathrm{BI}} \triangleq [\![P]\!]_{\mathrm{BI}} \mathrel{-\!\!*} [\![P']\!]_{\mathrm{BI}}$$
$$[\![0]\!]_{\mathrm{BI}} \triangleq 0$$

The derived semantics, given in Defns. 3.4 and 4.21 and Lemma 4.18, all match up as required.

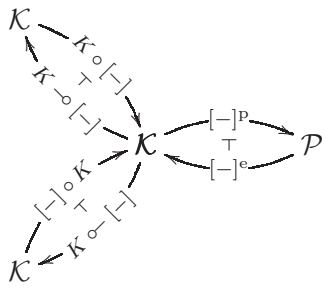
4.4 Embeddings and Projections

Apart from helping define a connective on data, the projection and embedding provided by zero also suggest an alternate way to represent Context Logic, which is explored in this section. The idea is to replace context application and its adjoints with this projection/embedding pair, and express the interaction between contexts and data using only this and context composition. The result is a seemingly simpler logic, which contains just a monoidal context structure based around composition, and a simple mapping from contexts to data. In fact, Thms. 4.34 and 4.36 show that the resulting logic is equivalent to standard Context Logic with zero and composition.

Definition 4.23 ($\mathrm{CL_p}$ formulæ)**.** The formulæ of Projective Context Logic ($\mathrm{CL_p}$) consist of data and context assertions from $\mathrm{CL}_\circ$ (Defn. 4.1), but with a projection operator on contexts K^{p} and an embedding operator on data P^{e} instead of context application and its adjoints:

$P ::= p$	propositional variables
$\mid \boldsymbol{K^{\mathrm{p}}}$	**structural formulæ**
$\mid P \wedge P \mid P \Rightarrow P \mid \mathrm{false}$	logical formulæ
$K ::= k$	propositional variables
$\mid I \mid \boldsymbol{P^{\mathrm{e}}} \mid K \circ K \mid K \multimap K \mid K \circ\!\!- K$	**structural formulæ**
$\mid K \wedge K \mid K \Rightarrow K \mid \mathrm{False}$	logical formulæ

The two additional formulæ, which replace context application application and its adjoints, represent projection and embedding. The projection formula K^{p} projects contexts satisfying K to data, and can be viewed as a simple case of application where a zero element is placed in the context hole. Similarly, the adjoint embedding P^{e} embeds data satisfying P back to contexts, and can be viewed as $(0 \rhd P)$, a simple case of the adjunct of application. The adjunctive structure of $\mathrm{CL_p}$ can therefore be illustrated by the following diagram:



Definition 4.24 ($\mathrm{CL_p}$ proof theory)**.** The proof theory for $\mathrm{CL_p}$ consists of the propositional logic proof rules in Defn. 3.2, the composition axioms in Defn. 4.2 (with the exception of the COMPOSITION axiom, which linked composition to context application), and:

$$\frac{K^{\mathrm{p}} \vdash_{\mathcal{D}} P}{K \vdash_{\mathcal{C}} P^{\mathrm{e}}}\ {}^{\mathrm{p},\,\mathrm{e}} \qquad \text{FUNCTIONALITY} \quad (\neg P)^{\mathrm{e}} \vdash_{\mathcal{C}} \neg(P^{\mathrm{e}}) \qquad \text{SURJECTIVITY} \quad P \vdash_{\mathcal{D}} P^{\mathrm{ep}}$$

$$\text{COMPOSITION} \quad (K_1 \circ K_2)^{\mathrm{p}} \dashv\vdash_{\mathcal{D}} (K_1 \circ {K_2}^{\mathrm{pe}})^{\mathrm{p}}$$

The additional rules define the behaviour of $[-]^{\mathrm{p}}$ and $[-]^{\mathrm{e}}$ as a projection/embedding pair. The first rule describes the adjunction between the projection and the embedding, allowing us to view $[-]^{\mathrm{p}}$ as a relation that is total on its first argument and $[-]^{\mathrm{e}}$ as its inverse. The second rule specifies that $[-]^{\mathrm{p}}$ is a function, since its preimages on disjoint sets must be disjoint. The third rule specifies that it is surjective, since any data element satisfying P must be in the range of $[-]^{\mathrm{p}}$. Finally, the last rule relates projection to composition, and is based on the intuition that projection only affects the deep part of a context, corresponding to its hole.

As in $\mathrm{CL}_\circ$, the models of $\mathrm{CL_p}$ consist of a context set, a data set and a partial composition operator, with the addition of a total surjective projection function satisfying an appropriate composition condition.

Definition 4.25 ($\mathrm{CL_p}$ models)**.** A *model* $\mathcal{M} = (\mathcal{C}, \mathcal{D}, \mathrm{cp}, \mathsf{I}, \mathrm{proj})$ of $\mathrm{CL_p}$ consists of a context set $\mathcal{C}$, a data set $\mathcal{D}$, a partial composition operator $\mathrm{cp} : \mathcal{C} \times \mathcal{C} \rightharpoonup \mathcal{C}$ that forms a partial monoid on context sets with identity $\mathsf{I} \subseteq \mathcal{C}$ (as in Defn. 4.3), and a total surjective projection function $\mathrm{proj} : \mathcal{C} \to \mathcal{D}$ satisfying the following composition condition, which states that projection only affects the 'inner' part of a context:

$$\forall \mathsf{c}, \mathsf{c}_1, \mathsf{c}_2 \in \mathcal{C}. \left(\begin{array}{c} (\mathrm{cp}(\mathsf{c}, \mathsf{c}_1)\downarrow \wedge \mathrm{proj}(\mathsf{c}_1) = \mathrm{proj}(\mathsf{c}_2)) \\ \Rightarrow (\mathrm{cp}(\mathsf{c}, \mathsf{c}_2)\downarrow \wedge \mathrm{proj}(\mathrm{cp}(\mathsf{c}, \mathsf{c}_1)) = \mathrm{proj}(\mathrm{cp}(\mathsf{c}, \mathsf{c}_2)) \end{array} \right)$$

Definition 4.26 ($\mathrm{CL_p}$ forcing semantics)**.** Given a model $\mathcal{M}$ of $\mathrm{CL_p}$ and an interpretation function $\sigma : (\mathcal{V}_{\mathcal{D}} \to \mathcal{P}(\mathcal{D})) \times (\mathcal{V}_{\mathcal{C}} \to \mathcal{P}(\mathcal{C}))$ mapping propositional variables to data and context sets, the forcing semantics is given by two satisfaction relations $\sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathcal{D}} P$ and $\sigma, \mathcal{M}, \mathsf{c} \vDash_{\mathcal{C}} K$ defined as in Defn. 4.4 where relevant and extended as follows:

$$\sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathcal{D}} K^{\mathrm{p}} \Leftrightarrow \exists \mathsf{c} \in \mathcal{C}.(\mathsf{d} = \mathrm{proj}(\mathsf{c}) \wedge \sigma, \mathcal{M}, \mathsf{c} \vDash_{\mathcal{C}} K)$$
$$\sigma, \mathcal{M}, \mathsf{c} \vDash_{\mathcal{C}} P^{\mathrm{e}} \Leftrightarrow \exists \mathsf{d} \in \mathcal{D}.(\mathsf{d} = \mathrm{proj}(\mathsf{c}) \wedge \sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathcal{D}} P)$$

Example 4.27 ($\mathrm{CL_p}$ models)**.** The following are all models of $\mathrm{CL_p}$:

(a) $Identity_{\mathcal{D},\circ,0} = (\mathcal{D}, \mathcal{D}, \circ, \{0\}, i)$, the *identity model*, where $(\mathcal{D}, \circ, 0)$ is an arbitrary partial monoid, and projection is the identity function.

(b) $Parity = (\mathbb{Z}, \{0,1\}, +, \{0\}, \mathrm{pty})$, the *parity model*, where projection returns 0 for even contexts and 1 for odd contexts, and the composition condition holds since $\mathrm{pty}(z_1) = \mathrm{pty}(z_2) \Rightarrow \mathrm{pty}(z + z_1) = \mathrm{pty}(z + z_2)$.

(c) $Length = (\Sigma^*, \mathbb{N}, \cdot, \epsilon, \mathrm{len})$, the *length model*, where the context monoid consists of sequences under concatenation, projection is the length function, and the composition condition holds since $\mathrm{len}(s_1) = \mathrm{len}(s_2) \Rightarrow \mathrm{len}(s \cdot s_1) = \mathrm{len}(s \cdot s_2)$.

(d) $Elements = (\Sigma^*, \mathcal{P}(\Sigma), \cdot, \epsilon, \mathrm{elm})$, the *elements model*, where projection maps sequences to their underlying element sets. A variant model uses linear sequences, where sequences have unique elements and concatenation is partial.

At the end of this section, $\mathrm{CL_p}$ is shown to be exactly equivalent to Context Logic with zero and composition. First, however, we consider some useful basic properties of $\mathrm{CL_p}$, including a self-duality property for embedding, and define the dual operator of projection, which forms a right adjoint to embedding.

Lemma 4.28 (Basic properties)**.** *The following basic properties hold:*

(a) functoriality of $[-]^{\mathrm{e}}$ *and* $[-]^{\mathrm{p}}$*:*

$$\frac{P_1 \vdash_{\mathcal{D}} P_2}{{P_1}^{\mathrm{e}} \vdash_{\mathcal{C}} {P_2}^{\mathrm{e}}} \qquad \frac{K_1 \vdash_{\mathcal{C}} K_2}{{K_1}^{\mathrm{p}} \vdash_{\mathcal{D}} {K_2}^{\mathrm{p}}}$$

(b) $\vee$*-distributivity and* $\wedge$*-semidistributivity for* $[-]^{\mathrm{e}}$*:*

$$(P_1 \vee P_2)^{\mathrm{e}} \dashv\vdash_{\mathcal{C}} {P_1}^{\mathrm{e}} \vee {P_2}^{\mathrm{e}}$$
$$(P_1 \wedge P_2)^{\mathrm{e}} \vdash_{\mathcal{C}} {P_1}^{\mathrm{e}} \wedge {P_2}^{\mathrm{e}}$$

Proof. Both results follow directly from the adjunction, just as in Lemma 3.10.

Lemma 4.29 (Self-duality of embedding)**.** *The embedding* $[-]^{\mathrm{e}}$ *exhibits the following self-duality property:*

$$P^{\mathrm{e}} \dashv\vdash_{\mathcal{C}} \neg((\neg P)^{\mathrm{e}})$$

Proof. $P^{\mathrm{e}} \vdash_{\mathcal{C}} \neg((\neg P)^{\mathrm{e}})$ follows directly from the FUNCTIONALITY axiom, while the reverse implication is proved by

$$\dfrac{\dfrac{\dfrac{\mathrm{True}^{\mathrm{p}} \vdash_{\mathcal{D}} \mathrm{true}}{\mathrm{True} \vdash_{\mathcal{C}} \mathrm{true}^{\mathrm{e}}}}{\mathrm{True} \vdash_{\mathcal{C}} (P \vee \neg P)^{\mathrm{e}} \quad (P \vee \neg P)^{\mathrm{e}} \vdash_{\mathcal{C}} P^{\mathrm{e}} \vee (\neg P)^{\mathrm{e}}}}{\mathrm{True} \vdash_{\mathcal{C}} P^{\mathrm{e}} \vee (\neg P)^{\mathrm{e}}}}{\neg((\neg P)^{\mathrm{e}}) \vdash_{\mathcal{C}} P^{\mathrm{e}}}$$

Definition 4.30 (Projection dual)**.** The projection dual K^{f} is defined by

$$K^{\mathrm{f}} \triangleq \neg((\neg K)^{\mathrm{p}})$$

and has the following derived semantics:

$$\sigma, \mathcal{M}, \mathsf{d} \vDash_{\mathcal{D}} K^{\mathrm{f}} \Leftrightarrow \forall \mathsf{c} \in \mathcal{C}.(\mathsf{d} = \mathrm{proj}(\mathsf{c}) \Rightarrow \sigma, \mathcal{M}, \mathsf{c} \vDash_{\mathcal{C}} K)$$

The projection dual $[-]^{\mathrm{f}}$ can be viewed as a type of filtering operation (hence the notation): while $[-]^{\mathrm{p}}$ is satisfied by any data that comes from projecting a context satisfying K, $[-]^{\mathrm{f}}$ is satisfied by data that can *only* come from projecting contexts satisfying K. For example, in the *Parity* model in Example 4.27, a formula K^{f} is satisfied by 0 only if K is satisfied by all the even numbers, and by 1 only if K is satisfied by all the odd numbers. This filter operation is in fact the right adjoint of embedding.

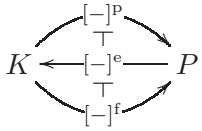
Lemma 4.31 (Filter/embedding adjunction)**.** *The projection dual is the right adjoint of embedding:*

$$\dfrac{P^{\mathrm{e}} \vdash_{\mathcal{C}} K}{P \vdash_{\mathcal{D}} K^{\mathrm{f}}}$$

Proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{P^{\mathrm{e}} \vdash_{\mathcal{C}} K}{\neg K \vdash_{\mathcal{C}} \neg(P^{\mathrm{e}})}}{\neg K \vdash_{\mathcal{C}} (\neg P)^{\mathrm{e}}}}{(\neg K)^{\mathrm{p}} \vdash_{\mathcal{D}} \neg P}}{P \vdash_{\mathcal{D}} \neg((\neg P)^{\mathrm{p}})}$$

The presence of this right adjoint of embedding leaves us with the following adjunctive diagram:



It is now possible to prove an exact equivalence between $\mathrm{CL_p}$ and Context Logic with both zero and context composition ($\mathrm{CL}_{\emptyset,\circ}$). This is done by giving translations between $\mathrm{CL}_{\emptyset,\circ}$ and $\mathrm{CL_p}$ formulæ and models, and showing that the proof theories and satisfaction relations are equivalent.

Definition 4.32 ($\mathrm{CL}_{\emptyset,\circ}$)**.** Context Logic with Zero and Composition ($\mathrm{CL}_{\emptyset,\circ}$) is made up of the combined assertions and proof rules of $\mathrm{CL}_\circ$ (Defns. 4.1 and 4.2) and $\mathrm{CL}_\emptyset$ (Defns. 4.9 and 4.10). Models consist of a combined model $(\mathcal{C}, \mathcal{D}, \mathrm{cp}, \mathrm{ap}, \mathsf{I}, 0)$, where $(\mathcal{C}, \mathcal{D}, \mathrm{cp}, \mathrm{ap}, \mathsf{I})$ is a model of $\mathrm{CL}_\circ$ and $(\mathcal{C}, \mathcal{D}, \mathrm{ap}, \mathsf{I}, 0)$ is a model of $\mathrm{CL}_\emptyset$. The forcing semantics is given by Defns. 4.4 and 4.12.

Definition 4.33 ($\mathrm{CL}_{\emptyset,\circ}/\mathrm{CL_p}$ formula translation)**.** The translation $[\![-]\!]_p$ of $\mathrm{CL}_{\emptyset,\circ}$ formulæ into $\mathrm{CL_p}$ formulæ, and the reverse translation $[\![-]\!]_{\emptyset,\circ}$ of $\mathrm{CL_p}$ formulæ into $\mathrm{CL}_{\emptyset,\circ}$ formulæ, are defined by induction on the structure of formulæ, mapping the shared connectives of $\mathrm{CL}_{\emptyset,\circ}$ and $\mathrm{CL_p}$ to themselves and the other connectives as follows:

$$
\begin{aligned}
[\![0]\!]_p &\triangleq I^{\mathrm{p}} & [\![K^{\mathrm{p}}]\!]_{\emptyset,\circ} &\triangleq [\![K]\!]_{\emptyset,\circ} \cdot 0 \\
[\![K \cdot P]\!]_p &\triangleq ([\![K]\!]_p \circ [\![P]\!]_p{}^{\mathrm{e}})^{\mathrm{p}} & [\![P^{\mathrm{e}}]\!]_{\emptyset,\circ} &\triangleq 0 \rhd [\![P]\!]_{\emptyset,\circ} \\
[\![K \lhd P]\!]_p &\triangleq ([\![K]\!]_p \circ\!\!- [\![P]\!]_p{}^{\mathrm{e}})^{\mathrm{f}} \\
[\![P_1 \rhd P_2]\!]_p &\triangleq [\![P_1]\!]_p{}^{\mathrm{e}} \mathrel{-\!\!\circ} [\![P_2]\!]_p{}^{\mathrm{e}}
\end{aligned}
$$

The formula translations translate formulæ in one logic to equivalent formulæ in the other logic. The translation $[\![-]\!]_{\emptyset,\circ}$ from $\mathrm{CL_p}$ to $\mathrm{CL}_{\emptyset,\circ}$ formalises the intuition that projection corresponds to the application of zero and embedding corresponds to its adjoint. The reverse translation $[\![-]\!]_p$, meanwhile, defines zero, context application and its adjoints using projection and embedding. The zero formula is defined as the projection of I, while context application $K \cdot P$ corresponds to composing K with the context embedding of P, and projecting the result back to data. Thus the projection/embedding pair is used to transform an interaction between contexts and data into one involving just contexts and expressible using context composition. The translations of the two adjoints are similar to that of application. $P_1 \rhd P_2$ is expressed using $-\!\!\circ$ by embedding both P_1 and P_2 into contexts. The translation of $K \lhd P$ using

$\circ\!\!-$ is more subtle: in mapping the interaction back to data, it is necessary to use the filter operation $[-]^{\mathrm{f}}$ rather than $[-]^{\mathrm{p}}$, since the adjoint statement on contexts should be satisfied by all possible embeddings of the data statement, not just some.

Theorem 4.34 ($\mathrm{CL}_{\emptyset,\circ}/\mathrm{CL}_{\mathrm{p}}$ proof theory equivalence)**.** *A formula is derivable in* $\mathrm{CL}_{\emptyset,\circ}$ *or* CL_{p} *if and only if the translated formula given by Defn. 4.33 is also derivable: that is,*

$$\mathrm{True} \vdash_{\mathcal{C}} K \Leftrightarrow \mathrm{True} \vdash_{\mathcal{C}} [\![K]\!]_p$$
$$\mathrm{true} \vdash_{\mathcal{D}} P \Leftrightarrow \mathrm{true} \vdash_{\mathcal{D}} [\![P]\!]_p$$

$$\mathrm{True} \vdash_{\mathcal{C}} K' \Leftrightarrow \mathrm{True} \vdash_{\mathcal{C}} [\![K']\!]_{\emptyset,\circ}$$
$$\mathrm{true} \vdash_{\mathcal{D}} P' \Leftrightarrow \mathrm{true} \vdash_{\mathcal{D}} [\![P']\!]_{\emptyset,\circ}$$

where K *and* P *are arbitrary formulæ in* $\mathrm{CL}_{\emptyset,\circ}$*, and* K' *and* P' *are arbitrary formulæ in* CL_{p}*.*

Proof. For the left-to-right implications, it is sufficient to show how to derive the missing proof rules in $\mathrm{CL}_{\emptyset,\circ}$ and CL_{p}, using the appropriate formula translations from Defn. 4.33 in place of the missing connectives. These derivations are given in Figure 4.1, with the proofs for $\mathrm{CL}_{\emptyset,\circ}$ relying extensively on the properties of zero described in Chapter 3.

For the right-to-left implication, it is enough to show the following weak bijectionality result for the formula translations, since the implication then follows by the left-to-right implication proven above.

$$[\![[\![K]\!]_p]\!]_{\emptyset,\circ} \vdash_{\mathcal{C}} K \quad [\![[\![K']\!]_{\emptyset,\circ}]\!]_p \vdash_{\mathcal{C}} K'$$
$$[\![[\![P]\!]_p]\!]_{\emptyset,\circ} \vdash_{\mathcal{D}} P \quad [\![[\![P']\!]_{\emptyset,\circ}]\!]_p \vdash_{\mathcal{D}} P'$$

Writing $[\![-]\!]$ for $[\![[\![-]\!]_p]\!]_{\emptyset,\circ}$, the left-hand results follow from:

$$\begin{aligned}
[\![0]\!] &= I \cdot 0 \vdash_{\mathcal{D}} 0 \\
[\![K \cdot P]\!] &= ([\![K]\!] \circ (0 \rhd [\![P]\!])) \cdot 0 \vdash_{\mathcal{D}} [\![K]\!] \cdot ((0 \rhd [\![P]\!]) \cdot 0) \vdash_{\mathcal{D}} [\![K]\!] \cdot [\![P]\!] \\
[\![K \lhd P]\!] &= \neg(\neg([\![K]\!] \circ\!\!- (0 \rhd [\![P]\!])) \cdot 0) \vdash_{\mathcal{D}} ([\![K]\!] \circ\!\!- (0 \rhd [\![P]\!])) \cdot 0 \vdash_{\mathcal{D}} [\![K]\!] \lhd [\![P]\!] \\
[\![P_1 \rhd P_2]\!] &= (0 \rhd [\![P_1]\!]) \multimap (0 \rhd [\![P_2]\!]) \vdash_{\mathcal{D}} [\![P_1]\!] \rhd [\![P_2]\!]
\end{aligned}$$

where the three entailments in the last two derivations follow from the derived prop-

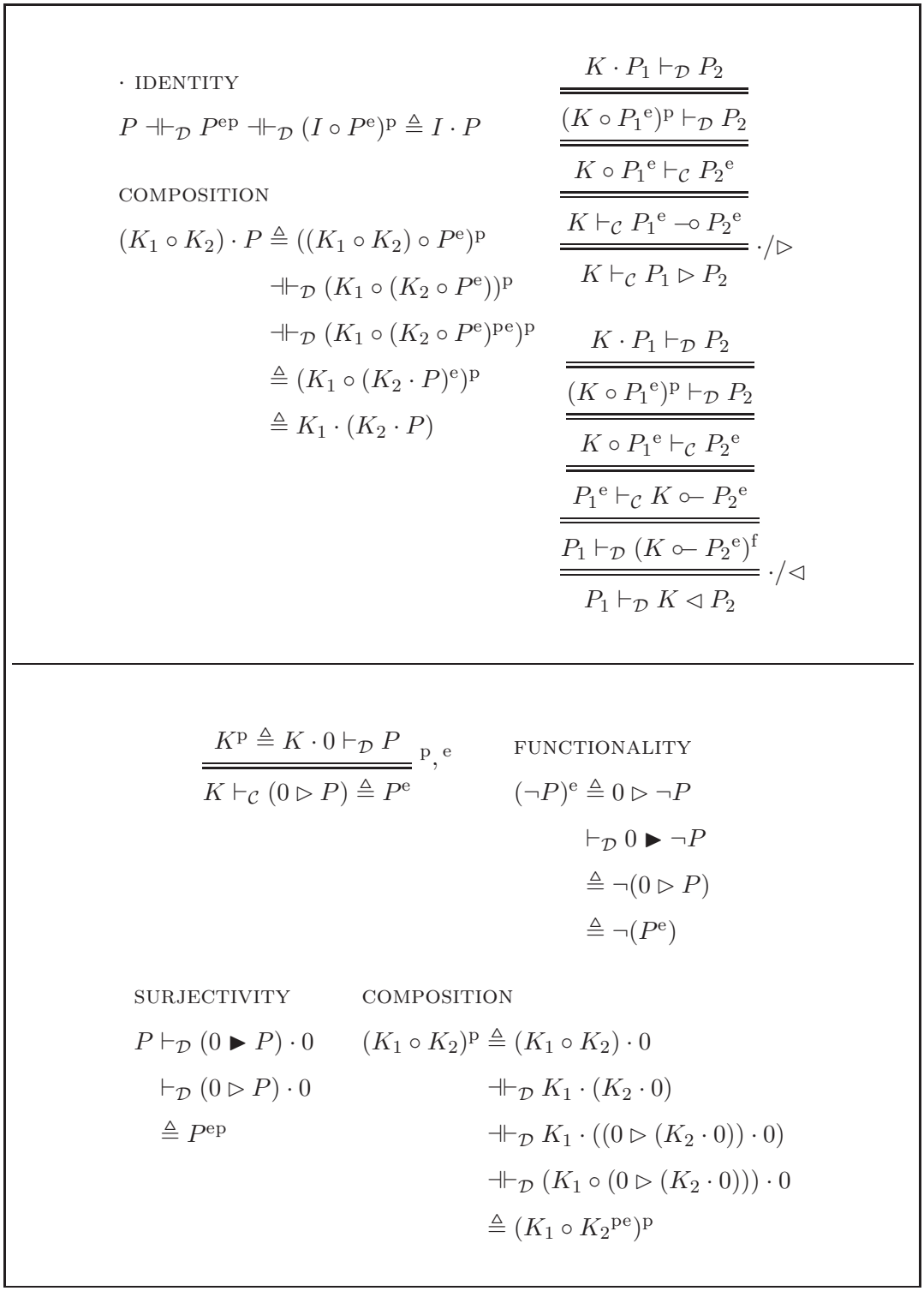


Figure 4.1: Proof theory derivations in CL_{p} (top) and $\mathrm{CL}_{\emptyset,\circ}$ (bottom).

erty $\neg((\neg K') \cdot 0) \vdash_{\mathcal{D}} K' \cdot 0$, which holds for arbitrary K', as well as:

$$\begin{aligned} K \cdot ((K \multimap (0 \rhd P)) \cdot 0) &\vdash_{\mathcal{D}} (K \circ (K \multimap (0 \rhd P))) \cdot 0 \\ &\vdash_{\mathcal{D}} (0 \rhd P) \cdot 0 \\ &\vdash_{\mathcal{D}} P \end{aligned}$$

$$\begin{aligned} ((0 \rhd [\![P_1]\!]) \multimap (0 \rhd [\![P_2]\!])) \cdot P_1 &\vdash_{\mathcal{D}} ((0 \rhd [\![P_1]\!]) \multimap (0 \rhd [\![P_2]\!])) \cdot ((0 \rhd P_1) \cdot 0) \\ &\vdash_{\mathcal{D}} (((0 \rhd [\![P_1]\!]) \multimap (0 \rhd [\![P_2]\!])) \circ (0 \rhd P_1)) \cdot 0 \\ &\vdash_{\mathcal{D}} (0 \rhd [\![P_2]\!]) \cdot 0 \\ &\vdash_{\mathcal{D}} [\![P_2]\!] \end{aligned}$$

Similarly, writing $[\![-]\!]'$ for $[\![[\![-]\!]_{\emptyset,\circ}]\!]_p$, the right-hand results follow from:

$$\begin{aligned} [\![K^{\mathrm{p}}]\!]' &= ([\![K]\!]' \circ I^{\mathrm{pe}})^{\mathrm{p}} \vdash_{\mathcal{D}} [\![K]\!]'^{\mathrm{p}} \\ [\![P^{\mathrm{e}}]\!]' &= I^{\mathrm{pe}} \multimap [\![P]\!]'^{\mathrm{e}} \vdash_{\mathcal{D}} I \multimap [\![P]\!]'^{\mathrm{e}} \vdash_{\mathcal{D}} [\![P]\!]'^{\mathrm{e}} \end{aligned}$$

where the final entailment follows from $I \vdash_{\mathcal{D}} I^{\mathrm{pe}}$ and the covariance of $\multimap$.

In addition to the formula translations, it is also possible to give translations between $\mathrm{CL}_{\emptyset,\circ}$ and CL_{p} models, and show that the satisfaction relations for these are the equivalent.

Definition 4.35 ($\mathrm{CL}_{\emptyset,\circ}/\mathrm{CL}_{\mathrm{p}}$ model translation)**.** The translation $[\![-]\!]_p$ of $\mathrm{CL}_{\emptyset,\circ}$ models into CL_{p} models, and the reverse translation $[\![-]\!]_{\emptyset,\circ}$ of CL_{p} models into $\mathrm{CL}_{\emptyset,\circ}$ models, are defined as follows.

- $[\![(\mathcal{C}, \mathcal{D}, \mathrm{cp}, \mathrm{ap}, \mathsf{I}, \mathsf{0})]\!]_p = (\mathcal{C}, \mathcal{D}, \mathrm{cp}, \mathsf{I}, \mathrm{proj})$, where $\mathrm{proj}(\mathsf{c}) = \{\mathrm{ap}(\mathsf{c}, \mathsf{o}) \mid \mathsf{o} \in \mathsf{0}\}$, which is a well-defined partial function by Defn. 4.11, and satisfies the composition condition by Defn. 4.3.

- $[\![(\mathcal{C}, \mathcal{D}, \mathrm{cp}, \mathsf{I}, \mathrm{proj})]\!]_{\emptyset,\circ} = (\mathcal{C}, \mathcal{D}, \mathrm{cp}, \mathrm{ap}, \mathsf{I}, \mathsf{0})$, where $\mathsf{0} = \{\mathrm{proj}(\mathsf{i}) \mid \mathsf{i} \in \mathsf{I}\}$ and $\mathrm{ap}(\mathsf{c}, \mathsf{d}) = \{\mathrm{proj}(\mathrm{cp}(\mathsf{c}, \mathsf{c}')) \mid \mathrm{proj}(\mathsf{c}') = \mathsf{d}\}$, which is a well-defined partial function satisfying the composition condition by Defn. 4.25.

The model translations use a similar approach to the formula translations. The translation $[\![-]\!]_p$ from $\mathrm{CL}_{\emptyset,\circ}$ to CL_{p} defines a projection function using the application of zero, while the reverse translation $[\![-]\!]_{\emptyset,\circ}$ reconstructs a zero set and an application function from the projection function.

Theorem 4.36 ($\mathrm{CL}_{\emptyset,\circ}/\mathrm{CL}_\mathrm{p}$ model equivalence)**.** *The model translations in Defn. 4.35 form a bijection: that is,* $[\![[\![\mathcal{M}_{\emptyset,\circ}]\!]_p]\!]_{\emptyset,\circ} = \mathcal{M}_{\emptyset,\circ}$ *and* $[\![[\![\mathcal{M}_p]\!]_{\emptyset,\circ}]\!]_p = \mathcal{M}_p$ *for arbitrary models* $\mathcal{M}_{\emptyset,\circ}$ *in* $\mathrm{CL}_{\emptyset,\circ}$ *and* $\mathcal{M}_p$ *in* CL_p*. Combined with the formula translations in Defn. 4.33, this results in equivalent satisfaction relations for* $\mathrm{CL}_{\emptyset,\circ}$ *and* CL_p*: that is*

$$\sigma, \mathsf{c}, \mathcal{M}_{\emptyset,\circ} \vDash_\mathcal{C} K \Leftrightarrow \sigma, \mathsf{c}, [\![\mathcal{M}_{\emptyset,\circ}]\!]_p \vDash_\mathcal{C} [\![K]\!]_p$$
$$\sigma, \mathsf{d}, \mathcal{M}_{\emptyset,\circ} \vDash_\mathcal{D} P \Leftrightarrow \sigma, \mathsf{d}, [\![\mathcal{M}_{\emptyset,\circ}]\!]_p \vDash_\mathcal{D} [\![P]\!]_p$$

$$\sigma, \mathsf{c}, \mathcal{M}_p \vDash_\mathcal{C} K' \Leftrightarrow \sigma, \mathsf{c}, [\![\mathcal{M}_p]\!]_{\emptyset,\circ} \vDash_\mathcal{C} [\![K']\!]_{\emptyset,\circ}$$
$$\sigma, \mathsf{d}, \mathcal{M}_p \vDash_\mathcal{D} P' \Leftrightarrow \sigma, \mathsf{d}, [\![\mathcal{M}_p]\!]_{\emptyset,\circ} \vDash_\mathcal{D} [\![P']\!]_{\emptyset,\circ}$$

for all models $\mathcal{M}_{\emptyset,\circ}$ *and* $\mathcal{M}_p$*, interpretations* σ*, contexts* $\mathsf{c} \in \mathcal{C}$ *and data elements* $\mathsf{d} \in \mathcal{D}$*, where* K *and* P *are arbitrary formulæ in* $\mathrm{CL}_{\emptyset,\circ}$*, and* K' *and* P' *arbitrary formulæ in* CL_p*.*

Proof. The bijectionality of the model translations follows directly from the translation semantics:

- $[\![[\![(\mathcal{C}, \mathcal{D}, \mathrm{cp}, \mathsf{I}, \mathrm{proj})]\!]_{\emptyset,\circ}]\!]_p = (\mathcal{C}, \mathcal{D}, \mathrm{cp}, \mathsf{I}, \mathrm{proj}')$ where
  $\mathrm{proj}'(\mathsf{c}) = \{\mathrm{proj}(\mathrm{cp}(\mathsf{c}, \mathsf{c}')) \mid \mathrm{proj}(\mathsf{c}') \in \{\mathrm{proj}(\mathsf{i}) \mid \mathsf{i} \in \mathsf{I}\}\}$
  $= \{\mathrm{proj}(\mathrm{cp}(\mathsf{c}, \mathsf{i})) \mid \mathsf{i} \in \mathsf{I}\}$ (by the composition condition) $= \mathrm{proj}(\mathsf{c})$.

- $[\![[\![(\mathcal{C}, \mathcal{D}, \mathrm{cp}, \mathrm{ap}, \mathsf{I}, \mathsf{0})]\!]_p]\!]_{\emptyset,\circ} = (\mathcal{C}, \mathcal{D}, \mathrm{cp}, \mathrm{ap}', \mathsf{I}, \mathsf{0}')$ where
  $\mathrm{ap}'(\mathsf{c}, \mathsf{d}) = \{\mathrm{ap}(\mathrm{cp}(\mathsf{c}, \mathsf{c}'), \mathsf{o}) \mid \mathsf{o} \in \mathsf{0}, \{\mathrm{ap}(\mathsf{c}', \mathsf{o}) | \mathsf{o} \in \mathsf{0}\} = \mathsf{d}\}$
  $= \{\mathrm{ap}(\mathsf{c}, \mathrm{ap}(\mathsf{c}', \mathsf{o})) \mid \mathsf{o} \in \mathsf{0}, \{\mathrm{ap}(\mathsf{c}', \mathsf{o}) | \mathsf{o} \in \mathsf{0}\} = \mathsf{d}\} = \mathrm{ap}(\mathsf{c}, \mathsf{d})$, and
  $\mathsf{0}' = \{\{\mathrm{ap}(\mathsf{i}, \mathsf{o}) \mid \mathsf{o} \in \mathsf{0}\} \mid \mathsf{i} \in \mathsf{I}\} = \mathsf{0}$.

Similarly, the left-to-right implications in the satisfaction relation equivalence follow immediately from the derived semantics of the model and formula translations. The right-to-left implications, meanwhile, follow from the bijectionality of the model translations shown above and the weak bijectionality of the formula translations given in the proof of Thm. 4.34.

Corollary 4.37 ($\mathrm{CL}_{\emptyset,\circ}/\mathrm{CL}_\mathrm{p}$ soundness and completeness)**.** *Using Thms. 4.34 and 4.36, it follows that the soundness and completeness of* $\mathrm{CL}_{\emptyset,\circ}$ *implies the soundness and completeness of* CL_p*.*

Chapter 5

Structured Data Models

This chapter adapts Context Logic to a number of structured data models, thereby illustrating many of the points from the previous chapters. The data structures considered here include sequences, multisets, sets, heaps, trees and terms.

5.1 Sequences

Sequences provide a simple but useful example of structured data. The sequences in this section are generated from an alphabet $\mathcal{A}$, with a concatenation operation : and an empty sequence 0. A natural interpretation of sequence contexts is as sequences with a hole in the middle, while context application corresponds to inserting sequences into the hole.

Definition 5.1 (Sequences)**.** Given an infinite set $\mathcal{A} = \{\mathsf{a}, \dots\}$ of elements, *sequences* $\mathsf{s} \in \mathcal{S}$ and *sequence contexts* $\mathsf{c} \in \mathcal{C}$ are defined by the grammars:

$$\mathsf{s} ::= 0 \mid \mathsf{a} \mid \mathsf{s} : \mathsf{s}$$
$$\mathsf{c} ::= - \mid \mathsf{c} : \mathsf{s} \mid \mathsf{s} : \mathsf{c}$$

with the following structural congruence $\equiv$ specifying that the concatenation operation : is associative with identity 0:

$$0 : \mathsf{s} \equiv \mathsf{s} \equiv \mathsf{s} : 0 \qquad 0 : \mathsf{c} \equiv \mathsf{c} \equiv \mathsf{c} : 0$$
$$\mathsf{s}_1 : (\mathsf{s}_2 : \mathsf{s}_3) \equiv (\mathsf{s}_1 : \mathsf{s}_2) : \mathsf{s}_3 \qquad \mathsf{s}_1 : (\mathsf{c} : \mathsf{s}_2) \equiv (\mathsf{s}_1 : \mathsf{c}) : \mathsf{s}_2$$
$$\mathsf{c} : (\mathsf{s}_1 : \mathsf{s}_2) \equiv (\mathsf{c} : \mathsf{s}_1) : \mathsf{s}_2 \qquad \mathsf{s}_1 : (\mathsf{s}_2 : \mathsf{c}) \equiv (\mathsf{s}_1 : \mathsf{s}_2) : \mathsf{c}$$

The insertion of sequences into sequence contexts is given by a total application

function (which is well-defined up to equivalence):

$$\begin{aligned} \mathrm{ap}(-, \mathsf{s}) &= \mathsf{s} \\ \mathrm{ap}(\mathsf{c} : \mathsf{s}', \mathsf{s}) &= \mathrm{ap}(\mathsf{c}, \mathsf{s}) : \mathsf{s}' \\ \mathrm{ap}(\mathsf{s}' : \mathsf{c}, \mathsf{s}) &= \mathsf{s}' : \mathrm{ap}(\mathsf{c}, \mathsf{s}) \end{aligned}$$

It is easy to show that $Seq = (\mathcal{S}, \mathcal{C}, \mathrm{ap}, -, 0)$ is a model of Context Logic with Zero. In fact, this can be extended to a model that includes context composition, by defining composition on sequence contexts. However, this is avoided for the sake of simplicity, especially since composition was found to be unnecessary for reasoning about tree update in Chapters 6 and 8. We therefore use $\mathrm{CL}_\emptyset$ formulæ for the model Seq. These formulæ, however, are not enough to allow useful reasoning about sequences: for example, there is nothing in the logic to allow us to differentiate between unequal elements. We therefore add sequence-specific formulæ to the logic for analysing the sequence structure, just as Separation Logic added heap-specific formulæ to BI in order to analyse heaps.

The added structural formulæ are chosen to correspond directly to the data structure definitions in Defn. 5.1. There are already formulæ corresponding to the empty sequence 0 and the empty context $-$, which means we only need to add formulæ for singleton sequences a and the concatenation operation $:$.

Definition 5.2 ($\mathrm{CL}_{\mathrm{seq}}$ formulæ)**.** The formulæ of Context Logic for Sequences ($\mathrm{CL}_{\mathrm{seq}}$) consist of the same context and data assertions as $\mathrm{CL}_\emptyset$, with the addition of special assertions for singleton sequences and context concatenation:

$P ::= p$		propositional variables
	$\mid \mathsf{a}$	**special formulæ**
	$\mid 0 \mid K \cdot P \mid K \lhd P$	structural formulæ
	$\mid P \Rightarrow P \mid \mathrm{false}$	logical formulæ
$K ::= k$		propositional variables
	$\mid \boldsymbol{P : K} \mid \boldsymbol{K : P}$	**special formulæ**
	$\mid I \mid P \rhd P$	structural formulæ
	$\mid K \Rightarrow K \mid \mathrm{False}$	logical formulæ

The interpretation of the new formulæ is straightforward: the data formula a is satisfied by the singleton sequence a; the context formula $P{:}K$ is satisfied by the result of concatenating a context satisfying K to a sequence satisfying P; and the context

formula $K : P$ is satisfied by the result of concatenating in the opposite direction. Thus, for example, both $(\mathsf{a} : I) \cdot \mathsf{b}$ and $(I : \mathsf{b}) \cdot \mathsf{a}$ describe the two-element sequence $\mathsf{a} : \mathsf{b}$. Note that this illustrates that it is not necessary to include a formula $P_1 : P_2$ to express the concatenation of two sequences, since this is derivable using the context concatenation connectives: $P_1{:}P_2$ can be expressed either as $(P_1{:}I){\cdot}P_2$ or as $(I{:}P_2){\cdot}P_1$. While the connective can therefore be omitted from the logic, its simpler notation is still used for shorthand.

Definition 5.3 ($\mathrm{CL}_{\mathrm{seq}}$ forcing semantics)**.** The semantics of $\mathrm{CL}_{\mathrm{seq}}$ is an extension of the semantics of $\mathrm{CL}_{\emptyset}$ for the model Seq. Given an interpretation function $\sigma : (\mathcal{V}_{\mathcal{D}} \to \mathcal{P}(\mathcal{S})) \times (\mathcal{V}_{\mathcal{C}} \to \mathcal{P}(\mathcal{C}))$ mapping propositional variables to sequence and context sets, the forcing semantics is given by two satisfaction relations $\sigma, \mathsf{s} \vDash_{\mathcal{S}} P$ and $\sigma, \mathsf{c} \vDash_{\mathcal{C}} K$ for sequences and sequence contexts, defined as in Defn. 4.12 and extended as follows:

$$\sigma, \mathsf{s} \vDash_{\mathcal{S}} \mathsf{a} \qquad \Leftrightarrow \mathsf{s} \equiv \mathsf{a}$$

$$\sigma, \mathsf{c} \vDash_{\mathcal{C}} P : K \Leftrightarrow \exists \mathsf{s} \in \mathcal{S}, \mathsf{c}' \in \mathcal{C}.\, (\sigma, \mathsf{s} \vDash_{\mathcal{S}} P \wedge \sigma, \mathsf{c}' \vDash_{\mathcal{C}} K \wedge \mathsf{c} \equiv \mathsf{s} : \mathsf{c}')$$
$$\sigma, \mathsf{c} \vDash_{\mathcal{C}} K : P \Leftrightarrow \exists \mathsf{s} \in \mathcal{S}, \mathsf{c}' \in \mathcal{C}.\, (\sigma, \mathsf{s} \vDash_{\mathcal{S}} P \wedge \sigma, \mathsf{c}' \vDash_{\mathcal{C}} K \wedge \mathsf{c} \equiv \mathsf{c}' : \mathsf{s})$$

As described in Chapters 3 and 4, it is possible to define a number of derived operators in $\mathrm{CL}_{\emptyset}$. These include a somewhere modality $\Diamond$ (Defn. 3.12), the existential adjoints $\blacktriangleright$ and $\blacktriangleleft$ (Defn. 3.11), and the $*$ connective on data (Defn. 4.15). These derived operations have intuitive interpretations on sequences. The somewhere modality describes arbitrary subsequences: $\Diamond P$ holds for a sequence if it contains a subsequence satisfying P. The star connective, meanwhile, describes the insertion of one sequence into another: $P_1 * P_2$ holds for a sequence if it is possible to remove a subsequence satisfying P_2 and obtain a sequence satisfying P_1. In addition to these derived connectives, it is also useful to define the adjoints of the concatenation connective, which describe adding sequences to either side of a sequence. Thus, the adjoint $P_1 \mathrel{-\!\!:} P_2$ holds if whenever a sequence satisfying P_1 is concatenated on the right, the result satisfies P_2; the other adjoint $P_1 \mathrel{:\!\!-} P_2$ is similar, but involves concatenating on the left.

Definition 5.4 (Concatenation adjoints)**.** The adjoints of the concatenation operation : are defined as follows:

$$P_1 \mathrel{-\!\!:} P_2 \triangleq (I : P_1) \lhd P_2 \qquad P_1 \mathrel{:\!\!-} P_2 \triangleq (P_1 : I) \lhd P_2$$

and have the following derived semantics:

$$\sigma, \mathsf{s} \vDash_{\mathcal{S}} P_1 \mathrel{-\!\!:} P_2 \Leftrightarrow \forall \mathsf{s}' \in \mathcal{S}.\, (\sigma, \mathsf{s}' \vDash_{\mathcal{S}} P_1 \Rightarrow \sigma, \mathsf{s} : \mathsf{s}' \vDash_{\mathcal{S}} P_2)$$
$$\sigma, \mathsf{s} \vDash_{\mathcal{S}} P_1 \mathrel{:\!\!-} P_2 \Leftrightarrow \forall \mathsf{s}' \in \mathcal{S}.\, (\sigma, \mathsf{s}' \vDash_{\mathcal{S}} P_1 \Rightarrow \sigma, \mathsf{s}' : \mathsf{s} \vDash_{\mathcal{S}} P_2)$$

Below are some examples of Context Logic formulæ on sequences.

Example 5.5 ($\mathrm{CL}_{\mathrm{seq}}$ formulæ)**.** The following are examples of formulæ in $\mathrm{CL}_{\mathrm{seq}}$, together with their interpretation on sequences:

(a) $\mathsf{a} : \mathsf{b} : \mathsf{a}$ *or* $(\mathsf{a} : I) \cdot ((\mathsf{b} : I) \cdot \mathsf{a})$ *or* $(\mathsf{a} : I) \cdot ((I : \mathsf{a}) \cdot \mathsf{b})$
the sequence containing a then b then a.

(b) $\mathsf{a} : \mathrm{true}$ *or* $(\mathsf{a} : I) \cdot \mathrm{true}$ *or* $(I : \mathrm{true}) \cdot \mathsf{a}$
any sequence beginning with an a.

(c) $\mathsf{a} * \mathrm{true}$ *or* $(0 \rhd \mathsf{a}) \cdot \mathrm{true}$ *or* $(\mathrm{true} : \mathsf{a}) \vee (\mathsf{a} : \mathrm{true})$
a sequence that either begins or ends with an a.

(d) $\Diamond \mathsf{a}$ *or* $\mathrm{True} \cdot \mathsf{a}$ *or* $\mathrm{true} * \mathsf{a}$ *or* $\mathrm{true} : \mathsf{a} : \mathrm{true}$
any sequence that contains an a.

(e) $(\mathsf{a} \mathrel{-\!\!:} P) : \mathsf{b}$ *or* $((\mathsf{a} : I) \lhd P) : \mathsf{b}$
a sequence ending in b that would satisfy P were the b replaced by an a.

(f) $(\mathsf{a} \mathrel{-\!\!*} P) * \mathsf{b}$ *or* $(0 \rhd ((\mathsf{a} \rhd P) \cdot 0)) \cdot \mathsf{b}$
a sequence containing a b that would satisfy P were the b removed and an a added anywhere in the sequence.

(g) $(\mathsf{a} \rhd P) \cdot \mathsf{b}$
a sequence containing a b that would satisfy P were the b removed and an a added *in the same place*.

(h) $0^+ \triangleq \mathrm{true}$, $(n+1)^+ \triangleq \neg I \cdot n^+$ and $n \triangleq n^+ \wedge \neg (n+1)^+$ for all $n \in \mathbb{N}$
any sequence with at least n elements (for n^+) or precisely n (for n).

(i) $\Box(1 \Rightarrow \mathsf{a})$ *or* $\neg\Diamond(1 \wedge \neg\mathsf{a})$
any sequence containing just a's

The various formula classes defined in Section 3.4 can be considered specifically for $\mathrm{CL}_{\mathrm{seq}}$. The pureness (Defn. 3.16), exactness (Defn. 3.20) and ubiquity (Defn. 3.28)

classes all correspond to simple properties of sequences. The only pure formulæ are true and false, since the only sequence sets that are closed under both context application and subsequencing are $\mathcal{S}$ and $\emptyset$. However, there do exist upward-closed formulæ (such as Example 5.5d) and downward-closed formulæ (such as Example 5.5i). The exact formulæ, meanwhile, are those that are satisfied by at most one sequence or context. Finally, while every satisfiable formula for the sequence model is omniplaceable (since context application is total), the only omnipresent (and consequently ubiquitous) formulæ are those that are satisfied by the empty context $-$ or empty sequence 0, since only these can be removed from every sequence.

The preciseness class (Defn. 3.24) is more interesting. Preciseness corresponds to the unique splitting of data into contexts and subdata. Thus, the only precise data formula for the sequence model is false, as any subsequence can occur multiple times within a sequence, allowing multiple splittings. For example, if P were a non-false precise formula, then $P : \mathsf{a} : P$ would describe a sequence containing at least two subsequences matching P, allowing two distinct splittings $(P : \mathsf{a} : I) \cdot P$ and $(I : \mathsf{a} : P) \cdot P$, and therefore breaking the preciseness condition: $P : \mathsf{a} : P = (P{:}\mathsf{a}{:}I)\cdot P \wedge (I{:}\mathsf{a}{:}P)\cdot P \not\Rightarrow ((P{:}\mathsf{a}{:}I)\wedge(I{:}\mathsf{a}{:}P))\cdot P = \text{false}$. In contrast, precise context formulæ *do* exist, and include any formula that describes only pairwise-incompatible contexts, where c_1 and c_2 are incompatible if there are no sequences s_1 and s_2 such that $\mathrm{ap}(\mathsf{c}_1, \mathsf{s}_1) = \mathrm{ap}(\mathsf{c}_2, \mathsf{s}_2)$. This can be expressed concretely: contexts $\mathsf{c}_1 = \mathsf{s}^1_l : - : \mathsf{s}^1_r$ and $\mathsf{c}_2 = \mathsf{s}^2_l : - : \mathsf{s}^2_r$ are incompatible if either s^1_l and s^2_l are incompatible on the left, meaning one is not an initial subsequence of the other, or s^1_r and s^2_r are incompatible on the right, meaning one is not a final subsequence of the other. Thus precise context formulæ include those that describe a specific context shape (for example, 'a context containing two elements, a hole, and an element'), incompatible starting sequences ('a context containing the sequence 020- or 0161-, followed by a hole'), and so on. Pair-precise pairs of formulæ, meanwhile, describe similar splitting properties, though neither formula has to be precise by itself: for example the context formula $(\text{true}{:}I)$ and the singleton sequence formula 1 from Example 5.5h form a precise pair.

There is a close correspondence between preciseness and the specification of data update, in that precise formulæ allow us to uniquely specify subsequences, and hence the location at which updates on subsequences, such as deletion, should take place. Thus, the three examples considered at the end of the last paragraph could

respectively be used to specify updates of: the subsequence consisting of all but the first two and last one elements; the subsequence following the prefix 020 or 0161; and the final element in a sequence.

5.2 Multisets

Multisets are the simplest interesting example where Context Logic collapses to Bunched Logic. The multisets in this section are constructed from an alphabet $\mathcal{A}$, a union operation : and an empty multiset 0, just like the sequences in Section 5.1, only with the additional condition that : is commutative. Multiset contexts, meanwhile, correspond to multisets with a hole that, due to the lack of ordering, is not 'located' anywhere.

Definition 5.6 (Multisets)**.** *Multisets* $\mathsf{m} \in \mathcal{M}$ and *multiset contexts* $\mathsf{c} \in \mathcal{C}$ are defined as sequences and sequence contexts (Defn. 5.1), with the additional congruence rules stating that the concatenation (union) operation is commutative:

$$\mathsf{m}_1 : \mathsf{m}_2 \equiv \mathsf{m}_2 : \mathsf{m}_1 \qquad \mathsf{m} : \mathsf{c} \equiv \mathsf{c} : \mathsf{m}$$

Context application is also as defined in Defn. 5.1.

As in the previous section, it is easy to show that $Mult = (\mathcal{M}, \mathcal{C}, \mathrm{ap}, \{-\}, \{0\})$ is a model of Context Logic with Zero. Since the hole in multiset contexts has no location, however, there is no real structural difference between contexts and multisets, and it is possible to view contexts as equal to their underlying multisets and context application as multiset union. Thus the above model corresponds to the monoidal model $(\mathcal{M}, \mathcal{M}, :, \{0\}, \{0\})$. By Thm. 4.22, the $\mathrm{CL}_\emptyset$ satisfaction relation for this model is just as expressive as the Bunched Logic satisfaction relation for the model $(\mathcal{M}, :, 0)$.

Again, we extend the logic with formulæ specific to the data model in order to increase expressivity. In this case, however, we only need to add formulæ for singleton multisets. It is unnecessary to add any formulæ for the union operation, as this is already expressible using context application: applying a context to a multiset corresponds to taking the union of their underlying elements, while multisets can be turned into contexts using the adjoint $\triangleright$ and 0. Thus, it is possible to express $K : P \equiv P : K$ as $K \cdot P$ and $P_1 : P_2$ as $(0 \triangleright P_1) \cdot P_2$.

Definition 5.7 ($\mathrm{CL}_{\mathrm{mult}}$ formulæ)**.** The formulæ of Context Logic for Multisets ($\mathrm{CL}_{\mathrm{mult}}$) consist of the same context and data assertions as $\mathrm{CL}_{\emptyset}$, with the addition of special assertions for singleton multisets:

$$
\begin{array}{lll}
P ::= p & & \text{propositional variables} \\
\quad \mid \mathsf{a} & & \textbf{special formulæ} \\
\quad \mid 0 \mid K \cdot P \mid K \lhd P & & \text{structural formulæ} \\
\quad \mid P \Rightarrow P \mid \text{false} & & \text{logical formulæ} \\
\\
K ::= k & & \text{propositional variables} \\
\quad \mid & & \text{special formulæ} \\
\quad \mid I \mid P \rhd P & & \text{structural formulæ} \\
\quad \mid K \Rightarrow K \mid \text{False} & & \text{logical formulæ}
\end{array}
$$

Definition 5.8 ($\mathrm{CL}_{\mathrm{mult}}$ forcing semantics)**.** The semantics of $\mathrm{CL}_{\mathrm{mult}}$ is an extension of the semantics of $\mathrm{CL}_{\emptyset}$ for the model $Mult$. Given an interpretation function $\sigma : (\mathcal{V}_{\mathcal{D}} \to \mathcal{P}(\mathcal{M})) \times (\mathcal{V}_{\mathcal{C}} \to \mathcal{P}(\mathcal{C}))$ mapping propositional variables to multiset and context sets, the forcing semantics is given by two satisfaction relations $\sigma, \mathsf{m} \vDash_{\mathcal{M}} P$ and $\sigma, \mathsf{c} \vDash_{\mathcal{C}} K$ for multisets and multiset contexts, defined as in Defn. 4.12 and extended as follows:

$$\sigma, \mathsf{m} \vDash_{\mathcal{M}} \mathsf{a} \Leftrightarrow \mathsf{m} \equiv \mathsf{a}$$

The same derived formulæ can be defined for multisets as were defined for sequences. Due to the monoidal nature of the model, the derived $*$ connective coincides with : on data (see Section 4.3). Likewise, the respective adjoints, $-\!\!:$ and $:\!\!-$, coincide with $-\!\!*$ and $*\!\!-$; in fact, due to commutativity, all four adjoint connectives are the same. Finally, context composition is now also expressible in the logic (which was not the case for sequences), since the data formula 0 is now both precise and omniplaceable: by Lemma 4.8, $K_1 \circ K_2$ can be expressed as $0 \rhd (K_1 \cdot (K_2 \cdot 0))$. Below are a few examples of Context Logic formulæ for multisets.

Example 5.9 ($\mathrm{CL}_{\mathrm{mult}}$ formulæ)**.** The following are examples of formulæ in $\mathrm{CL}_{\mathrm{mult}}$, together with their interpretation on multisets:

(a) $\mathsf{a} * \mathsf{a} * \mathsf{b}$ *or* $(0 \rhd \mathsf{a}) \cdot ((0 \rhd \mathsf{a}) \cdot \mathsf{b})$ *or* $(0 \rhd ((0 \rhd \mathsf{a}) \cdot \mathsf{a})) \cdot \mathsf{b}$
the multiset containing two a's and a b.

(b) $\Diamond \mathsf{a}$ *or* $\mathrm{True} \cdot \mathsf{a}$ *or* $\mathrm{true} * \mathsf{a}$ *or* $\mathsf{a} * \mathrm{true}$
any multiset containing an a.

(c) $(\mathsf{a} \mathrel{-\!\!*} P) * \mathsf{b}$ *or* $(\mathsf{a} \rhd P) \cdot \mathsf{b}$

a multiset containing a b that would satisfy P were the b removed and replaced by an a.

The commutativity of the union connective also changes the formula classes. Pureness, exactness and ubiquity still describe the same properties: only true and false are pure, formulæ describing unique multisets and contexts are exact, and anything satisfied by $-$ or 0 is ubiquitous. The concept of preciseness changes, however: the lack of ordered structure means that any exact data formula is now precise (since multiset subtraction, unlike sequence subtraction, is defined uniquely), while only exact context formulæ are precise (it is no longer possible to use the ordering of elements to differentiate contexts). A pair-precise formula comprises one precise formula (either context or data) and one arbitrary formula. In other words, a multiset can be located either by giving its exact elements, or the exact elements of its context.

5.3 Linear Sequences and Sets

A common feature of data models that allow data update, and in particular local update, is the presence of unique location identifiers for specifying the update location. Examples include heap structures with unique cell addresses, and a variety of tree structures containing unique node identifiers. Modelling data structures with unique identifiers in Context Logic is easy: the same formulæ can be defined as when modelling data structures without uniqueness, and then interpreted on data and contexts with uniqueness. The one novel feature of these models is that context application becomes a partial operation.

In this section we consider linear sequences and sets, by restricting the definitions of sequences and multisets given in Sections 5.1 and 5.2 to unique elements.

Definition 5.10 (Linear sequences and sets)**.** Linear sequences and sets are those sequences and multisets defined in Defns. 5.1 and 5.6 that possess unique elements: that is, those d where $\mathsf{d} = \mathsf{d}_1 : \mathsf{d}_2$ implies that the label sets $\mathrm{locs}(\mathsf{d}_1)$ and $\mathrm{locs}(\mathsf{d}_2)$ are disjoint. Similarly, linear sequence and set contexts are those contexts that possess unique labels: that is, those c where $\mathsf{c} = \mathsf{c}' : \mathsf{d}$ or $\mathsf{c} = \mathsf{d} : \mathsf{c}'$ implies that the label sets $\mathrm{locs}(\mathsf{c}')$ and $\mathrm{locs}(\mathsf{d})$ are disjoint. The context application function is defined as in Defn. 5.1 whenever the result has unique labels, and is undefined otherwise.

The introduction of unique identifiers has no effect on the logic: it is possible to define $\mathrm{CL}_{\mathrm{seq}}$ and $\mathrm{CL}_{\mathrm{mult}}$ as before, but to interpret them on the new data models with unique elements. Thus, the formula $\mathsf{a} : \mathsf{a}$ becomes unsatisfiable for both sequences and multisets, as there is no sequence or multiset with two a's, while the formula $\mathsf{a} : \mathsf{b}$ is satisfied by the same sequence and multiset as before. This change in interpretation affects the interpretations of the formula classes described in the previous sections. The interpretations of pureness and exactness remain the same, as does that of omnipresence. However, the uniqueness restriction means that a formula which is satisfied solely by data containing some specific element a, such as the formulæ a or $\mathrm{true} : \mathsf{a}$, cannot be omniplaceable. Similarly, the concept of preciseness does not change for sets, since locating a set still corresponds to specifying all its elements. However, preciseness for linear sequences *is* affected by the presence of unique identifiers. As noted before, precise formulæ describe only pairwise-incompatible sequences or contexts, where two things are incompatible if it is not possible to apply contexts or subsequences to them and obtain the same result. Adding unique identifiers to sequences only restricts context application and therefore results in more incompatible pairs, and hence more precise formulæ. For linear sequences, two sequences s and s' are only compatible if they can be split into pairs of subsequences $\mathsf{s}_1 : \mathsf{s}_2$ and $\mathsf{s}_2 : \mathsf{s}_3$, such that s_1 and s_3 have no shared elements. Hence, the data formula $\mathsf{a} : \mathrm{true} : \mathsf{b}$, describing sequences starting with a and ending with b, is precise, since any two such sequences are incompatible. Similar reasoning can be applied for context and pair preciseness.

5.4 Heaps

Heaps are a classic example of structured data, and the primary reasoning target of Separation Logic. A heap is typically viewed as a set of locations, with values at each location. For simplicity, the heaps considered here contain only pointer values, which can be either location addresses or the null value 'nil'. To allow the modelling of pointer arithmetic, location addresses are represented by the positive integers, while nil is represented by zero. As for multisets and sets, heap contexts consist of a heap and an unlocated hole.

In addition to heaps, Separation Logic also models a variable store, which maps variables to values. This corresponds to the more typical Hoare logic view of state,

and allows reasoning about variable-based heap update languages. For simplicity, the store considered here is a total function from variables to values. This allows us to assume that variable lookup is always defined, though uninitialised variables will have undetermined values. This idealisation is acceptable in practice, since any reasoning example will only depend on finitely many variables.

Definition 5.11 (Preheaps)**.** A *preheap* $\mathsf{h} \in \mathcal{H}_{\text{pre}}$ is defined by the grammar:

$$\mathsf{h} ::= \text{emp} \mid \mathsf{n} \mapsto \mathsf{v} \mid \mathsf{h} * \mathsf{h}$$

where $\mathsf{n} \in \mathbb{N}^+$ and $\mathsf{v} \in \mathbb{N}$. The set of locations $\text{locs}(\mathsf{h})$ in a preheap is given by:

$$\text{locs}(\text{emp}) = \emptyset \qquad \text{locs}(\mathsf{n} \mapsto \mathsf{v}) = \{\mathsf{n}\} \qquad \text{locs}(\mathsf{h}_1 * \mathsf{h}_2) = \text{locs}(\mathsf{h}_1) \cup \text{locs}(\mathsf{h}_2)$$

A structural congruence $\equiv$ between preheaps specifies that the separation connective $*$ is commutative, associative, and has identity emp. This is well-defined with respect to locs, corresponding to a multiset interpretation, and is the smallest congruence satisfying:

$$\text{emp} * \mathsf{h} \equiv \mathsf{h} * \text{emp} \equiv \mathsf{h} \quad \mathsf{h}_1 * (\mathsf{h}_2 * \mathsf{h}_3) \equiv (\mathsf{h}_1 * \mathsf{h}_2) * \mathsf{h}_3 \quad \mathsf{h}_1 * \mathsf{h}_2 \equiv \mathsf{h}_2 * \mathsf{h}_1$$

Definition 5.12 (Heaps)**.** A *heap* $\mathsf{h} \in \mathcal{H}$ is a preheap with unique locations: that is, where $\mathsf{h} = \mathsf{h}_1 * \mathsf{h}_2$ implies $\text{locs}(\mathsf{h}_1) \cap \text{locs}(\mathsf{h}_2) = \emptyset$. The structural congruence $\equiv$ is well-defined on heaps, with the separating connective $\mathsf{h}_1 * \mathsf{h}_2$ now a partial operation defined only when $\text{locs}(\mathsf{h}_1) \cap \text{locs}(\mathsf{h}_2) = \emptyset$.

Definition 5.13 (Heap contexts)**.** A *heap context* $\mathsf{c} \in \mathcal{C}$ is defined as a heap with a hole, written $\mathsf{h} * -$. Context application is given by $\text{ap}(\mathsf{h} * -, \mathsf{h}') \triangleq \mathsf{h} * \mathsf{h}'$, and is partial due to the uniqueness of identifiers.

Definition 5.14 (Stores)**.** Given an infinite set $\text{Var} = \{x, y, \dots\}$ of variables, a *store* $\mathsf{s} \in \mathcal{S}$ is a total function $\mathsf{s} : \text{Var} \to \mathbb{N}$, returning the value of each variable.

In addition to heaps and stores, it is also useful to define arithmetic expressions, which can be used for specifying pointer arithmetic.

Definition 5.15 (Expressions)**.** An *expression* E is given by the following grammar:

$$E ::= \text{nil} \mid \text{n} \mid x \mid E + E \mid E - E$$

where $\mathrm{n} \in \mathbb{N}$. The valuation of E on a store s is written $[\![E]\!]\mathsf{s}$ and given by

$$[\![\mathrm{nil}]\!]\mathsf{s} = 0 \quad [\![\mathrm{n}]\!]\mathsf{s} = \mathrm{n} \quad [\![x]\!]\mathsf{s} = \mathsf{s}(x) \qquad \begin{array}{l} [\![E + F]\!]\mathsf{s} = [\![E]\!]\mathsf{s} + [\![F]\!]\mathsf{s} \\ [\![E - F]\!]\mathsf{s} = [\![E]\!]\mathsf{s} - [\![F]\!]\mathsf{s} \end{array}$$

Like for multisets, it is easy to show that $Heap = (\mathcal{H}, \mathcal{C}, \mathrm{ap}, \{\mathrm{emp}\}, \{\mathrm{emp}\})$ is a model of Context Logic with Zero. A more interesting question is how to incorporate variables into the model. In fact, this is easily done using the set-indexing construction defined in Example 3.8: $Heap^{\mathcal{S}}$ describes a heap-store model where every heap and heap context is indexed by a store $\mathsf{s} \in \mathcal{S}$ and context application is only defined for states with matching stores.

As for multisets and sets, there is no real distinction between heaps and heap contexts, and heap composition can be expressed using context application and 0. Thus the only structural heap formula that has to be added to the logic is one corresponding to singleton heaps. This is written $E \mapsto F$, and uses expressions to refer to both the heap location and its value. Note that the presence of variables in the expression language means that this formula can depend on the store part of the state, as well as the heap. Boolean formulæ for expression equality and inequality are also included in the logic, though these will later be shown to be derivable. Finally, the presence of variables in the logic results in the inclusion of variable quantification.

Definition 5.16 ($\mathrm{CL}_{\mathrm{heap}}$ formulæ)**.** The formulæ of Context Logic for Heaps ($\mathrm{CL}_{\mathrm{heap}}$) consist of the same context and data assertions as $\mathrm{CL}_{\emptyset}$ with the addition of a points-to operator, expression equality and inequality, and quantification over variables:

$P ::= p$		propositional variables
	$\mid \boldsymbol{E \mapsto F} \mid \boldsymbol{E = F} \mid \boldsymbol{E < F}$	**special formulæ**
	$\mid 0 \mid K \cdot P \mid K \lhd P$	structural formulæ
	$\mid P \Rightarrow P \mid \mathrm{false} \mid \boldsymbol{\exists x.P}$	**logical formulæ**
$K ::= k$		propositional variables
	$\mid$	special formulæ
	$\mid I \mid P \rhd P$	structural formulæ
	$\mid K \Rightarrow K \mid \mathrm{False} \mid \boldsymbol{\exists x.K}$	**logical formulæ**

Definition 5.17 ($\mathrm{CL}_{\mathrm{heap}}$ forcing semantics)**.** The semantics of $\mathrm{CL}_{\mathrm{heap}}$ is an extension of the semantics of $\mathrm{CL}_{\emptyset}$ for the model $Heap^{\mathcal{S}}$. Given an interpretation function

$\sigma : (\mathcal{V}_\mathcal{D} \to \mathcal{P}(\mathcal{S} \times \mathcal{H})) \times (\mathcal{V}_\mathcal{C} \to \mathcal{P}(\mathcal{S} \times \mathcal{C}))$ mapping propositional variables to store-indexed heap and context sets, the forcing semantics is given by two satisfaction relations $\sigma, \mathsf{s}, \mathsf{h} \vDash_\mathcal{H} P$ and $\sigma, \mathsf{s}, \mathsf{c} \vDash_\mathcal{C} K$ for heaps and heap contexts with variable stores, defined as in Defn. 4.12 and extended by:

$$\begin{aligned}
\sigma, \mathsf{s}, \mathsf{h} \vDash_\mathcal{H} E \mapsto F &\Leftrightarrow \exists \mathsf{m} \in \mathbb{N}^+, \mathsf{v} \in \mathbb{N}. ([\![E]\!]\mathsf{s} = \mathsf{m} \wedge [\![F]\!]\mathsf{s} = \mathsf{v} \wedge \mathsf{h} \equiv \mathsf{m} \mapsto \mathsf{v}) \\
\sigma, \mathsf{s}, \mathsf{h} \vDash_\mathcal{H} E = F &\Leftrightarrow [\![E]\!]\mathsf{s} = [\![F]\!]\mathsf{s} \\
\sigma, \mathsf{s}, \mathsf{h} \vDash_\mathcal{H} E < F &\Leftrightarrow [\![E]\!]\mathsf{s} < [\![F]\!]\mathsf{s} \\
\sigma, \mathsf{s}, \mathsf{h} \vDash_\mathcal{H} \exists x.P &\Leftrightarrow \exists \mathsf{v} \in \mathbb{N}.\ \sigma, [\mathsf{s}|x \leftarrow \mathsf{v}], \mathsf{h} \vDash_\mathcal{H} P \\
\\
\sigma, \mathsf{s}, \mathsf{c} \vDash_\mathcal{C} \exists x.K &\Leftrightarrow \exists \mathsf{v} \in \mathbb{N}.\ \sigma, [\mathsf{s}|x \leftarrow \mathsf{v}], \mathsf{c} \vDash_\mathcal{C} K
\end{aligned}$$

where $[\mathsf{s}|x \leftarrow \mathsf{v}]$ refers to the function s overwritten with $\mathsf{s}(x) = \mathsf{v}$.

The same derived formulæ can be defined for heaps as for sequences and sets. For heaps with identical stores, the derived $*$ connective coincides with the heap composition $*$. Furthermore, the store-indexed model means that the partial validity formula defined in Defn. 3.15 describes the validity of a formula across all heaps, but only with respect to the current store. This in turn means that the formula for expression equality is derivable: it can be expressed as a validity statement on singleton heap formulæ, as shown below. Expression inequality, meanwhile, can be expressed using equality and quantification.

Lemma 5.18 (Boolean formulæ)**.** *Expression equality and inequality are derivable using the validity connective* valid_i *defined in Defn. 3.15:*

$$\begin{aligned}
(E = F) &\triangleq \mathrm{valid}_i(E \mapsto E \Leftrightarrow F \mapsto F) \\
(E < F) &\triangleq \exists x.\, (E + x = F) \wedge \neg(x = 0)
\end{aligned}$$

Proof. Results follow directly from the forcing semantics.

Below are some examples of $\mathrm{CL}_{\mathrm{heap}}$ formulæ. These are all written using the derived $*$ and $-\!\!*$ connectives, and can therefore be read as either Context Logic or Separation Logic formulæ.

Definition 5.19 (Shorthand notation)**.** The following shorthand formulæ are defined as in Separation Logic:

$$\begin{aligned}
E \mapsto - &\triangleq \exists x.\, E \mapsto x \\
E \mapsto F_1, F_2 &\triangleq (E \mapsto F_1) * (E + 1 \mapsto F_2) \\
E \hookrightarrow F &\triangleq \mathrm{true} * (E \mapsto F) \\
P_1 \mathrel{-\!\!*_\exists} P_2 &\triangleq \neg(P_1 -\!\!* \neg P_2)
\end{aligned}$$

These, in turn, describe: a single heap cell at E with an arbitrary value; two adjacent cells at E and $E+1$ with values F_1 and F_2; an arbitrary heap containing the cell $E \mapsto F$; and a heap to which it is possible to compose a subheap satisfying P_1 and obtain a heap satisfying P_2.

Example 5.20 ($\mathrm{CL}_{\mathrm{heap}}$ formulæ)**.** The following are examples of formulæ in $\mathrm{CL}_{\mathrm{heap}}$, together with their interpretation on heaps:

(a) $1 \mapsto 2 * 2 \mapsto \mathrm{nil}$

the 2-cell heap representing a null-terminated list with locations 1 and 2.

(b) $x \mapsto y * y \mapsto x$

a 2-cell heap representing a circular list from (the value of) x to (the value of) y and back. Note that x cannot equal y.

(c) $x \mapsto y \wedge y \mapsto x \ \mathit{or}\ x \mapsto x \wedge x = y$

a 1-cell heap representing a circular list at x, which must equal y.

(d) $(\mathrm{true} * x \mapsto y) \wedge (\mathrm{true} * y \mapsto x)$

an arbitrary heap either containing a circular list at x, if $x = y$, or one from x to y and back, if $x \neq y$.

(e) $[E_{start} : E_{len}] \triangleq \forall x.\, \Diamond(x \mapsto -) \Leftrightarrow (E_{start} \leq x \wedge x < E_{start} + E_{len})$

a heap block with E_{len} consecutive cells, starting at address E_{start}.

$x \in [E_{start} : E_{len}] \triangleq (E_{start} \leq x \wedge x < E_{start} + E_{len})$

a pure formula stating that x is in the block $[E_{start} : E_{len}]$.

(f) $\text{unique-ptrs} \triangleq \neg\exists x, y, z.\, \Diamond(x \mapsto z * y \mapsto z)$

a heap with unique pointer targets.

(g) $\mathrm{size}(E) \triangleq \exists x.\, ([x : E] \wedge \text{unique-ptrs}) \mathrel{-\!\!*}_{\exists}$
$\forall y, z.\, (y \in [x : E] \wedge \Diamond(y \mapsto z)) \Rightarrow (z \notin [x : E] \wedge \Diamond(z \mapsto -)) \wedge$
$\forall z.\, (z \notin [x : E] \wedge \Diamond(z \mapsto -)) \Rightarrow \exists y.\, (y \in [x : E] \wedge \Diamond(y \mapsto z))$

a heap with precisely E cells. This works by adding a block of cells of length E to the heap and forming a one-to-one correspondence between the block pointers and the original cells in the heap. This is made possible by ensuring that the pointers within the block are unique.

As already mentioned in Section 3.4, the formula classes in $\mathrm{CL}_{\mathrm{heap}}$ agree with the homonymous formula classes in Separation Logic. Pure formulæ describe independence from the heap: for example, the formula $E = F$ is pure as its satisfaction depends only on the store valuations of E and F. Syntactically, any assertion that does not contain 0 or $\mapsto$ is pure. Upward closed formulæ correspond to the 'intuitionistic' formulæ of Separation Logic [Rey02]: for example, the formula $E \hookrightarrow F$, defined above, is upward closed. Exact formulæ describe satisfaction by at most one heap or context for any given store: for example, the formula $x \mapsto \mathrm{nil}$, which describes different heaps for different values of x, is exact. Syntactically, any assertions built using just $\mapsto$ and $*$ are exact. As for multisets, any formula satisfied by $(0 * -)$ or 0 is ubiquitous.

Once again, the preciseness class is the most interesting. A heap assertion P is precise if for any store and heap there is at most one subheap satisfying P. The presence of unique identifiers and pointers makes preciseness richer than simple exactness. For example, the formula $E \mapsto -$, describing a heap cell at E with arbitrary value, is precise but not exact. Perhaps a more surprising example is the formula describing all heaps that represent a linked list structure starting at a location E and ending with a pointer to F, such as the heap satisfying $E \mapsto F$, or the different heaps satisfying $\exists x.\, E \mapsto x * x \mapsto F$. Despite describing heaps of arbitrary size, this too determines a unique subheap and is therefore precise.

5.5 Trees

Context-Logic based Hoare reasoning about tree update forms the bulk of the second part of this thesis. Here we consider Context Logic reasoning about trees. The trees considered in this section consist of a simple forest structure with unique node identifiers ranging over an alphabet $\mathcal{N}$, and are constructed using a branching operation $\mathsf{n}[-]$ and a commutative composition operation $|$. The choices are all practical: viewing trees as node-labelled forests is equivalent to viewing them as edge-labelled trees, but corresponds more closely to standard representations of semistructured data such as XML; the presence of unique node identifiers is necessary to allow local update, which is a key part of the reasoning framework considered in Part B; and the commutativity of composition makes the model simpler, while also corresponding to a view of trees as databases, describing hierarchies (as in Ambient Logic) rather than

documents (as in HTML). Later work will present a more complicated tree structure, incorporating node labels and pointers. However, this simple structure is sufficient for now.

In addition to trees, we also consider tree contexts, which simply consist of trees with a hole inside. Like for heaps, the model also includes a variable store. This time the store contains two types of variables: name variables and tree variables, which take names and trees as values. These are used in the tree update language considered in Chapter 6.

Definition 5.21 (Pretrees)**.** Given an infinite set $\mathcal{N} = \{\mathsf{m}, \mathsf{n}, \dots\}$ of location names, *pretrees* $\mathsf{t} \in \mathcal{T}_{\text{pre}}$ and *pretree contexts* $\mathsf{c} \in \mathcal{C}_{\text{pre}}$ are defined by the grammars:

$$\mathsf{t} ::= 0 \mid \mathsf{n}[\mathsf{t}] \mid (\mathsf{t} \mid \mathsf{t})$$
$$\mathsf{c} ::= - \mid \mathsf{n}[\mathsf{c}] \mid (\mathsf{c} \mid \mathsf{t}) \mid (\mathsf{t} \mid \mathsf{c})$$

where $\mathsf{n} \in \mathcal{N}$. The sets of locations $\text{locs}(\mathsf{t})$ and $\text{locs}(\mathsf{c})$ in pretrees and pretree contexts are given by:

$$\text{locs}(0) = \emptyset \quad \text{locs}(\mathsf{n}[\mathsf{t}']) = \{\mathsf{n}\} \cup \text{locs}(\mathsf{t}') \quad \text{locs}(\mathsf{t}_1 \mid \mathsf{t}_2) = \text{locs}(\mathsf{t}_1) \cup \text{locs}(\mathsf{t}_2)$$
$$\text{locs}(-) = \emptyset \quad \text{locs}(\mathsf{n}[\mathsf{c}']) = \{\mathsf{n}\} \cup \text{locs}(\mathsf{c}') \quad \left.\begin{array}{l}\text{locs}(\mathsf{c}' \mid \mathsf{t}) \\ \text{locs}(\mathsf{t} \mid \mathsf{c}')\end{array}\right\} = \text{locs}(\mathsf{c}') \cup \text{locs}(\mathsf{t})$$

The structural congruence $\equiv$ between pretrees and between pretree contexts specifies that $\mid$ is commutative, associative, and has identity 0. This is well-defined with respect to locs and is the smallest congruence satisfying:

$$0 \mid \mathsf{t} \equiv \mathsf{t} \mid 0 \equiv \mathsf{t} \quad \mathsf{t}_1 \mid (\mathsf{t}_2 \mid \mathsf{t}_3) \equiv (\mathsf{t}_1 \mid \mathsf{t}_2) \mid \mathsf{t}_3 \quad \mathsf{t}_1 \mid \mathsf{t}_2 \equiv \mathsf{t}_2 \mid \mathsf{t}_1$$
$$0 \mid \mathsf{c} \equiv \mathsf{c} \mid 0 \equiv \mathsf{c} \quad \mathsf{c} \mid (\mathsf{t}_1 \mid \mathsf{t}_2) \equiv (\mathsf{c} \mid \mathsf{t}_1) \mid \mathsf{t}_2 \qquad \mathsf{c} \mid \mathsf{t} \equiv \mathsf{t} \mid \mathsf{c}$$

The insertion of a pretree t into a pretree context c is given by a total application function $\text{ap} : \mathcal{C}_{\text{pre}} \times \mathcal{T}_{\text{pre}} \to \mathcal{T}_{\text{pre}}$, defined inductively by:

$$\text{ap}(-, \mathsf{t}) = t$$
$$\text{ap}(\mathsf{n}[\mathsf{c}], \mathsf{t}) = \mathsf{n}[\text{ap}(\mathsf{c}, \mathsf{t})]$$
$$\text{ap}(\mathsf{c} \mid \mathsf{t}', \mathsf{t}) = \text{ap}(\mathsf{c}, \mathsf{t}) \mid \mathsf{t}'$$
$$\text{ap}(\mathsf{t}' \mid \mathsf{c}, \mathsf{t}) = \mathsf{t}' \mid \text{ap}(\mathsf{c}, \mathsf{t})$$

Definition 5.22 (Trees and contexts)**.** A *tree* $\mathsf{t} \in \mathcal{T}$ is a pretree with unique locations: that is, where $\mathsf{t} = \mathsf{n}[\mathsf{t}']$ implies $\mathsf{n} \notin \text{locs}(\mathsf{t}')$ and $\mathsf{t} = \mathsf{t}_1 \mid \mathsf{t}_2$ implies $\text{locs}(\mathsf{t}_1) \cap \text{locs}(\mathsf{t}_2) = \emptyset$ (also written $\mathsf{t}_1 \# \mathsf{t}_2$). Similarly, a *tree context* $\mathsf{c} \in \mathcal{C}$ is a

pretree context with unique locations: that is, where $\mathsf{c} = \mathsf{n}[\mathsf{c}']$ implies $\mathsf{n} \notin \mathrm{locs}(\mathsf{c}')$ and $\mathsf{c} = \mathsf{c}' \mid \mathsf{t}$ or $\mathsf{c} = \mathsf{t} \mid \mathsf{c}'$ implies $\mathrm{locs}(\mathsf{c}') \cap \mathrm{locs}(\mathsf{t}) = \emptyset$. The structural congruence $\equiv$ is well-defined on both trees and tree contexts, with $\mathsf{n}[-]$ and $\mid$ now partial operations. The insertion of a tree t into a tree context c is given by the appropriate restriction of ap, which is well-defined.

Definition 5.23 (Stores)**.** Given infinite sets $\mathrm{Var}_{\mathcal{N}} = \{m, n, \dots\}$ of name variables and $\mathrm{Var}_{\mathcal{T}} = \{x, y, \dots\}$ of tree variables, a *store* $\mathsf{s} \in \mathcal{S}$ is a pair of total functions $\mathsf{s} : \mathrm{Var}_{\mathcal{N}} \to \mathcal{N} \times \mathrm{Var}_{\mathcal{T}} \to \mathcal{T}$, returning the values of name and tree variables.

It is easy to show that $Tree = (\mathcal{T}, \mathcal{C}, \mathrm{ap}, \{-\}, \{0\})$ is a model of Context Logic with Zero. As before, we incorporate the variable store by defining the model $Tree^{\mathcal{S}}$, which indexes the tree and tree context structure with specific stores using the set-indexing construction in Example 3.8. Similarly, we introduce new formulæ to the logic that correspond to the tree and context data structure definitions. Since zero is already defined, this can be done at just the context level, with the remaining data connectives derivable using context application. Thus, we introduce a context composition formula $P \mid K$, describing the composition of a tree satisfying P and a context satisfying K, and use it to define tree composition $P \mid P'$ as $(P \mid I) \cdot P'$. Likewise, we define a context branching formula $n[K]$, describing a tree with a root node n and a subcontext satisfying K, and use it to derive tree branching $n[P]$ as $n[I] \cdot P$. The use of a variable n, rather than a constant n, to specify the root node of a branch is motivated by a view of node identifiers as dynamically generated addresses, rather than user-provided values, discussed further in Chapter 6. It also provides a way to reason about name variables. Tree variable reasoning is added more directly by introducing a formula x, which is satisfied only by the value of x. Finally, quantification over both types of variables is also included.

Definition 5.24 ($\mathrm{CL}_{\mathrm{tree}}$ formulæ)**.** The formulæ of Context Logic for Trees ($\mathrm{CL}_{\mathrm{tree}}$) consist of the same context and data assertions as $\mathrm{CL}_{\emptyset}$ with the addition of formulæ representing tree variables, branching contexts, parallel composition contexts and

quantification:

$$
\begin{array}{lll}
P ::= p & & \text{propositional variables} \\
\quad \mid \boldsymbol{x} & & \textbf{special formulæ} \\
\quad \mid 0 \mid K \cdot P \mid K \lhd P & & \text{structural formulæ} \\
\quad \mid P \Rightarrow P \mid \text{false} \mid \exists \boldsymbol{n}.\boldsymbol{P} \mid \exists \boldsymbol{x}.\boldsymbol{P} & & \textbf{logical formulæ} \\
\\
K ::= k & & \text{propositional variables} \\
\quad \mid \boldsymbol{n}[\boldsymbol{K}] \mid (\boldsymbol{P} \mid \boldsymbol{K}) & & \textbf{special formulæ} \\
\quad \mid I \mid P \rhd P & & \text{structural formulæ} \\
\quad \mid K \Rightarrow K \mid \text{False} \mid \exists \boldsymbol{n}.\boldsymbol{K} \mid \exists \boldsymbol{x}.\boldsymbol{K} & & \textbf{logical formulæ}
\end{array}
$$

Definition 5.25 ($\mathrm{CL}_{\mathrm{tree}}$ forcing semantics)**.** The semantics of $\mathrm{CL}_{\mathrm{tree}}$ is an extension of the semantics of $\mathrm{CL}_{\emptyset}$ for the model $Tree^{\mathcal{S}}$. Given an interpretation function $\sigma : (\mathcal{V}_{\mathcal{D}} \to \mathcal{P}(\mathcal{S} \times \mathcal{T})) \times (\mathcal{V}_{\mathcal{C}} \to \mathcal{P}(\mathcal{S} \times \mathcal{C}))$ mapping propositional variables to store-indexed tree and context sets, the forcing semantics is given by two satisfaction relations $\sigma, \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} P$ and $\sigma, \mathsf{s}, \mathsf{c} \vDash_{\mathcal{C}} K$ for tree and tree contexts with variable stores, defined as in Defn. 4.12 and extended as follows:

$$
\begin{array}{ll}
\sigma, \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} x & \Leftrightarrow \mathsf{t} \equiv \mathsf{s}(x) \\
\sigma, \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} \exists n.P & \Leftrightarrow \exists \mathsf{n} \in \mathcal{N}.\ \sigma, [\mathsf{s}|n \leftarrow \mathsf{n}], \mathsf{t} \vDash_{\mathcal{T}} P \\
\sigma, \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} \exists x.P & \Leftrightarrow \exists \mathsf{t}' \in \mathcal{T}.\ \sigma, [\mathsf{s}|x \leftarrow \mathsf{t}'], \mathsf{t} \vDash_{\mathcal{T}} P \\
\\
\sigma, \mathsf{s}, \mathsf{c} \vDash_{\mathcal{C}} n[K] & \Leftrightarrow \exists \mathsf{n} \in \mathcal{N}, \mathsf{c}' \in \mathcal{C}.\ (\mathsf{n} = \mathsf{s}(n) \wedge \sigma, \mathsf{s}, \mathsf{c}' \vDash_{\mathcal{C}} K \wedge \mathsf{c} \equiv \mathsf{n}[\mathsf{c}']) \\
\sigma, \mathsf{s}, \mathsf{c} \vDash_{\mathcal{C}} P \mid K & \Leftrightarrow \exists \mathsf{t} \in \mathcal{T}, \mathsf{c}' \in \mathcal{C}.\ (\sigma, \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} P \wedge \sigma, \mathsf{s}, \mathsf{c}' \vDash_{\mathcal{C}} K \wedge \mathsf{c} \equiv \mathsf{t} \mid \mathsf{c}') \\
\sigma, \mathsf{s}, \mathsf{c} \vDash_{\mathcal{C}} \exists n.K & \Leftrightarrow \exists \mathsf{n} \in \mathcal{N}.\ \sigma, [\mathsf{s}|n \leftarrow \mathsf{n}], \mathsf{c} \vDash_{\mathcal{C}} K \\
\sigma, \mathsf{s}, \mathsf{c} \vDash_{\mathcal{C}} \exists x.K & \Leftrightarrow \exists \mathsf{t}' \in \mathcal{T}.\ \sigma, [\mathsf{s}|x \leftarrow \mathsf{t}'], \mathsf{c} \vDash_{\mathcal{C}} K
\end{array}
$$

As before, we define the derived formulæ $\Diamond$ and $*$. The somewhere modality $\Diamond$ allows us to reason about properties satisfied by subtrees arbitrarily deep inside a tree: $\Diamond P$ holds for any tree with a subtree satisfying P. The star connective $*$, meanwhile, describes an interesting way of combining two trees: while the projection $K \cdot 0$ describes placing an empty tree in the hole of a context satisfying K, the reverse embedding $(0 \rhd P)$ describes adding a hole at some arbitrary location in a tree satisfying P; thus $P_1 * P_2$, which is defined as $(0 \rhd P_1) \cdot P_2$, describes the insertion of a tree satisfying P_2 into an arbitrary location inside a tree satisfying P_1. Note that this operation is neither commutative nor associative.

We also derive the adjoints of both composition and branching, like we did for sequence concatenation. Finally, as for heaps, the validity formula in Defn. 3.15 describes validity for all trees with respect to a specific store. This can be used to express equality for both tree and node variables.

Definition 5.26 (Spatial adjoints)**.** The adjoints of composition and branching are defined as follows:

$$P_1 \dashv P_2 \triangleq (P_1 \mid I) \lhd P_2 \qquad P \mathbin{@} n \triangleq n[I] \lhd P$$

and have the following derived semantics:

$$\sigma, \mathsf{s}, \mathsf{t} \vDash P_1 \dashv P_2 \Leftrightarrow \forall \mathsf{t}' \in \mathcal{T}.(\sigma, \mathsf{s}, \mathsf{t}' \vDash P_1 \wedge (\mathsf{t}' \mid \mathsf{t})\downarrow \Rightarrow \sigma, \mathsf{s}, \mathsf{t} \mid \mathsf{t}' \vDash P_2)$$
$$\sigma, \mathsf{s}, \mathsf{t} \vDash P \mathbin{@} n \quad \Leftrightarrow \forall \mathsf{n} \in \mathcal{N}.(\mathsf{n} = \mathsf{s}(n) \wedge \mathsf{n} \notin \mathrm{locs}(\mathsf{t}) \Rightarrow \sigma, \mathsf{s}, \mathsf{n}[\mathsf{t}] \vDash P)$$

Definition 5.27 (Variable equality)**.** Tree and name variable equality are defined as follows, using the validity connective from Defn. 3.15:

$$(x_1 = x_2) \triangleq \mathrm{valid}_i(x_1 \Leftrightarrow x_2)$$
$$(n_1 = n_2) \triangleq \mathrm{valid}_i(n_1[0] \Leftrightarrow n_2[0])$$

and have the following derived semantics:

$$\sigma, \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} x_1 = x_2 \Leftrightarrow \mathsf{s}(x_1) \equiv \mathsf{s}(x_2)$$
$$\sigma, \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} n_1 = n_2 \Leftrightarrow \mathsf{s}(n_1) \equiv \mathsf{s}(n_2)$$

Below are some examples of $\mathrm{CL}_{\mathrm{tree}}$ formulæ. Additional examples will be considered when reasoning about tree update in Chapter 6.

Example 5.28 ($\mathrm{CL}_{\mathrm{tree}}$ formulæ)**.** The following are examples of formulæ in $\mathrm{CL}_{\mathrm{tree}}$, together with their interpretation on trees. It is assumed throughout that n is the current store value of n and m is the current store value of m.

(a) $n[0]$

the tree $\mathsf{n}[0]$.

(b) $n[\mathrm{true}]$

a tree with root node n.

(c) $\Diamond n[\mathrm{true}]$ *or* $\mathrm{True} \cdot n[\mathrm{true}]$ *or* $\mathrm{true} * n[\mathrm{true}]$

a tree containing a node n.

(d) $n[0] * \text{true}$ *or* $(0 \rhd n[0]) \cdot \text{true}$ *or* $n[\text{true}] \vee (n[0] \mid \text{true})$

a tree with root node n and either a subforest or siblings.

(e) $(n[\text{true}] \blacktriangleright P) \cdot 0$

a tree into which it is possible to add a subtree with root node n to obtain a tree satisfying P; in other words, the result of deleting the subtree at n from a tree satisfying P.

(f) $(0 \rhd P) \cdot n[\text{true}]$ *or* $P * n[\text{true}]$

a tree containing n that would satisfy P were the subtree at n replaced by a 0; in other words, the result of 'restoring' the subtree at n to a tree satisfying P.

(g) $\exists x, y.\, (n[x \mid m[y]] \rhd P) \cdot (n[x] * m[y])$

a tree that contains the distinct locations n and m, where m is not a direct ancestor of n, and which satisfies P whenever the subtree at m is moved to be underneath n; this corresponds to the weakest precondition of the command that moves the subtree at m to n, as described in Chapter 8.

Like for the previous logics, we consider the interpretation of the formula classes in $\mathrm{CL}_{\mathrm{tree}}$. The pure, exact and ubiquitous classes express the same properties as for heaps, signifying respectively independence from the tree value, satisfaction by at most one tree or context, and satisfaction by $-$ or 0. The interpretation of preciseness, however, changes again. Unlike in the heap model, there are no pointers and composition is not ordered. However, the presence of nesting together with unique identifiers provides other ways of uniquely splitting trees into contexts and subtrees. This can be illustrated with three examples, all using a node identifier n to determine a splitting. The precise tree formula $n[\text{true}]$ splits off the subtree with root node n. The precise context formula $\text{True} \circ n[I]$ (expressed here using context composition but see next paragraph) uses a context with a hole at n to split off the subforest beneath n. Finally, the precise formula pair $(\text{True} \circ n[\text{true} \mid I], 0)$ splits off an empty tree directly underneath n. All three examples can be used to specify tree update commands, replacing the specified subtrees by another. For example, the first two examples can be used in tree disposal, replacing the subtree at or beneath n by 0. Similarly, the last example can be used to specify a location for adding new tree nodes, replacing the selected empty tree by $\mathsf{m}[0]$ for some fresh label m.

Finally, we consider the expressibility of context composition and its adjoints in

$\mathrm{CL}_{\mathrm{tree}}$. Interestingly, these are now derivable due to presence of quantification over variables, which allows reasoning about all possible behaviours of context application. Thus $K_1 \circ K_2$ can be expressed as $\forall x.\, x \rhd (K_1 \cdot (K_2 \cdot x))$ or $\forall n.\, n[0] \rhd (K_1 \cdot (K_2 \cdot n[0]))$. Note the similarity to the derivations in Lemma 4.8: universal quantification is used here to emulate the omniplaceability condition in the Lemma for the precise formulæ $n[0]$ and x (or, more accurately, $(x \wedge \neg 0)$ since x by itself is not precise). Similarly, $K_1 \multimap K_2$ can be expressed as $\forall n.\, (K_1 \cdot n[0]) \rhd (K_2 \cdot n[0])$ and $K_1 \circ\!\!- K_1$ as $\forall n.\, n[0] \rhd (K_1 \lhd (K_2 \cdot n[0]))$, likewise corresponding to the definitions in Lemma 4.8.

5.6 Terms

The final example in this chapter consists of simple terms. These are constructed from a set of function symbols $\mathcal{F}$ with fixed arities. The model also includes a variable store, where variables take terms as values. This, together with an expression language consisting of terms that can contain variables to represent subterms, serves as an important step towards specifying term rewrites. In Chapter 7, a slightly more complicated term model will be considered, and used for reasoning about a local term rewriting language.

Terms are an interesting example of structured data since they do not generally contain a zero element, in the sense of Section 4.2. The fixed arities of function symbols mean that there can be no 'invisible' empty term as for heaps and trees, although there are constant terms of zero arity. Assuming there is more than one such constant, then no set of terms can be both omnipresent and exact. An omnipresent set must contain all the zero arity constants; an exact set, meanwhile, must be a singleton. In the case where there is just one constant, then that constant does function as a zero.

Definition 5.29 (Terms)**.** Given a set $\mathcal{F} = \{\mathsf{f}, \dots\}$ of function symbols and a signature $\Sigma : \mathcal{F} \to \mathbb{N}$ mapping function symbols to arities (where at least one function symbol has arity zero), terms $\mathsf{t} \in \mathsf{T}$ and term contexts $\mathsf{c} \in \mathcal{C}$ are defined by the following grammars:

$$\begin{aligned} \mathsf{t} &::= \mathsf{f}(\mathsf{t}_1, \dots, \mathsf{t}_k) && k = \Sigma(\mathsf{f}) \\ \mathsf{c} &::= - \mid \mathsf{f}(\mathsf{t}_1, \dots \mathsf{t}_{i-1}, \mathsf{c}, \mathsf{t}_{i+1}, \dots, \mathsf{t}_k) && k = \Sigma(\mathsf{f}) \geq i \geq 1 \end{aligned}$$

The insertion of terms into term contexts is given by the following total application

function:

$$\mathrm{ap}(-, \mathsf{t}) = \mathsf{t}$$

$$\mathrm{ap}(\mathsf{f}(\mathsf{t}_1, \ldots, \mathsf{c}, \ldots, \mathsf{t}_k), \mathsf{t}) = \mathsf{f}(\mathsf{t}_1, \ldots, \mathrm{ap}(\mathsf{c}, \mathsf{t}), \ldots, \mathsf{t}_k)$$

Definition 5.30 (Variable store)**.** Given an infinite set $\mathrm{Var}_{\mathsf{T}} = \{x, \ldots\}$ of term variables, a *store* $\mathsf{s} \in \mathcal{S}$ is a total function $\mathsf{s} : \mathrm{Var}_{\mathsf{T}} \to \mathsf{T}$, returning the values of those term variables.

Definition 5.31 (Expressions)**.** An *expression* E is a term that can also contain term variables, and is defined by the following grammar:

$$E ::= x \mid \mathsf{f}(E_1, \ldots E_k) \quad k = \Sigma(\mathsf{f})$$

The valuation of an expression in a store, written $[\![E]\!]\mathsf{s}$, is given by:

$$[\![x]\!]\mathsf{s} = \mathsf{s}(x)$$

$$[\![\mathsf{f}(E_1, \ldots E_k)]\!]\mathsf{s} = \mathsf{f}([\![E_1]\!]\mathsf{s}, \ldots [\![E_k]\!]\mathsf{s})$$

It is easy to show that $Term = (\mathsf{T}, \mathcal{C}, \mathrm{ap}, \{-\})$ forms a model of CL (without a zero). Like in the heap and tree cases, the variable store can be included in the model using the indexing construction of Example 3.8. The logic is extended with special formulæ corresponding to the data structure definitions. Similarly to before, adding a context connective $\mathsf{f}(P_1, \ldots, K, \ldots, P_k)$ allows us to define a term connective $\mathsf{f}(P_1, P_2, \ldots, P_k)$ as $\mathsf{f}(I, P_2, \ldots, P_k) \cdot P_1$, while zero-arity terms must be expressed directly. Term variables are incorporated in the logic by adding expressions E. Since these form a superset of terms, they can also be used to express the zero-arity terms. Finally, quantification is also included.

Definition 5.32 ($\mathrm{CL}_{\mathrm{term}}$ formulæ)**.** The formulæ of Context Logic for Terms ($\mathrm{CL}_{\mathrm{term}}$) consist of the same context and data assertions as CL with the addition of term contexts, expressions and quantification over variables:

$P ::= p$	propositional variables
$\mid \boldsymbol{E}$	**special formulæ**
$\mid K \cdot P \mid K \lhd P$	structural formulæ
$\mid P \Rightarrow P \mid \mathrm{false} \mid \exists \boldsymbol{x}.\boldsymbol{P}$	**logical formulæ**
$K ::= k$	propositional variables
$\mid \mathsf{f}(\boldsymbol{P_1}, \ldots, \boldsymbol{P_{i-1}}, \boldsymbol{K}, \boldsymbol{P_{i+1}}, \ldots, \boldsymbol{P_k})$	**special formulæ**
$\mid I \mid P \rhd P$	structural formulæ
$\mid K \Rightarrow K \mid \mathrm{False} \mid \exists \boldsymbol{x}.\boldsymbol{K}$	**logical formulæ**

where $k = \Sigma(\mathsf{f}) \geq i \geq 1$.

Definition 5.33 ($\mathrm{CL}_{\mathrm{term}}$ forcing semantics)**.** The semantics of $\mathrm{CL}_{\mathrm{term}}$ is an extension of the semantics of CL for the model $Term^{\mathcal{S}}$. Given an interpretation function $\sigma : (\mathcal{V}_{\mathcal{D}} \to \mathcal{P}(\mathcal{S} \times \mathsf{T})) \times (\mathcal{V}_{\mathcal{C}} \to \mathcal{P}(\mathcal{S} \times \mathcal{C}))$ mapping propositional variables to store-indexed term and context sets, the forcing semantics is given by two satisfaction relations $\sigma, \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} P$ and $\sigma, \mathsf{s}, \mathsf{c} \vDash_{\mathcal{C}} K$ for term and term contexts with variable stores, defined as in Defn. 3.4 and extended as follows:

$$\sigma, \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} E \quad \Leftrightarrow \mathsf{t} = [\![E]\!]\mathsf{s}$$

$$\sigma, \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} \exists x.P \Leftrightarrow \exists \mathsf{t}' \in \mathsf{T}.\ \sigma, [\mathsf{s}|x \leftarrow \mathsf{t}'], \mathsf{t} \vDash_{\mathcal{T}} P$$

$$\sigma, \mathsf{s}, \mathsf{c} \vDash_{\mathcal{C}} \mathsf{f}(P_1, \ldots, P_{i-1}, K, P_{i+1}, \ldots, P_k)$$
$$\Leftrightarrow \exists_{\substack{1 \leq j \leq k \\ j \neq i}} \mathsf{t_j} \in \mathsf{T}, \mathsf{c} \in \mathcal{C}.\ (\mathsf{t} = \mathsf{f}(\mathsf{t_1}, \ldots, \mathsf{c}, \ldots, \mathsf{t_k}) \wedge \sigma, \mathsf{s}, \mathsf{c} \vDash_{\mathcal{C}} K \wedge \bigwedge_{\substack{1 \leq j \leq k \\ j \neq i}} \sigma, \mathsf{s}, \mathsf{t_j} \vDash_{\mathcal{T}} P_j)$$

$$\sigma, \mathsf{s}, \mathsf{c} \vDash_{\mathcal{C}} \exists x.K \Leftrightarrow \exists \mathsf{t} \in \mathsf{T}.\ \sigma, [\mathsf{s}|x \leftarrow \mathsf{t}], \mathsf{c} \vDash_{\mathcal{C}} K$$

where $k = \Sigma(\mathsf{f}) \geq i \geq 1$.

The same derived formulæ can be defined for $\mathrm{CL}_{\mathrm{term}}$ as for sequences and trees, with the exception of the $*$ connective, which depends on the presence of zero. This illustrates the fact that, unlike for sequences or trees, there is no general way of combining two terms without adding or removing function symbols. Below are some examples of $\mathrm{CL}_{\mathrm{term}}$ formulæ.

Example 5.34 ($\mathrm{CL}_{\mathrm{term}}$ formulæ)**.** Consider a function symbol set $\mathcal{F} = \mathbb{N} \cup \{-, +, \times\}$, where the naturals $\mathbb{N}$ are constants of arity zero, $-$ represents the unary minus and has arity one, and $+$ and $\times$ have arity two. Then the following are examples of $\mathrm{CL}_{\mathrm{term}}$ formulæ, together with their interpretations on terms.

(a) $\times(2, +(1,3))$ *or* $\times(I, +(1,3)) \cdot 2$ *or* $\times(2, I) \cdot +(1,3)$
the multiplication term $\times(2, +(1,3))$.

(b) $+(\mathrm{true}, \mathrm{true})$
any addition term.

(c) $\Diamond{-}({-}(\mathrm{true}))$
a term that contains a double negation.

(d) $\mathrm{nat} \triangleq \neg((\neg I) \cdot \mathrm{true})$
any zero-arity term: that is, a natural number.

(e) $\Box(\,+(\text{true},\text{true}) \vee (-(I) \vee I)\cdot\text{nat}\,)$

a sum of positive or negative integers.

(f) $\exists x.\,(\times(2,x) \rhd P) \cdot +(x,x)$

a term that contains an addition $+(x,x)$, for some x, and which would satisfy P were the addition replaced by the multiplication $\times(2,x)$; in other words, this describes the result of performing a single rewrite on a term satisfying P, turning one multiplication by 2 into an addition.

Finally, we consider the formula classes for CL_{term}. As for trees, pureness specifies that the formula depends only on the variable store, while exactness specifies satisfaction by at most one term or context for any given store. Ubiquitous data formulæ are the ones satisfied by all the zero-arity terms. Finally, preciseness is similar to the sequence case. The lack of unique identifiers means that there are no precise term formulæ apart from false, as any subterm can occur multiple times inside a term, and hence does not uniquely specify a splitting. Precise context formulæ, meanwhile, correspond to satisfaction by incompatible contexts, where two contexts are incompatible if there are no two subterms that can be placed in them obtain the same result. Thus, for example, the formula $+(\text{true}, I)$ is precise.

Part B

Tree Update

Chapter 6

Basic Tree Update

This chapter introduces Context Logic-based local reasoning about a basic tree update language. It presents the language, the local reasoning framework, and an axiomatisation made up of 'Small Axioms'. It also proves the completeness of the program logic by deriving the weakest preconditions of the commands, and discusses an interesting update issue specific to tree update.

6.1 Basic Update Language

This section presents a basic high-level language for manipulating tree structures. Both the trees and the language considered here are intentionally simple: the trees have node identifiers but no additional data; the language manipulates whole trees using these identifiers, but contains no advanced control structures or queries. The resulting framework is easy to present, while still being complex enough to demonstrate the key points concerning the use of Context Logic for reasoning locally about tree update. A later chapter will extend both the tree data model, incorporating node labels and pointers, and the update language, adding various features including path queries and more control structures.

The trees considered in this chapter are the simple trees defined in Chapter 5. These consist of a forest structure with unique node identifiers ranging over an alphabet $\mathcal{N}$, which is constructed using a branching operator $\mathsf{n}[-]$ for $\mathsf{n} \in \mathcal{N}$ and a commutative composition operator $|$. We also consider tree contexts, which are trees with a hole inside, and a variable store containing both name variables and tree variables, which take names and trees as values. The presence of tree variables allows

the manipulation of whole trees in the language.

Definition 6.1 (Trees)**.** Trees $\mathtt{t} \in \mathcal{T}$, tree contexts $\mathtt{c} \in \mathcal{C}$ and variable stores $\mathtt{s} \in \mathcal{S}$ are defined as in Defns. 5.22 and 5.23.

The update language considered in this chapter is a simple variable-based language with memory faults, and is based closely on standard low-level heap update languages, such as the one in [ORY01]. The language is also local, in that the effect of a command on a tree does not depend on the tree's context. This locality condition is quite subtle and is explained thoroughly in the following section. The language consists of three basic types of command: variable assignment, tree updates, and command sequencing. Sequencing is standard. Variable assignment consists of assigning values to name and tree variables. For simplicity, the right-hand-side expressions representing these values are restricted to variables, rather than literal constants or more complex expressions. This corresponds to the view that the precise values of node identifiers are an issue of internal representation and should not be visible to the user; it is the high-level tree structure that is important. Consequently, the ultimate origin of variable values is always indirect, coming either from a tree lookup or an execution of the new command, which creates nodes with fresh identifiers.

The third type of command, the tree update commands, describes updates that act at a specific location in the tree, which is specified using a name variable. The types of tree updates correspond to the standard types of heap updates, namely disposal, update, lookup and new; the difference is that the tree updates act at the tree level rather than at the node level. Thus, for example, the tree dispose command removes an entire subtree at a given location, as opposed to a single node in the heap. Similarly, tree lookup obtains the value of a subtree at a location, rather than the value of a pointer target. The tree command corresponding to heap update adds a new subtree at a location, instead of updating the pointer. Here we have a choice: we can either replace the subtree already at the location, or we can append to it. We choose the latter option, since replacement is expressible as a disposal followed by an append. Finally, the new command is similar to its heap equivalent, in that it generates a single node in the tree. Unlike the heap case, however, the location of the new node has to be given. The reason for generating a single node, rather than a more complex subtree structure, is merely to keep the language simple; complicated structures can always be generated by repeated calls.

One consequence of the nested nature of the tree data structure is an ambiguity when specifying the location for an update. An update at a given node can reasonably take place at one of two locations: at the level of the node itself or at the level of its subforest. Hence, the dispose command can remove the entire tree at a given node, or just its subtree. Similarly, the append command can add a tree as a sibling to a given node, or as a sibling to its subforest. Since neither of these options is derivable from the other, both are given in the language, by subscripting update commands with either a T (tree) or SF (subforest) marker.

Definition 6.2 (BTU commands)**.** The commands of the Basic Tree Update Language (BTU) are defined formally by the following grammar:

$$\begin{array}{lll} \mathbb{C} ::= & n := n' \mid x := x' & \text{variable assignment} \\ & \mathbb{C}_{\text{up}}(n) & \text{update at location } n \\ & \mathbb{C} \,;\, \mathbb{C} & \text{sequencing} \end{array}$$

The tree updates $\mathbb{C}_{\text{up}}(n)$, which act at a location n, are defined as follows, with each update having two variants, corresponding to an action at the root of a tree and an action at its subforest:

$$\begin{array}{llll} \mathbb{C}_{\text{up}}(n) ::= & [n]_{\text{T}} := 0 & [n]_{\text{SF}} := 0 & \text{dispose} \\ & [n]_{\text{T}} *= x & [n]_{\text{SF}} *= x & \text{append} \\ & x := [n]_{\text{T}} & x := [n]_{\text{SF}} & \text{lookup} \\ & n' := \text{new } [n]_{\text{T}} & n' := \text{new } [n]_{\text{SF}} & \text{new} \end{array}$$

Before giving the formal semantics of the update language, it is necessary to briefly remention the issue of locality. As previously stated, the program logic considered in this work is based on the concept of local reasoning. This approach, which is discussed formally in the following sections, can be best summarised by the following statement:

> "To understand how a program works, it should be possible for reasoning and specification to be confined to the [parts of the data structure] that the program actually accesses. The value of any other [parts] will remain unchanged." [ORY01]

One underlying assumption of local reasoning is that this assertion holds: namely, that there exists a well-specified part of the data structure that a program accesses (a

'footprint'), and that commands behave in such a way that reasoning and specification can be confined to just that part, with the remaining data structure unchanging. This assumption gives a behavioural restriction on permissible commands, defined formally in Defn. 6.10. For now, it is enough to use the intuition given above: a local command is one whose behaviour is completely determined by its action on its footprint; in other words, its local behaviour determines its global behaviour.

One consequence of restricting reasoning to local commands is the obligatory use of memory faults for commands that try to access data that is not there. The dispose command, for example, must fault when asked to dispose of a location n that is not in the current tree, as opposed to (for instance) doing nothing. Otherwise, the assumption that global behaviour is completely determined by local action would be false: the behaviour in a small environment (without node n, where nothing happens) would not determine the execution in a larger one (with n, where the subtree at n is deleted). A similar argument holds for the other update commands.

Another, less natural consequence of the locality restriction involves the behaviour of the append command. The presence of unique node identifiers means that appending is in general a partial operation: if two trees share any identifiers then it is not possible to append one inside the other. Moreover, whether or not an append operation is possible is not determined by a local memory footprint at the target location, but by the nodes of the entire tree. Thus an append operation may be possible in one environment but not in a larger one, implying non-locality. This can be superficially fixed by having an undefined append diverge rather than fault, since the reasoning framework here is based on partial correctness. However, the problem actually demonstrates that the current append command is somewhat unnatural, as it forces the programmer to consider all the possible node values in the tree, including those generated non-deterministically using the new command. A more natural approach is to rename the node identifiers of an appended tree; after all, it is the high-level tree structure that matters, not the precise node names. This solution is considered at the end of this chapter. However, the simpler, potentially divergent behaviour is sufficient for the current exposition.

Definition 6.3 (BTU operational semantics)**.** The operational semantics of BTU is given in Figure 6.1 using an evaluation relation $\rightsquigarrow$ relating configuration triples $\mathbb{C}, \mathsf{s}, \mathsf{t}$, terminal states s, t and faults 'fault'.

$$\frac{\mathsf{s}(n') \equiv \mathsf{n}'}{n := n', \mathsf{s}, \mathsf{t} \rightsquigarrow [\mathsf{s}|n \leftarrow \mathsf{n}'], \mathsf{t}} \qquad \frac{\mathsf{s}(x') \equiv \mathsf{t}'}{x := x', \mathsf{s}, \mathsf{t} \rightsquigarrow [\mathsf{s}|x \leftarrow \mathsf{t}'], \mathsf{t}}$$

$$\frac{\mathsf{s}(n) = \mathsf{n} \quad \mathsf{t} \equiv \mathrm{ap}(\mathsf{c}, \mathsf{n}[\mathsf{t}'])}{[n]_{\mathrm{T}} := 0, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathsf{s}, \mathrm{ap}(\mathsf{c}, 0)} \qquad \frac{\mathsf{s}(n) = \mathsf{n} \quad \mathsf{t} \equiv \mathrm{ap}(\mathsf{c}, \mathsf{n}[\mathsf{t}'])}{[n]_{\mathrm{SF}} := 0, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathsf{s}, \mathrm{ap}(\mathsf{c}, \mathsf{n}[0])}$$

$$\frac{\mathsf{s}(n) = \mathsf{n} \quad \mathsf{t} \equiv \mathrm{ap}(\mathsf{c}, \mathsf{n}[\mathsf{t}']) \quad \mathsf{s}(x) \equiv \mathsf{t}'' \quad \mathsf{t}'' \# \mathsf{t}}{[n]_{\mathrm{T}} \mathrel{*}= x, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathsf{s}, \mathrm{ap}(\mathsf{c}, \mathsf{n}[\mathsf{t}' \mid \mathsf{t}''])}$$

$$\frac{\mathsf{s}(n) = \mathsf{n} \quad \mathsf{t} \equiv \mathrm{ap}(\mathsf{c}, \mathsf{n}[\mathsf{t}']) \quad \mathsf{s}(x) \equiv \mathsf{t}'' \quad \mathsf{t}'' \# \mathsf{t}}{[n]_{\mathrm{SF}} \mathrel{*}= x, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathsf{s}, \mathrm{ap}(\mathsf{c}, \mathsf{n}[\mathsf{t}' \mid \mathsf{t}''])}$$

$$\frac{\mathsf{s}(n) = \mathsf{n} \quad \mathsf{t} \equiv \mathrm{ap}(\mathsf{c}, \mathsf{n}[\mathsf{t}'])}{x := [n]_{\mathrm{T}}, \mathsf{s}, \mathsf{t} \rightsquigarrow [\mathsf{s}|x \leftarrow \mathsf{n}[\mathsf{t}']], \mathsf{t}} \qquad \frac{\mathsf{s}(n) = \mathsf{n} \quad \mathsf{t} \equiv \mathrm{ap}(\mathsf{c}, \mathsf{n}[\mathsf{t}'])}{x := [n]_{\mathrm{SF}}, \mathsf{s}, \mathsf{t} \rightsquigarrow [\mathsf{s}|x \leftarrow \mathsf{t}'], \mathsf{t}}$$

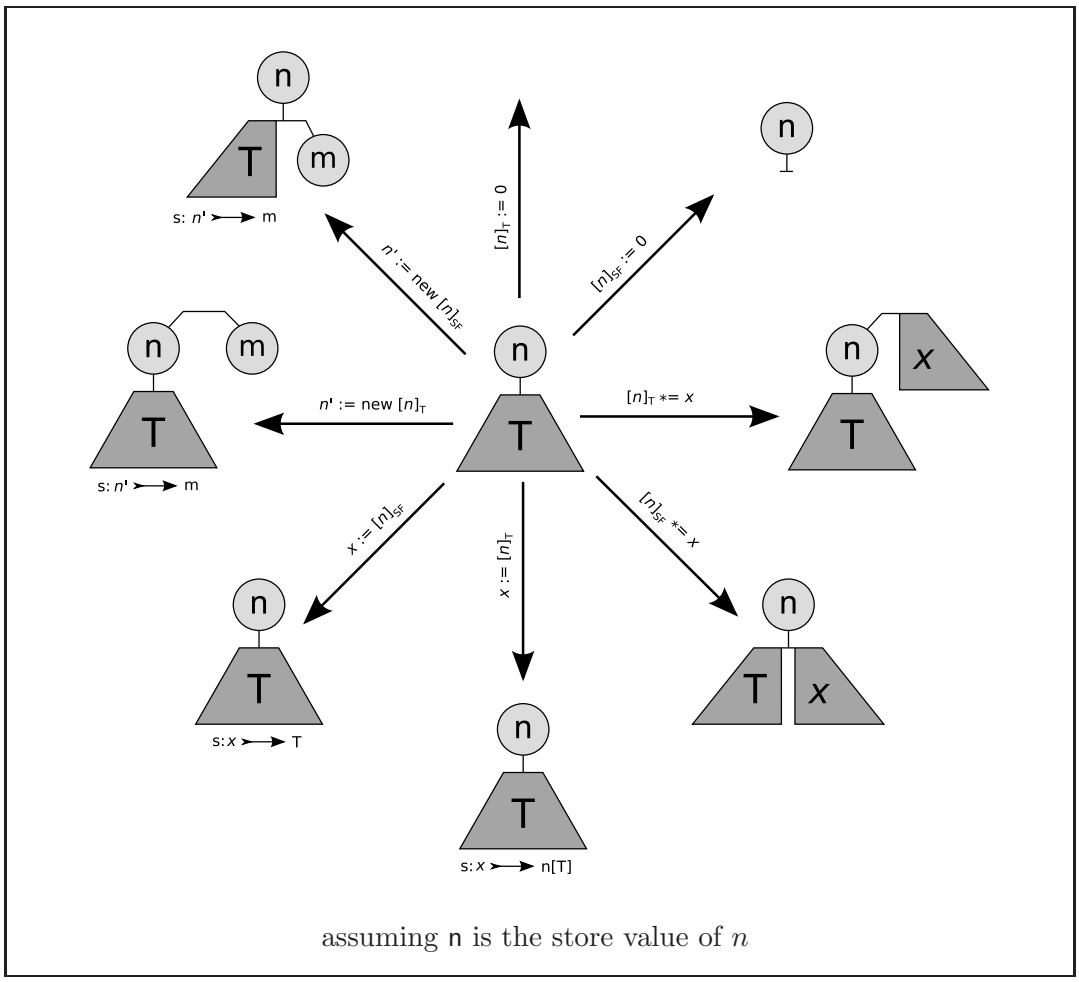
$$\frac{\mathsf{s}(n) = \mathsf{n} \quad \mathsf{t} \equiv \mathrm{ap}(\mathsf{c}, \mathsf{n}[\mathsf{t}']) \quad \mathsf{n}' \notin \mathrm{locs}(\mathsf{t})}{n' := \mathrm{new}\ [n]_{\mathrm{T}}, \mathsf{s}, \mathsf{t} \rightsquigarrow [\mathsf{s}|n' \leftarrow \mathsf{n}'], \mathrm{ap}(\mathsf{c}, \mathsf{n}[\mathsf{t}'] \mid \mathsf{n}'[0])}$$

$$\frac{\mathsf{s}(n) = \mathsf{n} \quad \mathsf{t} \equiv \mathrm{ap}(\mathsf{c}, \mathsf{n}[\mathsf{t}']) \quad \mathsf{n}' \notin \mathrm{locs}(\mathsf{t})}{n' := \mathrm{new}\ [n]_{\mathrm{SF}}, \mathsf{s}, \mathsf{t} \rightsquigarrow [\mathsf{s}|n' \leftarrow \mathsf{n}'], \mathrm{ap}(\mathsf{c}, \mathsf{n}[\mathsf{t}' \mid \mathsf{n}'[0]])}$$

$$\frac{\mathsf{s}(n) = \mathsf{n} \quad \mathsf{t} \not\equiv \mathrm{ap}(\mathsf{c}, \mathsf{n}[\mathsf{t}'])}{\mathbb{C}_{\mathrm{up}}(n), \mathsf{s}, \mathsf{t} \rightsquigarrow \mathrm{fault}} \qquad \frac{\mathbb{C}_1, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathrm{fault}}{(\mathbb{C}_1\ ;\ \mathbb{C}_2), \mathsf{s}, \mathsf{t} \rightsquigarrow \mathrm{fault}}$$

$$\frac{\mathbb{C}_1, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathbb{C}', \mathsf{s}', \mathsf{t}'}{(\mathbb{C}_1\ ;\ \mathbb{C}_2), \mathsf{s}, \mathsf{t} \rightsquigarrow (\mathbb{C}'\ ;\ \mathbb{C}_2), \mathsf{s}', \mathsf{t}'} \qquad \frac{\mathbb{C}_1, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathsf{s}', \mathsf{t}'}{(\mathbb{C}_1\ ;\ \mathbb{C}_2), \mathsf{s}, \mathsf{t} \rightsquigarrow \mathbb{C}_2, \mathsf{s}', \mathsf{t}'}$$

Figure 6.1: BTU Operational Semantics



assuming n is the store value of n

Figure 6.2: BTU Update Examples

This definition is a straightforward formalisation of the descriptions above, similar in style to the operational semantics given in [YO02]. There are, however, a few points worth noting. The first is the use of contexts to describe the semantics of located updates: in general, an update at a location n is described as turning a tree of the form $\mathrm{ap}(\mathsf{c}, \mathsf{n}[\mathsf{t}])$ into one of the form $\mathrm{ap}(\mathsf{c}, \mathsf{t}')$, where t' depends on the precise command. Meanwhile, if the location n is not in the tree then the update faults. As explained above, append is defined as a partial operation, diverging if the locations in the tree being appended are not disjoint from the ones in the target tree. Finally, the new command is defined non-deterministically, generating an arbitrary fresh location in the tree and storing it in some name variable.

Example 6.4 (BTU commands)**.** The behaviour of the BTU update commands on a simple tree of the form $\mathsf{n}[\mathsf{T}]$ is illustrated in Figure 6.2.

Example 6.5 (Move)**.** The following BTU program moves a subtree at a location n, placing it underneath location n'. The command faults if either of the locations is not in the tree, or if n' is underneath location n, but is total otherwise.

$$\begin{aligned} \mathrm{move}(n, n') \triangleq\ & x := [n]_{\mathrm{T}}\ ; && \text{store subtree at } n \\ & [n]_{\mathrm{T}} := 0\ ; && \text{dispose it} \\ & [n']_{\mathrm{SF}} *= x && \text{append it at } n' \end{aligned}$$

Finally, it is also useful to consider two behavioural properties of the commands in the language: the set of variables that are modified by a given command, and the set of variables which can interfere in the execution of the command. The sets are defined semantically, but can usually be calculated, or closely approximated, using a simple syntactic check, and will be used in specifying and reasoning about the commands in the program logic.

Definition 6.6 (BTU modified variables)**.** The set of *modified variables* $\mathrm{mod}(\mathbb{C})$ of a command $\mathbb{C}$ is defined by:

$$\begin{aligned} x \in \mathrm{mod}(\mathbb{C}) &\Leftrightarrow \exists \mathsf{s}, \mathsf{t}, \mathsf{s}', \mathsf{t}'.\, (\mathbb{C}, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathsf{s}', \mathsf{t}') \wedge \mathsf{s}(x) \neq \mathsf{s}'(x) \\ n \in \mathrm{mod}(\mathbb{C}) &\Leftrightarrow \exists \mathsf{s}, \mathsf{t}, \mathsf{s}', \mathsf{t}'.\, (\mathbb{C}, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathsf{s}', \mathsf{t}') \wedge \mathsf{s}(n) \neq \mathsf{s}'(n) \end{aligned}$$

For BTU, a close syntactic over-approximation of $\mathrm{mod}(\mathbb{C})$ is $\{x\}$ for tree assignment and lookup, $\{n\}$ for location assignment and new, $\emptyset$ for the other located update commands, and $\mathrm{mod}(\mathbb{C}_1) \cup \mathrm{mod}(\mathbb{C}_2)$ for $\mathbb{C}_1\ ;\ \mathbb{C}_2$. The reason that this is an over-approximation is that modification is an extensional property: thus $x := x$ actually has no modified variables, while $x := y\ ;\ y := x$ only modifies x. However, the approximation is good enough to be used in the program logic.

Definition 6.7 (BTU dependent variables)**.** The set of *dependent variables* $\mathrm{dep}(\mathbb{C})$ of a command $\mathbb{C}$ is defined by:

$$\begin{aligned} x \in \mathrm{dep}(\mathbb{C}) \Leftrightarrow\ & \exists \mathsf{s}, \mathsf{t}, \mathsf{s}', \mathsf{t}'.\, (\mathbb{C}, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathsf{s}', \mathsf{t}') \wedge \exists \mathsf{t}''.\, (\mathbb{C}, [\mathsf{s}|x \leftarrow \mathsf{t}''], \mathsf{t} \not\rightsquigarrow [\mathsf{s}|x \leftarrow \mathsf{t}''], \mathsf{t}') \\ & \vee \exists \mathsf{s}, \mathsf{t}.\, (\mathbb{C}, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathrm{fault}) \wedge \exists \mathsf{t}''.\, (\mathbb{C}, [\mathsf{s}|x \leftarrow \mathsf{t}''], \mathsf{t} \not\rightsquigarrow \mathrm{fault}) \\ n \in \mathrm{dep}(\mathbb{C}) \Leftrightarrow\ & \exists \mathsf{s}, \mathsf{t}, \mathsf{s}', \mathsf{t}'.\, (\mathbb{C}, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathsf{s}', \mathsf{t}') \wedge \exists \mathsf{n}.\, (\mathbb{C}, [\mathsf{s}|n \leftarrow \mathsf{n}], \mathsf{t} \not\rightsquigarrow [\mathsf{s}|n \leftarrow \mathsf{n}], \mathsf{t}') \\ & \vee \exists \mathsf{s}, \mathsf{t}.\, (\mathbb{C}, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathrm{fault}) \wedge \exists \mathsf{n}.\, (\mathbb{C}, [\mathsf{s}|n \leftarrow \mathsf{n}], \mathsf{t} \not\rightsquigarrow \mathrm{fault}) \end{aligned}$$

For BTU, a close syntactic over-approximation of $\mathrm{dep}(\mathbb{C})$ consists of all the free variables in the command. Again, this slight over-approximation is sufficient.

6.2 Local Hoare Reasoning

The local reasoning framework presented in this work is an extension of Hoare logic [Flo67, Hoa69] for reasoning about mutable data structures. Based closely on the approach of Separation Logic [IO01, ORY01, Rey02], this framework provides a natural way of inferring invariant properties implied by the locality of commands. The approach is resource-oriented [POY04] and based on the idea of partial state: the target of the reasoning is treated as a resource that can be split up and handled in bits, be it heap memory [ORY01], variables [BCY06], write and read permissions [BCOP05], or in this case tree structures. Considering the memory footprint of a program gives a concrete handle on the resources that a specification of the program needs to describe.

There are two key components to the local reasoning approach [YO02]. The first is the tight interpretation of specifications: a Hoare triple should mention all the resources relevant to the working of a program, so that other resources are automatically unaffected. The second is the presence of a 'Frame Rule' in the Hoare Logic that allows the inference of the invariant properties implied by this tightness. The suitability of spatial logics for describing the separation of resources mean that they make ideal assertion languages for the Hoare Logic, allowing a simple formulation of the Frame Rule. Thus, Separation Logic uses an assertion language based on Bunched Logic to express the separation of heaps into partial subheaps (where partiality is clearly expressed by the presence of dangling pointers). Here, the idea is to use the $\mathrm{CL}_{\mathrm{tree}}$ logic of Defn. 5.24 to express the separation of trees into contexts and subtrees, which can then be reasoned about separately.

The interpretation of Hoare triples in this framework is the standard one for partial correctness, with the addition of a crucial safety requirement necessary for local reasoning. The tightness condition above states that a Hoare triple should include all the resources necessary for executing a command. Since attempting to access resources that are not present results in a run-time memory fault, this is enforced by using a *fault-avoiding* interpretation of triples: a triple $\{P\}\ \mathbb{C}\ \{Q\}$ is valid only if executing $\mathbb{C}$ from a state satisfying P never faults.

Definition 6.8 (Interpretation of Hoare triples)**.** Given a BTU command $\mathbb{C}$ and two tree formulæ P, Q in $\mathrm{CL}_{\mathrm{tree}}$ (Defn. 5.24), a Hoare triple $\{P\}\,\mathbb{C}\,\{Q\}$ is said to hold iff whenever $\sigma, \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} P$ then:

$$\text{Consequence:}\quad \frac{P' \Rightarrow P \quad \{P\}\ \mathbb{C}\ \{Q\} \quad Q \Rightarrow Q'}{\{P'\}\ \mathbb{C}\ \{Q'\}}$$

$$\text{Auxiliary Variable Elimination:}\quad \frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{\exists x.\, P\}\ \mathbb{C}\ \{\exists x.\, Q\}}\ x \notin \mathrm{dep}(\mathbb{C})$$

$$\frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{\exists n.\, P\}\ \mathbb{C}\ \{\exists n.\, Q\}}\ n \notin \mathrm{dep}(\mathbb{C})$$

$$\text{Auxiliary Variable Renaming:}\quad \frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{P[x'/x]\}\ \mathbb{C}\ \{Q[x'/x]\}}\ x, x' \notin \mathrm{dep}(\mathbb{C}) \wedge x' \notin \mathrm{free}(P, Q)$$

$$\frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{P[n'/n]\}\ \mathbb{C}\ \{Q[n'/n]\}}\ n, n' \notin \mathrm{dep}(\mathbb{C}) \wedge n' \notin \mathrm{free}(P, Q)$$

$$\text{Frame Rule:}\quad \frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{K \cdot P\}\ \mathbb{C}\ \{K \cdot Q\}}\ \mathrm{mod}(\mathbb{C}) \cap \mathrm{free}(K) = \emptyset$$

Figure 6.3: BTU Inference Rules

(a) $\mathbb{C}, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathsf{s}', \mathsf{t}' \Rightarrow \sigma, \mathsf{s}', \mathsf{t}' \vDash_{\mathcal{T}} Q$ (*partial correctness*)

(b) $\mathbb{C}, \mathsf{s}, \mathsf{t} \not\rightsquigarrow$ fault (*fault avoiding interpretation*)

Just as in [ORY01], there are four basic inference rules used in the local reasoning framework. The first three are standard Hoare Logic rules: the Rule of Consequence allows the strengthening of preconditions or weakening of postconditions; Auxiliary Variable Elimination allows the elimination of auxiliary variables, where non-interference is guaranteed by the side-condition; and Auxiliary Variable Renaming allows variable renaming, with a similar non-interference condition. The final rule is the Frame Rule, which generalises the Frame Rule in [ORY01] and allows the inference of invariant properties.

Definition 6.9 (Inference Rules)**.** The inference rules for the Context Logic-based local reasoning framework are given in Figure 6.3.

Soundness for the first three rules is standard, following directly from the partial correctness interpretation of Hoare triples and the standard properties of existentials and variables. The soundness of the Frame Rule is shown in the next section.

6.3 Frame Rule and Locality

The purpose of the Frame Rule is to provide a simple way of inferring invariant properties in the program logic by exploiting the tightness of specifications. In Separation Logic, invariant 'frames' were added by the Frame Rule using the heap separation operation $*$. The rule here is a generalisation of that: instead of $*$, it uses the context application operator $\cdot$ (which is a generalisation of $*$ in the sense of Thm. 4.22). Thus, the rule states:

$$\frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{K \cdot P\}\ \mathbb{C}\ \{K \cdot Q\}} \quad \mathrm{mod}(\mathbb{C}) \cap \mathrm{free}(K) = \emptyset$$

In other words, given a specification $\{P\}\ \mathbb{C}\ \{Q\}$, it is possible to apply to both sides a context frame given by an arbitrary context formula K, as long as K does not mention any modified store variables. The rule thus allows the inference of global specifications from local ones.

The soundness of the Frame Rule relies on the update language behaving locally. A *local* command satisfies two properties: the *frame property*, which states that if a command executes safely (does not fault) in a given state, then any result of executing it in a larger state can be tracked back to some execution on the smaller state; and the *safety monotonicity property*, which states that if a command is safe in a given state, then it is safe in any larger state.

Definition 6.10 (Locality)**.** A command $\mathbb{C}$ is *local* if it satisfies the following two properties:

(a) *safety monotonicity*: $\mathbb{C}, \mathsf{s}, \mathsf{t} \not\rightsquigarrow \mathrm{fault} \wedge \mathrm{ap}(\mathsf{c}, \mathsf{t})\downarrow \Rightarrow \mathbb{C}, \mathsf{s}, \mathrm{ap}(\mathsf{c}, \mathsf{t}) \not\rightsquigarrow \mathrm{fault}$;

(b) *frame property*: $\mathbb{C}, \mathsf{s}, \mathsf{t} \not\rightsquigarrow \mathrm{fault} \wedge \mathrm{ap}(\mathsf{c}, \mathsf{t})\downarrow \wedge\ \mathbb{C}, \mathsf{s}, \mathrm{ap}(\mathsf{c}, \mathsf{t}) \rightsquigarrow \mathsf{s}', \mathsf{t}'$ then $\exists \mathsf{t}''$ such that $\mathbb{C}, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathsf{s}', \mathsf{t}'' \wedge \mathsf{t}' \equiv \mathrm{ap}(\mathsf{c}, \mathsf{t}'')$.

Theorem 6.11 (Soundness of Frame Rule)**.** *If a command* $\mathbb{C}$ *is local, then the Frame Rule holds for that command.*

Proof. The aim is to prove that the premises of the Frame Rule, $\{P\}\ \mathbb{C}\ \{Q\}$ and $\mathrm{mod}(\mathbb{C}) \cap \mathrm{free}(K) = \emptyset$, imply its conclusion, $\{K \cdot P\}\ \mathbb{C}\ \{K \cdot Q\}$. This consists of showing partial correctness and fault-avoidance for $\{K{\cdot}P\}\ \mathbb{C}\ \{K{\cdot}Q\}$, as per Defn. 6.8.

Let σ, s and t be an arbitrary interpretation function, store and tree satisfying $\sigma, \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} K \cdot P$. Then $\mathsf{t} \equiv \mathrm{ap}(\mathsf{c}, \mathsf{t}_0)$ where $\sigma, \mathsf{s}, \mathsf{t}_0 \vDash_{\mathcal{T}} P$ and $\sigma, \mathsf{s}, \mathsf{c} \vDash_{\mathcal{C}} K$. The fault-avoidance condition for the premise of the Frame Rule states that $\mathbb{C}, \mathsf{s}, \mathsf{t}_0 \not\rightsquigarrow$ fault. This leads directly to the fault-avoidance condition for the conclusion, $\mathbb{C}, \mathsf{s}, \mathsf{t} \not\rightsquigarrow$ fault, by safety monotonicity. For partial correctness, it is enough to assume that $\mathbb{C}, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathsf{s}', \mathsf{t}'$ and show that $\sigma, \mathsf{s}', \mathsf{t}' \vDash_{\mathcal{T}} K \cdot Q$. By the frame property, there exists t'_0 such that $\mathbb{C}, \mathsf{s}, \mathsf{t}_0 \rightsquigarrow \mathsf{s}', \mathsf{t}'_0$ and $\mathsf{t}' \equiv \mathrm{ap}(\mathsf{c}, \mathsf{t}'_0)$. By the partial correctness of the premise, $\sigma, \mathsf{s}', \mathsf{t}'_0 \vDash_{\mathcal{T}} Q$. Hence $\sigma, \mathsf{s}', \mathsf{t}' \vDash_{\mathcal{T}} K \cdot Q$ as required.

Using the definitions above, it is possible to show that all the commands in BTU are local, and that consequently the Frame Rule is sound for BTU.

Lemma 6.12 (Locality of BTU)**.** *All the commands in BTU are local.*

Proof. Variable assignment is trivially local as it depends only on the variable store and not the tree. Consider next an arbitrary tree update at location n. This only faults if n is not in the tree; hence applying a context to a safe state will keep it safe, giving safety monotonicity. Meanwhile, the update semantics describes the transformation of a tree $\mathsf{t}_0 \equiv \mathrm{ap}(\mathsf{c}, \mathsf{n}[\mathsf{t}'])$ to a tree $\mathsf{t}_1 \equiv \mathrm{ap}(\mathsf{c}, \mathsf{t}'')$ for varying values of t''. Successfully executing the update on a larger state $\mathrm{ap}(\mathsf{c}', \mathsf{t}_0) \equiv \mathrm{ap}(\mathsf{c}' \circ \mathsf{c}, \mathsf{n}[\mathsf{t}'])$ therefore gives a transformation to $\mathrm{ap}(\mathsf{c}' \circ \mathsf{c}, \mathsf{t}'') \equiv \mathrm{ap}(\mathsf{c}', \mathsf{t}_1)$, as required by frame property (where $\circ$ is the composition of tree contexts, discussed in Section 5.5). Note that for the divergent instance of append, there is nothing to prove. Finally, the case for $\mathbb{C}_1 \,;\, \mathbb{C}_2$ follows by induction on the structure of commands, with the atomic commands above serving as the base cases. Assuming $\mathbb{C}_1$ and $\mathbb{C}_2$ to be local, safety monotonicity and the frame property follow directly from the sequence semantics.

Corollary 6.13. *The Frame Rule is sound for BTU.*

6.4 Small Axiomatisation

The remaining step in specifying the program logic is the axiomatisation of the commands. One of the observations in [ORY01] is that it is possible to axiomatise heap

$$
\begin{array}{rll}
\{0\} & n := n' & \{0 \wedge (n = n')\} \\
\{0\} & x := x' & \{0 \wedge (x = x')\} \\
\\
\{n[\text{true}]\} & [n]_{\text{T}} := 0 & \{0\} \\
\{n[\text{true}]\} & [n]_{\text{SF}} := 0 & \{n[0]\} \\
\{n[y]\} & [n]_{\text{T}} \mathrel{*}= x & \{n[y] \mid x\} \\
\{n[y]\} & [n]_{\text{SF}} \mathrel{*}= x & \{n[y \mid x]\} \\
\{y \wedge n[\text{true}]\} & x := [n]_{\text{T}} & \{y \wedge (x = y)\} \\
\{n[y]\} & x := [n]_{\text{SF}} & \{n[y] \wedge (x = y)\} \\
\{n[y]\} & n' := \text{new } [n]_{\text{T}} & \{n[y] \mid n'[0]\} \\
\{n[y]\} & n' := \text{new } [n]_{\text{SF}} & \{n[y \mid n'[0]]\}
\end{array}
$$

where $n', x', y \notin \text{dep}(\mathbb{C})$

Figure 6.4: BTU Small Axioms

update commands using precise specifications that refer only to their memory footprints. The locality restrictions on the commands mean that these 'Small Axioms' determine the behaviour of the commands on arbitrary states. Furthermore, the expressibility of enough frames, which can be added using the Frame Rule, means that the Small Axioms can be used to derive any other valid specification.

This section gives Small Axiom specifications for the tree update commands in BTU. Update commands that act at a location n can all be specified on just the singleton tree with root node n. Hence, for example, the axiom for the dispose-tree command describes disposing a tree $n[\text{true}]$, leaving 0. Auxiliary variables are used to refer to constant values in the specifications: thus, the specification for appending a tree x describes turning a tree $n[y]$ into a tree $n[y] \mid x$, where the auxiliary variable y can later be renamed or eliminated if necessary. For consistency, the axiom for assignment is also given locally, and is consequently defined on just the empty tree, the tree footprint of assignment being empty.

Definition 6.14 (BTU Axioms)**.** The Small Axioms for the atomic update commands are given in Figure 6.4. The inference rule for sequencing is the standard

Hoare rule:

$$\frac{\{P\}\ \mathbb{C}_1\ \{Q\} \quad \{Q\}\ \mathbb{C}_2\ \{R\}}{\{P\}\ \mathbb{C}_1\ ;\mathbb{C}_2\ \{R\}}$$

The simplicity of the Small Axioms means that their soundness is trivial to verify from the formula and command semantics.

6.5 Weakest Preconditions

Having presented the Small Axioms, this section shows how to express in $\mathrm{CL}_{\mathrm{tree}}$ the weakest preconditions of the update commands, and derive the weakest precondition axioms. This provides a straightforward completeness result for straightline code, as demonstrated below. Furthermore, it can also be used in constructing a verification tool: a standard method is to use a verification generator on code annotated with pre- and postconditions to derive the weakest preconditions of the given postconditions, and then verify using a theorem prover (or an automatic decision procedure if possible) that the corresponding given preconditions imply the weakest ones, thus turning a verification problem into one of validity testing.

Definition 6.15 (Weakest Preconditions)**.** The *weakest precondition states* of a command $\mathbb{C}$ with respect to a formula P and interpretation function σ is a set of states $\mathrm{wpstates}(\mathbb{C}, P, \sigma)$ where

$$(\mathsf{s},\mathsf{t}) \in \mathrm{wpstates}(\mathbb{C}, P, \sigma) \Leftrightarrow \mathbb{C}, \mathsf{s}, \mathsf{t} \not\rightsquigarrow \mathrm{fault} \wedge \forall \mathsf{s}', \mathsf{t}'.\, \mathbb{C}, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathsf{s}', \mathsf{t}' \Rightarrow \sigma, \mathsf{s}', \mathsf{t}' \vDash_{\mathcal{T}} P$$

The *weakest precondition formula* of $\mathbb{C}$ with respect to P is any formula $\mathrm{wp}(\mathbb{C}, P)$ that agrees with $\mathrm{wpstates}(\mathbb{C}, P, \sigma)$ for all interpretation functions σ:

$$\forall \sigma, \mathsf{s}, \mathsf{t}.\ \sigma, \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} \mathrm{wp}(\mathbb{C}, P) \Leftrightarrow (\mathsf{s},\mathsf{t}) \in \mathrm{wpstates}(\mathbb{C}, P, \sigma)$$

Lemma 6.16 (Weakest Precondition axioms)**.** *The weakest precondition axiom* $\{\mathrm{wp}(\mathbb{C}, P)\}\ \mathbb{C}\ \{P\}$ *is a valid Hoare triple for all commands* $\mathbb{C}$ *and formulæ* P*, and whenever* $\{P'\}\ \mathbb{C}\ \{P\}$ *holds then* $P' \Rightarrow \mathrm{wp}(\mathbb{C}, P)$*.*

Proof. Validity follows directly from the left-to-right implication in the definition of weakest precondition states, while $P' \Rightarrow \mathrm{wp}(\mathbb{C}, P)$ follows from the reverse implication.

$$
\begin{array}{rll}
\{P[n'/n]\} & n := n' & \{P\} \\
\{P[x'/x]\} & x := x' & \{P\} \\
 & & \\
\{(0 \rhd P) \cdot n[\mathrm{true}]\} & [n]_{\mathrm{T}} := 0 & \{P\} \\
\{(n[0] \rhd P) \cdot n[\mathrm{true}]\} & [n]_{\mathrm{SF}} := 0 & \{P\} \\
\{\exists y.\, ((n[y] \mid x) \rhd P) \cdot n[y]\} & [n]_{\mathrm{T}} *= x & \{P\} \\
\{\exists y.\, (n[y \mid x] \rhd P) \cdot n[y]\} & [n]_{\mathrm{SF}} *= x & \{P\} \\
\{\exists y.\, \Diamond(y \wedge n[\mathrm{true}]) \wedge P[y/x]\} & x := [n]_{\mathrm{T}} & \{P\} \\
\{\exists y.\, \Diamond n[y] \wedge P[y/x]\} & x := [n]_{\mathrm{SF}} & \{P\} \\
\{\exists y. \forall n'.\, ((n[y] \mid n'[0]) \rhd P) \cdot n[y]\} & n' := \mathrm{new}\ [n]_{\mathrm{T}} & \{P\} \\
\{\exists y. \forall n'.\, (n[y \mid n'[0]] \rhd P) \cdot n[y]\} & n' := \mathrm{new}\ [n]_{\mathrm{SF}} & \{P\}
\end{array}
$$

where $y \notin \mathrm{dep}(\mathbb{C}) \cup \mathrm{free}(P)$

Figure 6.5: BTU Weakest Preconditions

Theorem 6.17 (BTU Weakest Preconditions)**.** *The weakest precondition axioms for the atomic update commands are given in Figure 6.5. The weakest precondition of the sequence* $\mathbb{C}_1 \,;\, \mathbb{C}_2$ *is given inductively by* $\mathrm{wp}(\mathbb{C}_1, \mathrm{wp}(\mathbb{C}_2, P))$*.*

Proof. For each axiom $\{P_{wp}\}\ \mathbb{C}\ \{P\}$ in Figure 6.5, the weakest precondition P_{wp} corresponds directly to a reverse statement of the operational semantics of $\mathbb{C}$. Taking the commands one by one, we have:

(a) Assign: P_{wp} is satisfied by any tree that satisfies P after replacing the not-yet-assigned variable (x or n) by its value-to-be (x' or n'). This is the standard Hoare axiom.

(b) Dispose: P_{wp} is satisfied by any tree that contains a subnode n and satisfies P when the subtree at or below n is removed.

(c) Append: P_{wp} is satisfied by any tree that contains a subnode n and satisfies P when the tree x is successfully appended as a sibling or subforest of n.

(d) Lookup: P_{wp} is satisfied by any tree that contains a subnode n and satisfies P after replacing the not-yet-assigned variable x by its value-to-be, the subtree at or below n.

(e) New: P_{wp} is satisfied by any tree that contains a subnode n and satisfies P when a node $n'[0]$ is successfully appended as a sibling or a subforest of n. Note that this must hold for all possible values of n', since new assigns a value non-deterministically and a specification must be true for every execution. Furthermore, the quantification over n' captures any occurrences of the not-yet-assigned variable n' in P.

The precondition for sequences is standard, and follows directly from the semantics.

Note that the weakest preconditions for dispose, append and new all make use of the $\rhd$ adjoint of context application. Indeed, the expressibility of this adjoint in Context Logic is a major component of its expressivity (distinguishing it for example from the Ambient Logic) and helps provide the necessary context frames for completeness. In fact, this should not be surprising, since $\rhd$ here plays the same role in backward-reasoning as $-\!\!*$ does in Separation Logic. Indeed, there exists a consistent way of deriving the weakest preconditions above from the specifications of the commands: $\mathrm{wp}(\mathbb{C}, P')$ can be derived from any specification $\{P\}\ \mathbb{C}\ \{Q\}$ by applying the context frame $(Q \rhd P')$, leaving the triple $\{(Q \rhd P') \cdot P\}\ \mathbb{C}\ \{P'\}$ by the modus ponens result in Thm. 3.13. Further simplification, using the Rule of Consequence and Auxiliary Variable Elimination, results in the weakest precondition axioms.

Lemma 6.18 (Derivability of Weakest Preconditions)**.** *The weakest preconditions in Figure 6.5 are derivable from the small axioms in Figure 6.4 using the proof rules in Figure 6.3. Similarly, the weakest precondition for sequencing is derivable from the sequencing inference rule.*

Proof. The derivations for the atomic commands are given in Figure 6.7 at the end of the chapter. Note that the frame contexts introduced by the Frame Rule are marked in bold, while some uses of the Rule of Consequence reference the theoretical results on which they depend. Additionally, there is an implicit use of Auxiliary Variable Renaming in the derivations to ensure that the auxiliary variables in the Small Axioms are distinct from the free variables in the postcondition P. The derivation for sequencing is the standard Hoare Logic one:

$$\frac{\{\mathrm{wp}(\mathbb{C}_1, \mathrm{wp}(\mathbb{C}_2, P))\}\ \mathbb{C}_1\ \{\mathrm{wp}(\mathbb{C}_2, P)\} \quad \{\mathrm{wp}(\mathbb{C}_2, P)\}\ \mathbb{C}_2\ \{P\}}{\{\mathrm{wp}(\mathbb{C}_1, \mathrm{wp}(\mathbb{C}_2, P))\}\ \mathbb{C}_1\ ;\ \mathbb{C}_2\ \{P\}}$$

The derivability of the weakest preconditions gives a simple completeness result for BTU (a more complicated completeness result based on the consistent shape of the derivations, as in [Yan01b], is not considered here).

Theorem 6.19 (Completeness)**.** *The program logic defined in this section is complete with respect to the interpretation of Hoare triples. In other words, every valid triple* $\{P\}\ \mathbb{C}\ \{Q\}$ *is derivable.*

Proof. Any valid triple $\{P\}\ \mathbb{C}\ \{Q\}$ is derivable by deriving $\{\mathrm{wp}(\mathbb{C}, Q)\}\ \mathbb{C}\ \{Q\}$ and applying the Rule of Consequence, since $P \Rightarrow \mathrm{wp}(\mathbb{C}, Q)$.

6.6 Examples

This section considers two simple examples of local program reasoning using Context Logic. Both examples are based on the move program of Example 6.5. The first uses simple backward-reasoning to derive a safety precondition for the move program; the second uses forward-reasoning and the Frame Rule to derive a partial specification.

Example 6.20 (Backward reasoning)**.** Calculating the weakest precondition of a program starting with the postcondition 'true' provides the necessary and sufficient safety condition for non-faulting execution. This is illustrated below for the move program in Example 6.5, applying the weakest preconditions axioms sequentially, starting at the end of the program, and simplifying where appropriate using the Rule of Consequence:

$\{\mathrm{true}\}$

$\quad [n']_{\mathrm{SF}} \mathrel{*\!\!=} x$

$\{\exists y.\, ((n'[y \mid x] \rhd \mathrm{true}) \cdot n'[y])\} \Leftrightarrow \{\Diamond n'[\mathrm{true}]\}$

$\quad [n]_{\mathrm{T}} := 0\ ;$

$\{(0 \rhd \Diamond n'[\mathrm{true}]) \cdot n[\mathrm{true}]\}$

$\quad x := [n]_{\mathrm{T}}\ ;$

$\{\exists y.\, \Diamond(y \wedge n[\mathrm{true}]) \wedge ((0 \rhd \Diamond n'[\mathrm{true}]) \cdot n[\mathrm{true}])\} \Leftrightarrow \{(0 \rhd \Diamond n'[\mathrm{true}]) \cdot n[\mathrm{true}]\}$

Thus, the formula $((0 \rhd \Diamond n'[\mathrm{true}]) \cdot n[\mathrm{true}])$ expresses the safety precondition of the move command. This states concisely just what one expects: that both the source

location n and target location n' must be in the tree, and that n' must not be a direct ancestor of n.

Example 6.21 (Forward reasoning)**.** It is not difficult to extend the safety precondition above to a partial specification of move. The derivation, given in Figure 6.6, employs auxiliary tree variables as placeholders for both the tree that is moved and the subforest at the target, and reasons forward using the Frame Rule. The resulting Hoare triple is surprisingly concise:

$$\{(0 \rhd \diamond n'[y]) \cdot (n[\text{true}] \wedge z)\}\ \text{move}(n, n')\ \{\diamond n'[y \mid z]\}$$

Note, however, that this is not a complete specification for move since it does not specify that the part of the tree outside n and n' is preserved. It is also far from being a local specification, since the precondition does not specify the minimal tree containing n and n'. The problem of providing complete, local specifications for programs such as move is discussed in Chapter 8.

6.7 Node Renaming

As mentioned previously, the append command given in Section 6.1 is necessarily partial for reasons of locality. This choice is problematic: repeatedly appending a tree is bound to diverge, for example, while in general predicting divergence in the presence of the non-deterministic new command is impossible. However, what this actually illustrates is that direct appending in an environment with unique identifiers is a somewhat artificial operation. In reality, the focus of tree reasoning is the tree structure, not the explicit identifier names, whose purpose is merely to allow location references (and possibly pointers). Hence, a more natural approach to appending would be to allow a renaming of the nodes in the added tree. This makes append a total operation, and allows repeat appending. The solution is not perfect, though: by renaming identifiers, append no longer preserves external references. This new issue is discussed further at the end of this chapter and in Chapter 8.

Definition 6.22 (Tree equivalence)**.** Two trees are *equivalent modulo renaming*, written $\mathtt{t}_1 \simeq \mathtt{t}_2$, if each can be mapped onto the other by renaming its respective nodes.

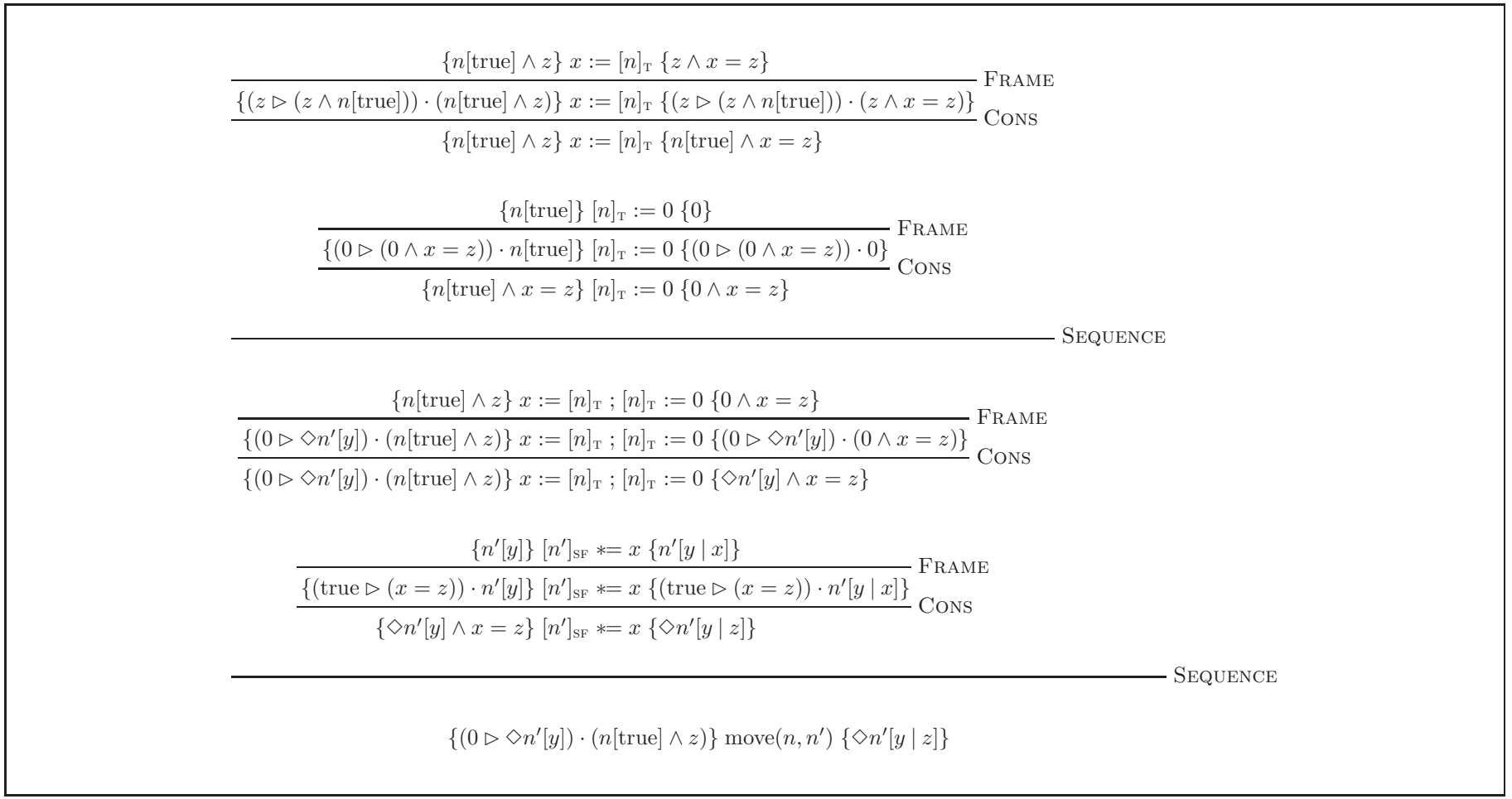


Figure 6.6: BTU Forward Reasoning Example

Definition 6.23 (Append with renaming)**.** The append-with-renaming commands have the following (non-deterministic) operational semantics.

$$\frac{\mathsf{s}(n) = \mathsf{n} \quad \mathsf{t} \equiv \mathrm{ap}(\mathsf{c}, \mathsf{n}[\mathsf{t}']) \quad \mathsf{s}(x) \simeq \mathsf{t}'' \quad \mathsf{t}'' \# \mathsf{t}}{[x]_{\mathrm{T}} \mathrel{*}\!= n, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathsf{s}, \mathrm{ap}(\mathsf{c}, \mathsf{n}[\mathsf{t}'] \mid \mathsf{t}'')}$$

$$\frac{\mathsf{s}(n) = \mathsf{n} \quad \mathsf{t} \equiv \mathrm{ap}(\mathsf{c}, \mathsf{n}[\mathsf{t}']) \quad \mathsf{s}(x) \simeq \mathsf{t}'' \quad \mathsf{t}'' \# \mathsf{t}}{[x]_{\mathrm{SF}} \mathrel{*}\!= n, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathsf{s}, \mathrm{ap}(\mathsf{c}, \mathsf{n}[\mathsf{t}' \mid \mathsf{t}''])}$$

Expressing node renaming in the logic can be achieved most cleanly by introducing a renaming modality, $[\![-]\!]$, where $[\![P]\!]$ is satisfied by any tree that is equivalent modulo renaming to a tree that satisfies P. The Small Axioms and weakest preconditions of appending can then be restated exactly as before, with the addition of renaming.

Definition 6.24 (Renaming modality)**.** The renaming modality $[\![-]\!]$ has the following semantics:

$$\sigma, \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} [\![P]\!] \Leftrightarrow \exists \mathsf{t}'.\, \mathsf{t} \simeq \mathsf{t}' \wedge \sigma, \mathsf{s}, \mathsf{t}' \vDash_{\mathcal{T}} P$$

Definition 6.25 (Small axioms)**.** The Small Axioms of the append-with-renaming commands are given below.

$$\{n[y]\} \quad [n]_{\mathrm{T}} \mathrel{*}\!= x \quad \{n[y] \mid [\![x]\!]\}$$
$$\{n[y]\} \quad [n]_{\mathrm{SF}} \mathrel{*}\!= x \quad \{n[y \mid [\![x]\!]]\}$$

Soundness follows easily from the formula semantics.

Lemma 6.26 (Weakest preconditions)**.** *The weakest preconditions of the append-with-renaming commands are:*

$$\{\exists y.\, ((n[y] \mid [\![x]\!]) \rhd P) \cdot n[y]\} \quad [n]_{\mathrm{T}} \mathrel{*}\!= x \quad \{P\}$$
$$\{\exists y.\, (n[y \mid [\![x]\!]] \rhd P) \cdot n[y]\} \quad [n]_{\mathrm{SF}} \mathrel{*}\!= x \quad \{P\}$$

Proof. This follows directly from the operational semantics: the weakest preconditions are satisfied by any tree that contains a subnode n and satisfies P when some renaming of the tree x is successfully appended as a sibling or a subforest of n.

Lemma 6.27 (Derivability of Weakest Preconditions)**.** *The weakest preconditions of the append-with-renaming commands are derivable from the Small Axioms.*

Proof. The derivations are identical to those of append-without-renaming (see Figure 6.7), only with the renaming modality around x throughout.

As noted before, the only problem with the append-with-renaming command is that it does not necessarily preserve external references. While this is usually acceptable, and is in some cases necessary (such as when repeatedly appending the same tree), there are some circumstances where one would expect references to be preserved, and where renaming is not necessary or appropriate. The most obvious such example is for a program that moves a subtree from one location to another, such as the example in Example 6.5. Here, there is no need to rename any nodes, since moving is always a total operation. One solution, considered in Chapter 8, is to introduce move as a core command in addition to append-with-renaming. The issue of specifying such a command locally is interesting, since the definition of a memory footprint is less obvious for commands that act at two locations in a tree (the target and destination), particularly since these locations may be either in disjoint parts of the tree or nested. This is discussed further in Chapter 8.

Dispose

$$\dfrac{\dfrac{\dfrac{}{\{n[\mathrm{true}]\}\ [n]_{\mathrm{T}} := 0\ \{0\}}\ \textsc{Frame}}{\{(\mathbf{0} \rhd \boldsymbol{P}) \cdot n[\mathrm{true}]\}\ [n]_{\mathrm{T}} := 0\ \{(\mathbf{0} \rhd \boldsymbol{P}) \cdot 0\}}\ \textsc{Cons}}{\{(0 \rhd P) \cdot n[\mathrm{true}]\}\ [n]_{\mathrm{T}} := 0\ \{P\}}$$

$$\dfrac{\dfrac{\dfrac{}{\{n[\mathrm{true}]\}\ [n]_{\mathrm{SF}} := 0\ \{n[0]\}}\ \textsc{Frame}}{\{(\boldsymbol{n[0]} \rhd \boldsymbol{P}) \cdot n[\mathrm{true}]\}\ [n]_{\mathrm{SF}} := 0\ \{(\boldsymbol{n[0]} \rhd \boldsymbol{P}) \cdot n[0]\}}\ \textsc{Cons}}{\{(n[0] \rhd P) \cdot n[\mathrm{true}]\}\ [n]_{\mathrm{SF}} := 0\ \{P\}}$$

Append

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{n[y]\}\ [n]_{\mathrm{T}} \mathrel{*}\!= x\ \{n[y] \mid x\}}\ \textsc{Frame}}{\{((\boldsymbol{n[y]} \mid \boldsymbol{x}) \rhd \boldsymbol{P}) \cdot n[y]\}\ [n]_{\mathrm{T}} \mathrel{*}\!= x\ \{((\boldsymbol{n[y]} \mid \boldsymbol{x}) \rhd \boldsymbol{P}) \cdot (n[y] \mid x)\}}\ \textsc{Cons}}{\{((n[y] \mid x) \rhd P) \cdot n[y]\}\ [n]_{\mathrm{T}} \mathrel{*}\!= x\ \{P\}}\ \textsc{Elim}}{\{\exists y.\, ((n[y] \mid x) \rhd P) \cdot n[y]\}\ [n]_{\mathrm{T}} \mathrel{*}\!= x\ \{P\}}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{n[y]\}\ [n]_{\mathrm{SF}} \mathrel{*}\!= x\ \{n[y \mid x]\}}\ \textsc{Frame}}{\{(\boldsymbol{n[y \mid x]} \rhd \boldsymbol{P}) \cdot n[y]\}\ [n]_{\mathrm{SF}} \mathrel{*}\!= x\ \{(\boldsymbol{n[y \mid x]} \rhd \boldsymbol{P}) \cdot n[y \mid x]\}}\ \textsc{Cons}}{\{(n[y \mid x] \rhd P) \cdot n[y]\}\ [n]_{\mathrm{SF}} \mathrel{*}\!= x\ \{P\}}\ \textsc{Elim}}{\{\exists y.\, (n[y \mid x] \rhd P) \cdot n[y]\}\ [n]_{\mathrm{SF}} \mathrel{*}\!= x\ \{P\}}$$

Figure 6.7: Derivations of the Weakest Preconditions (1/3)

Lookup

$$\dfrac{\dfrac{\dfrac{\dfrac{\{y \wedge n[\text{true}]\}\ x := [n]_{\text{T}}\ \{y \wedge n[\text{true}] \wedge (x = y)\}}{\{(\boldsymbol{(y \wedge n[\text{true}]) \rhd P[y/x])} \cdot (y \wedge n[\text{true}])\}\ x := [n]_{\text{T}}\ \{\boldsymbol{(}(y \wedge n[\text{true}]) \rhd P[y/x]\boldsymbol{)} \cdot (y \wedge n[\text{true}] \wedge (x = y))\}}\ \textsc{Frame}}{\{((y \wedge n[\text{true}]) \rhd P[y/x]) \cdot (y \wedge n[\text{true}])\}\ x := [n]_{\text{T}}\ \{((y \wedge n[\text{true}]) \rhd P[y/x]) \cdot (y \wedge n[\text{true}]) \wedge (x = y)\}}\ \textsc{Cons} : \text{C3.19}}{\{\Diamond(y \wedge n[\text{true}]) \wedge P[y/x]\}\ x := [n]_{\text{T}}\ \{P[y/x] \wedge (x = y)\}}\ \textsc{Cons} : \text{T3.23}}{\{\exists y.\, \Diamond(y \wedge n[\text{true}]) \wedge P[y/x]\}\ x := [n]_{\text{T}}\ \{P\}}\ \textsc{Cons/Elim}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\{n[y]\}\ x := [n]_{\text{SF}}\ \{n[y] \wedge (x = y)\}}{\{\boldsymbol{(n[y] \rhd P[y/x])} \cdot n[y]\}\ x := [n]_{\text{SF}}\ \{\boldsymbol{(n[y] \rhd P[y/x])} \cdot (n[y] \wedge (x = y))\}}\ \textsc{Frame}}{\{(n[y] \rhd P[y/x]) \cdot n[y]\}\ x := [n]_{\text{SF}}\ \{((n[y] \rhd P[y/x]) \cdot n[y]) \wedge (x = y)\}}\ \textsc{Cons} : \text{C3.19}}{\{\Diamond n[y] \wedge P[y/x]\}\ x := [n]_{\text{SF}}\ \{(x = y) \wedge P[y/x]\}}\ \textsc{Cons} : \text{L3.23}}{\{\exists y.\, \Diamond n[y] \wedge P[y/x]\}\ x := [n]_{\text{SF}}\ \{P\}}\ \textsc{Cons/Elim}$$

New

$$\dfrac{\dfrac{\dfrac{\{n[y]\}\ n' := \text{new}\ [n]_{\text{T}}\ \{n[y] \mid n'[0]\}}{\{\boldsymbol{(\forall n'.\, (n[y] \mid n'[0]) \rhd P)} \cdot n[y]\}\ n' := \text{new}\ [n]_{\text{T}}\ \{\boldsymbol{(\forall n'.\, (n[y] \mid n'[0]) \rhd P)} \cdot (n[y] \mid n'[0])\}}\ \textsc{Frame}}{\{\exists y. \forall n'. ((n[y] \mid n'[0]) \rhd P) \cdot n[y]\}\ n' := \text{new}\ [n]_{\text{T}}\ \{P\}}\ \textsc{Cons/Elim}$$

$$\dfrac{\dfrac{\dfrac{\{n[y]\}\ n' := \text{new}\ [n]_{\text{SF}}\ \{n[y \mid n'[0]]\}}{\{\boldsymbol{(\forall n'.\, n[y \mid n'[0]] \rhd P)} \cdot n[y]\}\ n' := \text{new}\ [n]_{\text{SF}}\ \{\boldsymbol{(\forall n'.\, n[y \mid n'[0]] \rhd P)} \cdot (n[y \mid n'[0]])\}}\ \textsc{Frame}}{\{\exists y. \forall n'. (n[y \mid n'[0]] \rhd P) \cdot n[y]\}\ n' := \text{new}\ [n]_{\text{SF}}\ \{P\}}\ \textsc{Cons/Elim}$$

Figure 6.7: Derivations of the Weakest Preconditions (2/3)

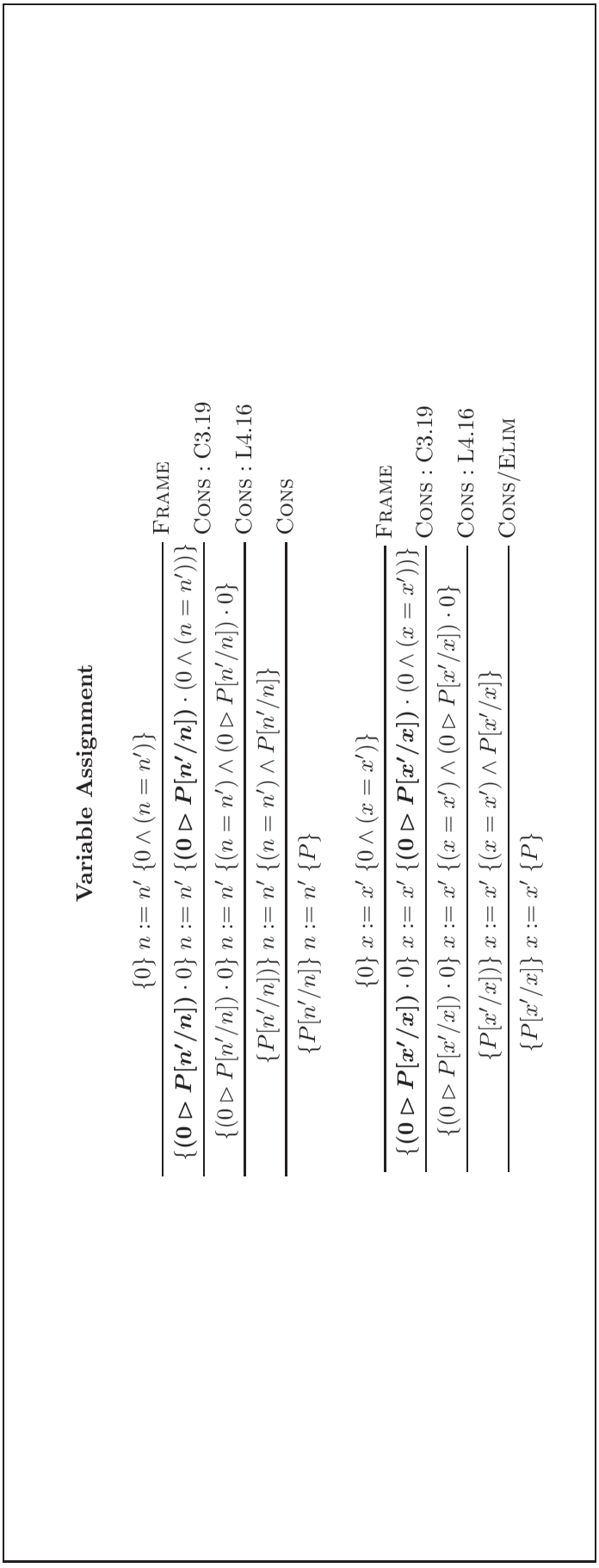


Figure 6.7: Derivations of the Weakest Preconditions (3/3)

Chapter 7

Heap Update and Term Rewriting

This short chapter illustrates the robustness of the local reasoning framework introduced in the previous chapter by applying it to other forms of data update. Two adaptations are considered: heap update and term rewriting.

7.1 Heap Update

The first adaptation of Context Logic-based local reasoning is to heap update. This adaptation is particularly straightforward, due to the collapse of Context Logic to Separation Logic for the monoidal heap structure (Thm. 4.22). Thus, the presentation given here corresponds almost exactly to the Separation Logic presentation of heap update given in [ORY01]. However, the strong similarities between heap and tree update, both in reasoning and specification, make this exposition instructive.

The heap structure, variable store and expression language used in this section were all previously defined in Chapter 5. The update language, meanwhile, is identical to that presented in [ORY01], with the exception of a slightly simplified allocation command. Thus, it comprises a simple local update language with commands for variable assignment, disposal, mutation, lookup and allocation. As noted in Chapter 6, these heap updates correspond closely to the basic tree update commands in BTU.

Definition 7.1 (Heaps, stores and expressions)**.** Heaps $\mathsf{h} \in \mathcal{H}$, heap contexts $\mathsf{c} \in \mathcal{C}$ and variable stores $\mathsf{s} \in \mathcal{S}$ are defined as in Defns. 5.12, 5.13 and 5.14. Expressions E are defined as in Defn. 5.15.

$$\frac{[\![E]\!]\mathsf{s} = \mathsf{n}}{x := E, \mathsf{s}, \mathsf{h} \rightsquigarrow [\mathsf{s}|x \leftarrow \mathsf{n}], \mathsf{h}} \qquad \frac{\mathsf{n} \notin \mathrm{locs}(\mathsf{h})}{x := \mathrm{new}(), \mathsf{s}, \mathsf{t} \rightsquigarrow [\mathsf{s}|x \leftarrow \mathsf{n}], \mathsf{h} * \mathsf{n} \mapsto \mathrm{nil}}$$

$$\frac{[\![E]\!]\mathsf{s} = \mathsf{n} \quad \mathsf{h} \equiv \mathsf{h}' * \mathsf{n} \mapsto \mathsf{v}}{\mathrm{dispose}\ E, \mathsf{s}, \mathsf{h} \rightsquigarrow \mathsf{s}, \mathsf{h}'} \qquad \frac{[\![E]\!]\mathsf{s} = \mathsf{n} \quad \mathsf{h} \equiv \mathsf{h}' * \mathsf{n} \mapsto \mathsf{v} \quad [\![F]\!]\mathsf{s} = \mathsf{v}'}{[E] := F, \mathsf{s}, \mathsf{h} \rightsquigarrow \mathsf{s}, \mathsf{h}' * \mathsf{n} \mapsto \mathsf{v}'}$$

$$\frac{[\![E]\!]\mathsf{s} = \mathsf{n} \quad \mathsf{h} \equiv \mathsf{h}' * \mathsf{n} \mapsto \mathsf{v}}{x := [E], \mathsf{s}, \mathsf{h} \rightsquigarrow [\mathsf{s}|x \leftarrow \mathsf{v}], \mathsf{h}}$$

$$\frac{\mathbb{C}_1, \mathsf{s}, \mathsf{h} \rightsquigarrow \mathbb{C}', \mathsf{s}', \mathsf{h}'}{(\mathbb{C}_1\ ;\ \mathbb{C}_2), \mathsf{s}, \mathsf{h} \rightsquigarrow (\mathbb{C}'\ ;\ \mathbb{C}_2), \mathsf{s}', \mathsf{h}'} \qquad \frac{\mathbb{C}_1, \mathsf{s}, \mathsf{h} \rightsquigarrow \mathsf{s}', \mathsf{h}'}{(\mathbb{C}_1\ ;\ \mathbb{C}_2), \mathsf{s}, \mathsf{h} \rightsquigarrow \mathbb{C}_2, \mathsf{s}', \mathsf{h}'}$$

$$\frac{\mathbb{C}_1, \mathsf{s}, \mathsf{h} \rightsquigarrow \mathrm{fault}}{(\mathbb{C}_1\ ;\ \mathbb{C}_2), \mathsf{s}, \mathsf{h} \rightsquigarrow \mathrm{fault}} \qquad \frac{[\![E]\!]\mathsf{s} = \mathsf{n} \quad \mathsf{h} \not\equiv \mathsf{h}' * \mathsf{n} \mapsto \mathsf{v}}{\mathbb{C}_{\mathrm{up}}(E), \mathsf{s}, \mathsf{t} \rightsquigarrow \mathrm{fault}}$$

Figure 7.1: HUL Operational Semantics

Definition 7.2 (HUL commands)**.** The commands of the Heap Update Language (HUL) are given by the following grammar:

$$\begin{array}{rll} \mathbb{C} ::= & x := E & \text{variable assignment} \\ & x := \mathrm{new}() & \text{allocation} \\ & \mathbb{C}_{\mathrm{up}}(E) & \text{update at location } E \\ & \mathbb{C}\ ;\ \mathbb{C} & \text{sequencing} \end{array}$$

The heap update commands $\mathbb{C}_{\mathrm{up}}(E)$ acting at a location given by an expression E are defined as follows:

$$\begin{array}{rll} \mathbb{C}_{\mathrm{up}}(E) ::= & \mathrm{dispose}\ E & \text{dispose} \\ & [E] := F & \text{mutation} \\ & x := [E] & \text{lookup} \end{array}$$

Note that the flatness of the heap means that, unlike in BTU, the allocation command does not require a target location, and is therefore grouped separately from the other update commands. Similarly, there is only one version of each of the update commands, as opposed to two, while the mutation command replaces a value,

$$\begin{array}{rll}
\{0 \wedge (x' = E)\} & x := E & \{0 \wedge (x = x')\} \\
\{0\} & x := \mathrm{new}() & \{x \mapsto \mathrm{nil}\} \\
\\
\{E \mapsto -\} & \mathrm{dispose}\ E & \{0\} \\
\{E \mapsto -\} & [E] := F & \{E \mapsto F\} \\
\{(x' \mapsto y) \wedge (x' = E)\} & x := [E] & \{(x' \mapsto y) \wedge (x = y)\}
\end{array}$$

where $x', y \notin \mathrm{mod}(\mathbb{C})$

Figure 7.2: HUL Small Axioms

rather than appending to it. Finally, there is only one type of variables, for names, since the language manipulates only values, not entire heaps.

Definition 7.3 (HUL operational semantics)**.** The operational semantics of HUL is given in Figure 7.1, using an evaluation relation $\rightsquigarrow$ defined on configuration triples $\mathbb{C}, \mathsf{s}, \mathsf{h}$, terminal states s, h and memory faults. The set of modified variables $\mathrm{mod}(\mathbb{C})$, defined as in Defn. 6.6, is approximated by $\{x\}$ for assignment, lookup and new, $\emptyset$ for dispose and mutation, and $\mathrm{mod}(\mathbb{C}_1) \cup \mathrm{mod}(\mathbb{C}_2)$ for $\mathbb{C}_1 \,;\, \mathbb{C}_2$. The set of dependent variables $\mathrm{dep}(\mathbb{C})$, defined as in Defn. 6.7, is approximated by the syntactically free variables in $\mathbb{C}$.

Observation 7.4. All the commands in HUL are local, as per Defn. 6.10.

To reason about HUL, we use the $\mathrm{CL}_{\mathrm{heap}}$ logic from Defn. 5.16. With this as the underlying assertion language, one can then apply exactly the same local Hoare-style reasoning as in Chapter 6. The interpretation of Hoare Triples (Defn. 6.8), the Frame Rule, and the other standard Hoare inference rules (Defn. 6.9) are all adapted to heaps in the obvious way. As before, the soundness of the Frame Rule follows from the locality of the commands.

As for trees, it is possible to give Small Axioms for each of the commands. Unsurprisingly, these resemble the Small Axioms of the corresponding basic tree updates. It is also possible to state and derive the weakest precondition axioms for the commands, which gives a basic completeness result for straightline code. Again, these derivations correspond closely to those in BTU.

Definition 7.5 (HUL Small Axioms)**.** The Small Axioms for the atomic HUL commands are given in Figure 7.2.

$$\begin{array}{rll} \{P[E/x]\} & x := E & \{P\} \\ \{\forall x.\,(x \mapsto \mathrm{nil}) \mathrel{-\!\!*} P\} & x := \mathrm{new}() & \{P\} \\ \\ \{P * (E \mapsto -)\} & \mathrm{dispose}\ E & \{P\} \\ \{((E \mapsto F) \mathrel{-\!\!*} P) * (E \mapsto -)\} & [E] := F & \{P\} \\ \{\exists y.\,(\diamond E \mapsto y) \wedge P[y/x]\} & x := [E] & \{P\} \end{array}$$

where $y \notin \mathrm{free}(\mathbb{C}) \cup \mathrm{free}(P)$

Figure 7.3: HUL Weakest Preconditions

As for tree update, the Small Axioms describe the behaviour of the commands only on their memory footprint; global behaviour can be inferred using the Frame Rule. For example, the specification for dispose describes removing a single cell $(E \mapsto -)$ and being left with the empty heap. This is analogous to the specification given for tree disposal in BTU (Figure 6.4), which described removing a subtree $n[\mathrm{true}]$ and being left with the empty tree.

Lemma 7.6 (HUL Weakest Preconditions)**.** *The weakest precondition axioms of the basic update commands are the ones given in Figure 7.3. Furthermore, these are derivable from the Small Axioms in Figure 7.2.*

Proof. As in BTU, the weakest preconditions correspond to reverse statements of the operational semantics of the commands. For assignment, this is the standard Hoare assignment axiom. For new, it states that the postcondition P must hold whenever a fresh location x is added to the heap. For dispose, it states that it is possible to remove location E from the heap and have P hold afterwards. For mutation, it states that the update location E is present in the heap, and that updating it with the new value F makes P hold. Finally, for lookup it states that the same P holds as after the execution, except that the value-to-be y at location E must be used for the not-yet assigned variable x.

The derivations of the weakest preconditions from the Small Axioms are shown in Figure 7.4. These all consist of applying the Frame Rule, followed by Consequence and Variable Elimination. Notice the similarity of the derivations to those of the weakest preconditions of BTU.

$$\dfrac{\dfrac{\{0 \wedge (x' = E)\}\ x := E\ \{0 \wedge (x = x')\}}{\{P[x'/x] * (0 \wedge (x' = E))\}\ x := E\ \{P[x'/x] * (0 \wedge (x = x'))\}}\ \textsc{Frame}}{\{P[E/x]\}\ x := E\ \{P\}}\ \textsc{Cons/Elim}$$

$$\dfrac{\dfrac{\{0\}\ x := \text{new}()\ \{x \mapsto \text{nil}\}}{\{(\forall x.\, (x \mapsto \text{nil}) \mathrel{-\!\!*} P) * 0\}\ x := \text{new}()\ \{(\forall x.\, (x \mapsto \text{nil}) \mathrel{-\!\!*} P) * (x \mapsto \text{nil})\}}\ \textsc{Frame}}{\{\forall x.\, (x \mapsto \text{nil}) \mathrel{-\!\!*} P\}\ x := \text{new}()\ \{P\}}\ \textsc{Cons}$$

$$\dfrac{\dfrac{\{E \mapsto -\}\ \text{dispose}\ E\ \{0\}}{\{P * E \mapsto -\}\ \text{dispose}\ E\ \{P * 0\}}\ \textsc{Frame}}{\{P * E \mapsto -\}\ \text{dispose}\ E\ \{P\}}\ \textsc{Cons}$$

$$\dfrac{\dfrac{\{E \mapsto -\}\ [E] := F\ \{E \mapsto F\}}{\{((E \mapsto F) \mathrel{-\!\!*} P) * (E \mapsto -)\}\ [E] := F\ \{((E \mapsto F) \mathrel{-\!\!*} P) * (E \mapsto F)\}}\ \textsc{Frame}}{\{((E \mapsto F) \mathrel{-\!\!*} P) * (E \mapsto -)\}\ [E] := F\ \{P\}}\ \textsc{Cons}$$

$$\dfrac{\dfrac{\dfrac{\{(x' \mapsto y) \wedge (x' = E)\}\ x := [E]\ \{(x' \mapsto y) \wedge (x = y)\}}{\{((x' \mapsto y) \mathrel{-\!\!*} P[y/x]) * (x' \mapsto y) \wedge (x' = E)\}\ x := [E]\ \{((x' \mapsto y) \mathrel{-\!\!*} P[y/x]) * (x' \mapsto y) \wedge (x = y)\}}\ \textsc{Frame}}{\{(\diamond x' \mapsto y) \wedge P[y/x] \wedge (x' = E)\}\ x := [E]\ \{P[y/x] \wedge (x = y)\}}\ \textsc{Cons}}{\{\exists y.\, (\diamond E \mapsto y) \wedge P[y/x]\}\ x := [E]\ \{P\}}\ \textsc{Cons/Elim}$$

Figure 7.4: Derivations of the HUL Weakest Preconditions

7.2 Term Rewriting

The second adaptation of Context Logic-based local reasoning is to term rewriting. In many ways, this is a more interesting adaptation than the one for heaps, since an update language consisting of atomic rewrite commands is markedly different from the tree and heap update languages considered so far. Furthermore, terms themselves, like trees but unlike heaps, do not have a natural model in Separation Logic.

Applying local reasoning to term rewriting is non-trivial, since term rewriting is not generally local: a rewrite can typically be applied to multiple redexes, dependent on the context. However, once a specific redex is identified, the behaviour of that rewrite *is* local, as only the corresponding subterm is affected. One natural way of formalising this locality is to consider terms with unique location identifiers at the function symbols and, by analogy to tree update, restrict our reasoning to rewrites that act at given locations.

In order to specify term rewrites, it is also necessary to introduce variables. Two types of variable are considered. The first, name variables, are used by the rewrite commands to refer to existing location identifiers and store freshly-generated ones; these variables are part of the execution state and are kept in a variable store. The second, term variables, are used to specify subterm binding in the rewrites; these are reasoned about in the logic but are not part of the state, and are thus kept in a separate environment. Finally, we introduce an expression language consisting of terms combined with name and term variables to explicitly specify rewrites.

Definition 7.7 (Located preterms)**.** Given a set $\mathcal{N} = \{\mathsf{n}, \dots\}$ of location names, a set $\mathcal{F} = \{\mathsf{f}, \dots\}$ of function symbols, and a signature $\Sigma : \mathcal{F} \to \mathbb{N}$ mapping function symbols to arities, located preterms $\mathsf{t} \in \mathsf{T}_{\text{locpre}}$ and located preterm contexts $\mathsf{c} \in \mathcal{C}_{\text{locpre}}$ are defined by the following grammars:

$$\begin{aligned} \mathsf{t} &::= \mathsf{f}_{\mathsf{n}}(\mathsf{t}_1, \dots \mathsf{t}_k) && k = \Sigma(\mathsf{f}) \\ \mathsf{c} &::= - \mid \mathsf{f}_{\mathsf{n}}(\mathsf{t}_1, \dots \mathsf{t}_{i-1}, \mathsf{c}, \mathsf{t}_{i+1}, \dots \mathsf{t}_k) && k = \Sigma(\mathsf{f}) \geq i \geq 1 \end{aligned}$$

The set of locations in a located preterm or preterm context is given by:

$$\begin{aligned} \operatorname{locs}(\mathsf{f}_{\mathsf{n}}(\mathsf{t}_1, \dots \mathsf{t}_k) &= \{\mathsf{n}\} \cup \bigcup_{j=1}^{k} \operatorname{locs}(\mathsf{t}_j) \\ \operatorname{locs}(-) &= \emptyset \\ \operatorname{locs}(\mathsf{f}_{\mathsf{n}}(\mathsf{t}_1, \dots \mathsf{t}_{i-1}, \mathsf{c}, \mathsf{t}_{i+1}, \dots \mathsf{t}_k)) &= \{\mathsf{n}\} \cup \operatorname{locs}(\mathsf{c}) \cup \bigcup_{j=1}^{k}{}_{j \neq i} \operatorname{locs}(\mathsf{t}_j) \end{aligned}$$

The insertion of a located preterm into a located preterm context is given by:

$$\mathrm{ap}(-, \mathsf{t}) = \mathsf{t}$$
$$\mathrm{ap}(\mathsf{f_n}(\mathsf{t}_1, \ldots \mathsf{c}, \ldots \mathsf{t}_k), \mathsf{t}) = \mathsf{f_n}(\mathsf{t}_1, \ldots \mathrm{ap}(\mathsf{c}, \mathsf{t}), \ldots \mathsf{t}_k)$$

Definition 7.8 (Located terms)**.** A *located term* $\mathsf{t} \in \mathsf{T}_{\mathrm{loc}}$ is a located preterm with unique locations: that is, where $\mathsf{t} = \mathsf{f_n}(\mathsf{t}_1, \ldots \mathsf{t}_k)$ implies $\{\mathsf{n}\}$ and $\mathrm{locs}(\mathsf{t}_i)$ are all disjoint. A *located term context* $\mathsf{c} \in \mathcal{C}_{\mathrm{loc}}$ is a located preterm context with unique locations: that is, where $\mathsf{c} = \mathsf{f_n}(\mathsf{t}_1, \ldots \mathsf{c}, \ldots \mathsf{t}_k)$ implies $\{\mathsf{n}\}$, $\mathrm{locs}(\mathsf{c})$ and $\mathrm{locs}(\mathsf{t}_i)$ are all disjoint. The insertion of a located term t into a located term context c is given by the appropriate restriction of ap, which is well-defined.

Definition 7.9 (Store and environment)**.** Given an infinite set $\mathrm{Var}_{\mathcal{N}} = \{n, \ldots\}$ of name variables, a *store* $\mathsf{s} \in \mathcal{S}$ is a total function $\mathsf{s} : \mathrm{Var}_{\mathcal{N}} \to \mathcal{N}$ returning the value of each name variable. Given an infinite set $\mathrm{Var}_{\mathsf{T}} = \{x, \ldots\}$ of term variables, an *environment* $\mathsf{e} \in \mathcal{E}$ is a total function $\mathsf{e} : \mathrm{Var}_{\mathsf{T}} \to \mathsf{T}_{\mathrm{loc}}$ returning the value of each term variable.

Definition 7.10 (Expressions)**.** An *expression* E consists of a term with variables, defined by the following grammar:

$$E, F ::= x \mid \mathsf{f}_n(E_1, \ldots E_k) \quad k = \Sigma(\mathsf{f})$$

The free name and term variables in an expression E are expressed by $\mathrm{free}_n(E)$ and $\mathrm{free}_t(E)$ respectively. The valuation of an expression in an environment and store, written $[\![E]\!]\mathsf{es}$, is either a term or an error, and is given by:

$$[\![x]\!]\mathsf{es} = \mathsf{e}(x)$$
$$[\![\mathsf{f}_n(E_1, \ldots E_k)]\!]\mathsf{es} = \begin{cases} \mathsf{f}_{\mathsf{s}(n)}([\![E_1]\!]\mathsf{es}, \ldots [\![E_k]\!]\mathsf{es}) & \text{if a well-defined located term} \\ \text{error} & \text{otherwise} \end{cases}$$

Using these definitions, it is straightforward to define a local term update language based on sequences of atomic rewrite commands. Reductions simply consist of replacing a subterm matching one expression by a subterm matching another, where name variables are used to specify the location of the redex, and term variables to bind the subterms. Any name variables used in the target expression but not in the source are assigned fresh locations. For all this to work, however, it is necessary to place some natural constraints on the expressions used in the rewrite rules. These

$$\frac{\mathsf{t} = \mathrm{ap}(\mathsf{c}, [\![E]\!]\mathsf{es}) \quad \mathsf{s}' = [\mathsf{s}|(\mathrm{free}_n(F) \setminus \mathrm{free}_n(E)) \leftarrow \vec{\mathsf{n}}] \quad \mathsf{t}' = [\![F]\!]\mathsf{es}' \quad \mathrm{ap}(\mathsf{c}, \mathsf{t}')\downarrow}{E \to F, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathsf{s}', \mathrm{ap}(\mathsf{c}, \mathsf{t}')}$$

$$\frac{\mathbb{C}_1, \mathsf{s}, \mathsf{h} \rightsquigarrow \mathbb{C}', \mathsf{s}', \mathsf{h}'}{(\mathbb{C}_1 \,;\, \mathbb{C}_2), \mathsf{s}, \mathsf{h} \rightsquigarrow (\mathbb{C}' \,;\, \mathbb{C}_2), \mathsf{s}', \mathsf{h}'} \qquad \frac{\mathbb{C}_1, \mathsf{s}, \mathsf{h} \rightsquigarrow \mathsf{s}', \mathsf{h}'}{(\mathbb{C}_1 \,;\, \mathbb{C}_2), \mathsf{s}, \mathsf{h} \rightsquigarrow \mathbb{C}_2, \mathsf{s}', \mathsf{h}'}$$

$$\frac{\mathbb{C}_1, \mathsf{s}, \mathsf{h} \rightsquigarrow \mathrm{fault}}{(\mathbb{C}_1 \,;\, \mathbb{C}_2), \mathsf{s}, \mathsf{h} \rightsquigarrow \mathrm{fault}} \qquad \frac{\mathsf{t} \not\equiv \mathrm{ap}(\mathsf{c}, [\![E]\!]\mathsf{es})}{E \to F, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathrm{fault}}$$

Figure 7.5: TRL Operational Semantics

consist of: a well-formedness condition on the variable bindings; a locatedness condition on the source expression, to ensure locality; and a linearity constraint on the term variables, due to the uniqueness of location names.

Definition 7.11 (TRL commands)**.** The commands of the Term-Rewriting Language (TRL) are given by the following grammar:

$$\mathbb{C} ::= E \to F \mid \mathbb{C} \,;\, \mathbb{C}$$

where the atomic rewrite commands $E \to F$ satisfy the following restrictions:

(a) well-formedness: $\mathrm{free}_t(F) \subseteq \mathrm{free}_t(E)$;

(b) locatedness: E is not simply a term variable x;

(c) linearity: term variables in E and F occur at most once in each.

Definition 7.12 (TRL operational semantics)**.** The operational semantics of TRL is given in Figure 7.5, using an evaluation relation $\rightsquigarrow$ defined on configuration triples $\mathbb{C}, \mathsf{s}, \mathsf{t}$, terminal states s, t and memory faults.

Note that the premise of the rewrite semantics in Figure 7.5 refers to an arbitrary environment e (which is used for variable binding but is not part of the state) and an arbitrary set of fresh node values $\vec{\mathsf{n}}$. The set of modified variables $\mathrm{mod}(E \to F)$ of a rewrite is taken to be $\mathrm{free}_n(F) \setminus \mathrm{free}_n(E)$, while the set of dependent variables $\mathrm{dep}(E \to F)$ is taken to be $\mathrm{free}_n(E) \cup \mathrm{free}_n(F)$. Since term variables are only used internally for pattern matching, they are never modified or dependent.

Example 7.13 (Term Rewrite)**.** The execution of the rewrite command

$$\mathsf{f}_n(x, y) \to \mathsf{g}_n(x, \mathsf{h}_m(y))$$

on the working term $\mathsf{h}_{\mathsf{n}_1}(\mathsf{f}_{\mathsf{n}_2}(\mathsf{c}_{\mathsf{n}_3}, \mathsf{c}_{\mathsf{n}_4}))$ in a store s where $[\![n]\!]\mathsf{s} = \mathsf{n}_2$ results in the term $\mathsf{h}_{\mathsf{n}_1}(\mathsf{g}_{\mathsf{n}_2}(\mathsf{c}_{\mathsf{n}_3}, \mathsf{h}_{\mathsf{n}_5}(\mathsf{c}_{\mathsf{n}_4})))$ where n_5 is a fresh node value assigned to m in the store. Notice that n_5 must be fresh for the resulting term to be well-formed.

Observation 7.14. The commands in TRL are local, as per Defn. 6.10.

Like for normal terms, it is easy to see that $\mathit{LocTerm} = (\mathsf{T}_{\mathrm{loc}}, \mathcal{C}_{\mathrm{loc}}, \mathrm{ap}, \{-\})$ forms a model of basic Context Logic with no zero. As before, we use the indexing construction of Example 3.8 to include both the store variables and environment variables in the model. The added formulæ, meanwhile, include expressions, term contexts (this time with location variables) and quantification over both name and term variables.

Definition 7.15 ($\mathrm{CL}_{\mathrm{locterm}}$ formulæ)**.** The formulæ of Context Logic for Located Terms ($\mathrm{CL}_{\mathrm{locterm}}$) consist of the same context and data assertions as CL with the addition of formulæ representing term contexts, expressions and quantification:

$$\begin{array}{lll}
P ::= \boldsymbol{E} & & \textbf{special formulæ} \\
\quad \mid K \cdot P \mid K \lhd P & & \text{structural formulæ} \\
\quad \mid P \Rightarrow P \mid \mathrm{false} \mid \exists \boldsymbol{n}.\boldsymbol{P} \mid \exists \boldsymbol{x}.\boldsymbol{P} & & \textbf{logical formulæ} \\
 & & \\
K ::= \mathbf{f_n}(\boldsymbol{P_1}, \ldots, \boldsymbol{P_{i-1}}, \boldsymbol{K}, \boldsymbol{P_{i+1}}, \ldots, \boldsymbol{P_k}) & & \textbf{special formulæ} \\
\quad \mid - \mid P \rhd P & & \text{structural formulæ} \\
\quad \mid K \Rightarrow K \mid \mathrm{False} \mid \exists \boldsymbol{n}.\boldsymbol{K} \mid \exists \boldsymbol{x}.\boldsymbol{K} & & \textbf{logical formulæ}
\end{array}$$

where $k = \Sigma(\mathsf{f}) \geq i \geq 1$.

Definition 7.16 ($\mathrm{CL}_{\mathrm{locterm}}$ forcing semantics)**.** The semantics of $\mathrm{CL}_{\mathrm{locterm}}$ is an extension of the semantics of CL for the model $\mathit{LocTerm}^{\mathcal{E}\times\mathcal{S}}$. Given an interpretation function $\sigma : (\mathcal{V}_{\mathcal{D}} \to \mathcal{P}(\mathcal{E} \times \mathcal{S} \times \mathsf{T}_{\mathrm{loc}})) \times (\mathcal{V}_{\mathcal{C}} \to \mathcal{P}(\mathcal{E} \times \mathcal{S} \times \mathcal{C}))$ mapping propositional variables to sets of located terms and contexts indexed by environments and stores, the forcing semantics is given by two satisfaction relations $\sigma, \mathsf{e}, \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} P$ and

$\sigma, \mathsf{e}, \mathsf{s}, \mathsf{c} \vDash_{\mathcal{C}} K$ for terms and contexts, defined as in Defn. 3.4 and extended as follows:

$$\sigma, \mathsf{e}, \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} E \quad \Leftrightarrow \mathsf{t} \equiv [\![E]\!]\mathsf{es}$$

$$\sigma, \mathsf{e}, \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} \exists n.P \Leftrightarrow \exists \mathsf{n} \in \mathcal{N}.\ \sigma, \mathsf{e}, [\mathsf{s}|n \leftarrow \mathsf{n}], \mathsf{t} \vDash_{\mathcal{T}} P$$

$$\sigma, \mathsf{e}, \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} \exists x.P \Leftrightarrow \exists \mathsf{t}' \in \mathsf{T}.\ \sigma, [\mathsf{e}|x \leftarrow \mathsf{t}'], \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} P$$

$$\sigma, \mathsf{e}, \mathsf{s}, \mathsf{c} \vDash_{\mathcal{C}} \mathsf{f}_n(P_1, \ldots, P_{i-1}, K, P_{i+1}, \ldots, P_k)$$
$$\Leftrightarrow \exists\, \mathsf{t_j} \in \mathsf{T}_{\mathrm{loc}}, \mathsf{c} \in \mathcal{C}.\ (\mathsf{t} \equiv \mathsf{f}_{[\![n]\!]\mathsf{s}}(\mathsf{t_1}, \ldots, \mathsf{c}, \ldots, \mathsf{t_k}) \wedge \sigma, \mathsf{e}, \mathsf{s}, \mathsf{c} \vDash_{\mathcal{C}} K \wedge \bigwedge_{\substack{1 \le j \le k \\ j \neq i}} \sigma, \mathsf{s}, \mathsf{t_j} \vDash_{\mathcal{T}} P_j)$$

$$\sigma, \mathsf{e}, \mathsf{s}, \mathsf{c} \vDash_{\mathcal{C}} \exists n.K \Leftrightarrow \exists \mathsf{n} \in \mathcal{N}.\ \sigma, \mathsf{e}, [\mathsf{s}|n \leftarrow \mathsf{n}], \mathsf{c} \vDash_{\mathcal{C}} K$$

$$\sigma, \mathsf{e}, \mathsf{s}, \mathsf{c} \vDash_{\mathcal{C}} \exists x.K \Leftrightarrow \exists \mathsf{t} \in \mathsf{T}.\ \sigma, [\mathsf{e}|x \leftarrow \mathsf{t}], \mathsf{s}, \mathsf{c} \vDash_{\mathcal{C}} K$$

Using this logic as the underlying assertion language, it is possible to apply exactly the same local Hoare reasoning framework as in Chapter 6 and the first section of this chapter. The interpretation of Hoare Triples (Defn. 6.8), the Frame Rule, and the other standard Hoare inference rules (Defn. 6.9) are all adapted in the obvious way, with the auxiliary variable rules applying to both name and term variables. The only remaining task is to give a local specification of the rewrite command $E \to F$. This turns out to be trivial: the appropriate Small Axiom simply has the rewrite premise E as its precondition and the target F as its postcondition. As before, it is possible to derive the weakest precondition axiom from this specification using the Frame Rule.

Definition 7.17 (TRL Small Axioms)**.** The Small Axiom for the term rewrite command is given by:

$$\{E\}\ E \to F\ \{F\}$$

Lemma 7.18 (Weakest Precondition)**.** *The weakest precondition axiom of the term rewrite command is given by:*

$$\{\exists \vec{x}_0.\, (\forall \vec{n}.(F[\vec{x}_0/\vec{x}] \rhd P)) \cdot E[\vec{x}_0/\vec{x}]\}\ E \to F\ \{P\}$$

where $\vec{x} = \mathrm{free}_t(E)$*,* $\vec{n} = \mathrm{free}_n(F) \setminus \mathrm{free}_n(E)$ *and* $\vec{x}_0$ *are fresh term variables not free in* P*. Furthermore, this is derivable from the Small Axiom above.*

Proof. The weakest precondition simply states that a redex satisfying E exists somewhere (for some bindings $\vec{x}_0$) and that whenever this redex is replaced by F (using the same bindings) then the result satisfies P. Furthermore, this must hold for all possible assignments to the free location variables $\vec{n}$ which are in F but not in E.

The derivation of the weakest precondition follows by a simple application of Variable Renaming, the Frame Rule, Consequence and Variable Elimination:

$$\dfrac{\dfrac{\dfrac{\{E\}\ E \to F\ \{F\}}{\{E[\vec{x}_0/\vec{x}]\}\ E \to F\ \{F[\vec{x}_0/\vec{x}]\}}\ \textsc{Rename}}{\{(\forall \vec{n}.(F[\vec{x}_0/\vec{x}] \rhd P))\cdot E[\vec{x}_0/\vec{x}]\}\ E \to F\ \{(\forall \vec{n}.(F[\vec{x}_0/\vec{x}] \rhd P))\cdot F[\vec{x}_0/\vec{x}]\}}\ \textsc{Frame}}{\{\exists \vec{x}_0.\,(\forall \vec{n}.(F[\vec{x}_0/\vec{x}] \rhd P))\cdot E[\vec{x}_0/\vec{x}]\}\ E \to F\ \{P\}}\ \textsc{Cons/Elim}$$

Finally, we consider two simple reasoning examples, as in Chapter 6.

Example 7.19 (Weakest Precondition)**.** For the term rewrite command $\mathsf{f}_n(x,y) \to \mathsf{g}_n(x,\mathsf{h}_m(y))$ considered in Example 7.13, the weakest precondition is:

$$\exists x_0, y_0.(\forall m.(\mathsf{g}_n(x_0,\mathsf{h}_m(y_0)) \rhd P)) \cdot \mathsf{f}_n(x_0,y_0)$$

where $x_0, y_0 \notin \mathrm{free}(P)$. In the case when P is true, this gives the safety precondition of the command, namely $\Diamond \mathsf{f}_n(\mathrm{true},\mathrm{true})$, which asserts the presence of the function symbol f at location n.

Example 7.20 (Global Specification)**.** It is easy to use the safety precondition derived in Example 7.19 to obtain a global specification of the rewrite command $\mathbb{C} = \mathsf{f}_n(x,y) \to \mathsf{g}_n(x,\mathsf{h}_m(y))$. Employing an auxiliary term variable z to represent the arbitrary term in the precondition, we can derive:

$$\dfrac{\dfrac{\dfrac{\{\mathsf{f}_n(x,y)\}\ \mathbb{C}\ \{\mathsf{g}_n(x,\mathsf{h}_m(y))\}}{\{(\mathsf{f}_n(x,y) \blacktriangleright z)\cdot \mathsf{f}_n(x,y)\}\ \mathbb{C}\ \{(\mathsf{f}_n(x,y) \blacktriangleright z)\cdot \mathsf{g}_n(x,\mathsf{h}_m(y))\}}\ \textsc{Frame}}{\{z \wedge \Diamond \mathsf{f}_n(x,y)\}\ \mathbb{C}\ \{(\mathsf{f}_n(x,y) \blacktriangleright z)\cdot \mathsf{g}_n(x,\mathsf{h}_m(y))\}}\ \textsc{Cons}\,(\mathit{Thm.}\ 3.13)}{\{z \wedge \Diamond \mathsf{f}_n(\mathrm{true},\mathrm{true})\}\ \mathbb{C}\ \{\exists x,y.(\mathsf{f}_n(x,y) \blacktriangleright z)\cdot \mathsf{g}_n(x,\mathsf{h}_m(y))\}}\ \textsc{Cons/Elim}$$

The precondition describes an arbitrary term z satisfying the safety condition, while the postcondition states that after execution, it is possible to 'undo' the rewrite and recover z.

Chapter 8

Extended Tree Update

This chapter describes local reasoning about a more realistic tree update language than the one considered in Chapter 6. The extended tree structure includes pointers and labels, while the update language includes queries and commands that act at multiple locations. The work shows the feasibility of reasoning about real update to semistructured data, and raises an interesting point concerning local specifications.

8.1 Motivation and Outline

This final chapter describes a first step towards specifying and reasoning about a more realistic tree update language, applicable to XML and other semistructured data. The work, which pushes the boundaries of Context Logic reasoning, is of a more exploratory nature than the previous chapters, and raises a number of interesting directions for future research.

The key technical jump exhibited here over previous work involves the presence of update commands that act at multiple locations. This allows us to incorporate into the update language both node queries, which are a staple part of XML manipulation, and a move-tree command, whose importance for maintaining external references was previously discussed in Section 6.7. Additionally, we introduce a more complicated tree structure, containing labels and pointers, which corresponds better to XML.

We continue from here, applying the principles of local Context Logic reasoning, and encountering a number of interesting observations along the way. For example, in modelling queries, we discover that, in light of our local reasoning approach, we must also introduce the notion of query locality. This concept, which is not present in the

standard literature, turns out to be a natural constraint (especially in a distributed or concurrent setting). Similarly, specifying the tree update language involves a choice on how best to handle simple updates at multiple locations. The declarative approach used here considers single actions at an unordered set of locations; other approaches are possible, such as the one, preferred by the DOM interface, of traversing sequences of locations and manipulating them one at a time. The extension of Context Logic to the new tree data structure is straightforward, as one would expect from Chapter 5. However, we find that, in order to reason about the sort of actions considered above, we must employ inductive predicates that recurse over the set of update locations. This, more than anything, adds a level of complexity to the reasoning not previously seen for Context Logic. Finally, extending the Hoare Reasoning framework is a simple adaptation. Nevertheless, it raises an important point concerning the rôle of local specifications in local reasoning.

On the whole, this chapter has a slightly less rigid flavour than the previous ones. Most importantly, it demonstrates that Context Logic can be used to reason about a complicated tree update language. Additionally, however, it opens up questions regarding the various choices made and the complexity of the various solutions presented, and suggests many directions for future work.

8.2 Extended Tree Model

The first stage in extending the reasoning framework involves expanding the tree data model to include both labels and sets of pointers at every node. These choices are both practically-oriented: labels allow the traversal of trees using path expressions, while pointers allow the modelling of arbitrary graph structures using references between different parts of the tree. Furthermore, the resulting data structure can be easily be described in XML using ID and IDREFS type arguments. Unlike XML, the model considered will remain unordered, both for reasons of simplicity and in order to conform with the 'trees-as-database' approach [ABS99], where the purpose of the tree structure is to describe data hierarchy, as opposed to the 'trees-as-document' approach, where sibling ordering matters. Adapting Context Logic to an ordered environment is simple, as shown in Chapter 5.

Definition 8.1 (Trees and contexts)**.** Trees $\mathsf{t} \in \mathcal{T}$ and contexts $\mathsf{c} \in \mathcal{C}$ are defined as in Defn. 5.22, but with an associated label and pointer set at every node. Hence, given an infinite set $\mathcal{N} = \{\mathsf{m}, \mathsf{n}, \dots\}$ of location names, an infinite set $\mathcal{A} = \{\mathsf{a}, \mathsf{b}, \dots\}$ of labels with some default label $\star$, and defining pointer sets $\mathsf{p} \in \mathcal{P}_{\text{fin}}(\mathcal{N})$ to be finite sets of locations, the pretrees and precontexts are given by the grammar:

$$\begin{aligned} \mathsf{t} &::= 0 \mid \mathsf{n}_{\mathsf{a}:\mathsf{p}}[\mathsf{t}'] \mid (\mathsf{t}_1 \mid \mathsf{t}_2) \\ \mathsf{c} &::= - \mid \mathsf{n}_{\mathsf{a}:\mathsf{p}}[\mathsf{c}'] \mid (\mathsf{c}' \mid \mathsf{t}) \mid (\mathsf{t} \mid \mathsf{c}') \end{aligned}$$

where $\mathsf{n}_{\mathsf{a}:\mathsf{p}}$ corresponds to a node at location n, with label a and pointer set p. The set of locations in a tree or context is defined as before:

$$\begin{array}{llll} \text{locs}(0) = \emptyset & \text{locs}(\mathsf{n}_{\mathsf{a}:\mathsf{p}}[\mathsf{t}']) = \{\mathsf{n}\} \cup \text{locs}(\mathsf{t}') & \text{locs}(\mathsf{t}_1 \mid \mathsf{t}_2) = & \text{locs}(\mathsf{t}_1) \cup \text{locs}(\mathsf{t}_2) \\ \text{locs}(-) = \emptyset & \text{locs}(\mathsf{n}_{\mathsf{a}:\mathsf{p}}[\mathsf{c}']) = \{\mathsf{n}\} \cup \text{locs}(\mathsf{c}') & \left.\begin{array}{l} \text{locs}(\mathsf{c}' \mid \mathsf{t}) \\ \text{locs}(\mathsf{t} \mid \mathsf{c}') \end{array}\right\} & = \text{locs}(\mathsf{t}) \cup \text{locs}(\mathsf{c}') \end{array}$$

as is the structural congruence between trees and contexts:

$$\begin{array}{llll} 0 \mid \mathsf{t} \equiv \mathsf{t} \mid 0 \equiv 0 & \mathsf{t}_1 \mid (\mathsf{t}_2 \mid \mathsf{t}_3) \equiv (\mathsf{t}_1 \mid \mathsf{t}_2) \mid \mathsf{t}_3 & \mathsf{t}_1 \mid \mathsf{t}_2 \equiv \mathsf{t}_2 \mid \mathsf{t}_1 \\ - \mid \mathsf{t} \equiv \mathsf{t} \mid - \equiv - & \mathsf{c} \mid (\mathsf{t}_1 \mid \mathsf{t}_2) \equiv (\mathsf{c} \mid \mathsf{t}_1) \mid \mathsf{t}_2 & \mathsf{c} \mid \mathsf{t} \equiv \mathsf{t} \mid \mathsf{c} \end{array}$$

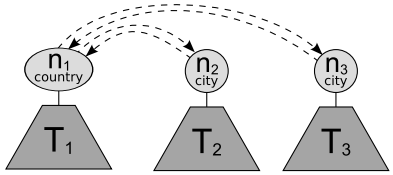
Trees and contexts are defined as pretrees and precontexts with unique locations.

Definition 8.2 (Tree equivalence)**.** Two trees are *equivalent modulo renaming*, written $\mathsf{t}_1 \simeq \mathsf{t}_2$, if each can be mapped onto the other by renaming their respective nodes as well as any internal pointers to those nodes. Hence, for example $\mathsf{n}_{\mathsf{a}:\{\mathsf{n},\mathsf{m}\}}[0] \simeq \mathsf{n}'_{\mathsf{a}:\{\mathsf{n}',\mathsf{m}\}}[0]$.

Definition 8.3 (XML translation)**.** There is a natural translation from the tree structure above to a simple fragment of XML, translating labels to tag names, nodes to ID attributes and pointers to IDREFS attributes. Since the tree structure, unlike XML, is unordered, the translation picks an arbitrary ordering for parallel composition.

$$\begin{aligned} [\![0]\!] &= \\ [\![\mathsf{n}_{\mathsf{a}:\mathsf{p}}[\mathsf{t}]]\!] &= \langle \texttt{a ID="n" IDREFS="p"} \rangle [\![\mathsf{t}]\!] \langle / \texttt{a} \rangle \\ [\![\mathsf{t}_1 \mid \mathsf{t}_2]\!] &= [\![\mathsf{t}_1]\!][\![\mathsf{t}_2]\!] \end{aligned}$$

Example 8.4 (Tree)**.** Below is a simple example of a tree with pointers, together with a possible XML representation:



$$\mathsf{n}_{1\mathsf{country}:\{\mathsf{n}_2,\mathsf{n}_3\}}[\mathsf{T}_1] \mid \mathsf{n}_{2\mathsf{city}:\{\mathsf{n}_1\}}[\mathsf{T}_2] \mid \mathsf{n}_{3\mathsf{city}:\{\mathsf{n}_1\}}[\mathsf{T}_3]$$

$$\langle \texttt{country ID="n1" IDREFS="n2 n3"} \rangle\ \mathsf{T}_1\ \langle \texttt{/country} \rangle$$

$$\langle \texttt{city ID="n2" IDREFS="n1"} \rangle\ \mathsf{T}_2\ \langle \texttt{/city} \rangle$$

$$\langle \texttt{city ID="n3" IDREFS="n1"} \rangle\ \mathsf{T}_3\ \langle \texttt{/city} \rangle$$

Extending the tree model makes it necessary to extend the storage model as well: to allow the manipulation of labels and pointer sets, the store must be expanded to include label and pointer set variables in addition to the ones for trees. The presence of pointer set variables, however, means that it is no longer necessary to include variables for single node identifiers, particularly since the update language considered deals only with updates at sets of locations.

Definition 8.5 (Stores)**.** Given infinite sets $\mathrm{Var}_{\mathcal{T}} = \{x, \dots\}$ of tree variables, $\mathrm{Var}_{\mathcal{A}} = \{a, \dots\}$ of label variables and $\mathrm{Var}_{\mathcal{P}} = \{p, \dots\}$ of pointer variables, a store $\mathsf{s} \in \mathcal{S}$ is a triple of total functions $\mathsf{s} : \mathrm{Var}_{\mathcal{T}} \to \mathcal{T} \times \mathrm{Var}_{\mathcal{A}} \to \mathcal{A} \times \mathrm{Var}_{\mathcal{P}} \to \mathcal{P}_{\mathrm{fin}}(\mathcal{N})$, returning the values of the variables.

Additionally, it is useful to introduce expressions to represent the permitted right-hand values of tree, label and pointer set assignment. As before, these typically consist of variables, while the use of explicit node identifiers in expressions is still forbidden to avoid dealing with ill-formed trees. However, other literals, such as labels, the empty tree and the empty set of pointers are permitted, and expressions are handled abstractly in the logic, with a view to extending them at some point to include more complicated structures.

Definition 8.6 (Value Expressions)**.** Tree expressions $E_{\mathcal{T}}$, label expressions $E_{\mathcal{A}}$ and pointer expressions $E_{\mathcal{P}}$ consist of either variables or simple literals:

$$E_{\mathcal{T}} ::= x \mid 0 \qquad E_{\mathcal{A}} ::= a \mid \mathsf{a} \in \mathcal{A} \qquad E_{\mathcal{P}} ::= p \mid \emptyset$$

The valuation of a value expression on a store s is written $[\![E]\!]\mathsf{s}$ and given by:

$$[\![x]\!]\mathsf{s} = \mathsf{s}(x) \qquad [\![a]\!]\mathsf{s} = \mathsf{s}(a) \qquad [\![p]\!]\mathsf{s} = \mathsf{s}(p)$$
$$[\![0]\!]\mathsf{s} = 0 \qquad [\![\mathsf{a}]\!]\mathsf{s} = \mathsf{a} \qquad [\![\emptyset]\!]\mathsf{s} = \emptyset$$

Finally, to allow conditional control structures it is also necessary to introduce some sort of boolean expressions. The ones considered here include equality tests for value expressions together with the standard classical logic connectives.

Definition 8.7 (Boolean Expressions)**.** Boolean expressions B consist of a logical combination of equality tests:

$$B ::= \text{false} \mid B \Rightarrow B \mid E_{\mathcal{T}} = E_{\mathcal{T}} \mid E_{\mathcal{A}} = E_{\mathcal{A}} \mid E_{\mathcal{P}} = E_{\mathcal{P}}$$

The valuation of a boolean expression on a store s is written $[\![B]\!]\mathsf{s}$ and given by:

$$[\![\text{false}]\!]\mathsf{s} = \text{false} \qquad [\![E_{\mathcal{T}} = F_{\mathcal{T}}]\!]\mathsf{s} = [\![E_{\mathcal{T}}]\!]\mathsf{s} \equiv [\![F_{\mathcal{T}}]\!]\mathsf{s}$$
$$[\![B_1 \Rightarrow B_2]\!]\mathsf{s} = [\![B_1]\!]\mathsf{s} \Rightarrow [\![B_2]\!]\mathsf{s} \qquad [\![E_{\mathcal{A}} = F_{\mathcal{A}}]\!]\mathsf{s} = [\![E_{\mathcal{A}}]\!]\mathsf{s} = [\![F_{\mathcal{A}}]\!]\mathsf{s}$$
$$[\![E_{\mathcal{P}} = F_{\mathcal{P}}]\!]\mathsf{s} = [\![E_{\mathcal{P}}]\!]\mathsf{s} = [\![F_{\mathcal{P}}]\!]\mathsf{s}$$

8.3 Local Query Languages

The key language extension presented in this chapter is the introduction of commands that act at multiple locations. This is primarily achieved by incorporating node queries into the language. Node queries can be viewed abstractly as functions of state that select and return a set of locations from that state. A typical example of node queries is path expressions, which select nodes by analysing label values.

Just as for update commands, the focus on local reasoning makes it necessary to place a locality restriction on queries. This non-standard restriction corresponds closely to the one on commands, and turns out to be quite natural, giving a sense of local behaviour that is particularly useful in a distributed or concurrent context. The restriction states that any successful execution of a query must be unaffected by the (non-local) context of execution. Thus, for example, a query $*$ that returns all the nodes in a tree is not local, since its result would increase by placing the tree in a larger context; in contrast, the query $\mathsf{n}/*$, which returns all the nodes underneath some location n, is local. One consequence of the locality condition is the need for query errors, corresponding to memory faults, to describe the result of a query that depends on a missing part of the tree: for example, the query $\mathsf{n}/*$ when n is not in

the tree, or a query which tries to follow a dangling pointer. Note that this is quite distinct from queries that return no nodes, but stay within the bounds of the current tree, such as the case for $\mathsf{n}/*$ when n has no nodes underneath it.

Definition 8.8 (Local query)**.** A *query* q is a function $q : \mathcal{S} \times \mathcal{T} \to \mathcal{P}_{\text{fin}}(\mathcal{N}) \cup \{\text{error}\}$, where $q(\mathsf{s},\mathsf{t}) \in \mathcal{P}_{\text{fin}}(\mathcal{N})$ implies $q(\mathsf{s},\mathsf{t}) \subseteq \text{locs}(\mathsf{t})$. A query q is *local* iff $\forall \mathsf{s} \in \mathcal{S}, \mathsf{t} \in \mathcal{T}, \mathsf{c} \in \mathcal{C}.\, q(\mathsf{s},\mathsf{t}) \in \mathcal{P}_{\text{fin}}(\mathcal{N}) \wedge \text{ap}(\mathsf{c},\mathsf{t})\downarrow$ implies $q(\mathsf{s},\text{ap}(\mathsf{c},\mathsf{t})) = q(\mathsf{s},\mathsf{t})$.

The technical results that follow all work using this abstract definition of queries. However, it is interesting to consider a concrete local query language, both to help with examples and demonstrate that little expressive power has been lost by moving to local queries. The language, motivated by XPATH [XPA99], is based on the idea of 'rooted' path expressions, where expressions all start from a given set of nodes.

Definition 8.9 (Path Query Language)**.** The Path Query Language (PQL) is defined as follows:

$$\begin{array}{rll} q ::= p/\pi \mid q \cup q \mid q \cap q \mid q - q & & \text{query} \\ \pi ::= \pi_a :: \pi_f \mid \pi/\pi \mid \pi \cup \pi \mid \pi \cap \pi \mid \pi - \pi & & \text{path} \\ \pi_a ::= \text{self} \mid \text{child} \mid \text{desc.} \mid \text{parent} \mid \text{link} & & \text{path axis} \\ \pi_f ::= E_{\mathcal{A}} \mid * & & \text{label filter} \end{array}$$

The evaluation of a query on a given store and tree is given by:

$$\begin{array}{rl} (p/\pi)(\mathsf{s},\mathsf{t}) = [\![\pi]\!]_{\mathsf{s},\mathsf{t}}(\mathsf{s}(p)) & \text{if } \mathsf{s}(p) \subseteq \text{locs}(\mathsf{t}), \text{ error otherwise} \\ \left.\begin{array}{l} (q_1 \cup q_2)(\mathsf{s},\mathsf{t}) = q_1(\mathsf{s},\mathsf{t}) \cup q_2(\mathsf{s},\mathsf{t}) \\ (q_1 \cap q_2)(\mathsf{s},\mathsf{t}) = q_1(\mathsf{s},\mathsf{t}) \cap q_2(\mathsf{s},\mathsf{t}) \\ (q_1 - q_2)(\mathsf{s},\mathsf{t}) = q_1(\mathsf{s},\mathsf{t}) \setminus q_2(\mathsf{s},\mathsf{t}) \end{array}\right\} & \begin{array}{l} \text{if } q_1(\mathsf{s},\mathsf{t}) \neq \text{error} \wedge q_1(\mathsf{s},\mathsf{t}) \neq \text{error}, \\ \text{error otherwise} \end{array} \end{array}$$

where $[\![\pi]\!]_{\mathsf{s},\mathsf{t}}$ is the state-dependent function defined in Figure 8.1.

A basic PQL query p/π is 'rooted' at an initial set of nodes given by a variable p, and consists of a combination of path steps $\pi_a :: \pi_f$. Each step consists of a path axis π_a, which describes the next 'movement' direction, and a path filter π_f, which provides a label condition for the movements. The path axes are standard XPATH axes (self, child, descendant and parent), with the addition of a link axis, which follows all the pointers from the current nodes. Unlike in XPATH, an ancestor axis is not included as it would break the locality condition.

$$
\begin{aligned}
[\![\pi_a :: \pi_f]\!]_{\mathsf{s,t}}(\mathsf{p}) &= [\![\pi_f]\!]_{\mathsf{s,t}}([\![\pi_a]\!]_{\mathsf{s,t}}(\mathsf{p})) \\
[\![\pi_1/\pi_2]\!]_{\mathsf{s,t}}(\mathsf{p}) &= [\![\pi_2]\!]_{\mathsf{s,t}}([\![\pi_1]\!]_{\mathsf{s,t}}(\mathsf{p})) \\
[\![\pi_1 \cup \pi_2]\!]_{\mathsf{s,t}}(\mathsf{p}) &= [\![\pi_1]\!]_{\mathsf{s,t}}(\mathsf{p}) \cup [\![\pi_2]\!]_{\mathsf{s,t}}(\mathsf{p}) \\
[\![\pi_1 \cap \pi_2]\!]_{\mathsf{s,t}}(\mathsf{p}) &= [\![\pi_1]\!]_{\mathsf{s,t}}(\mathsf{p}) \cap [\![\pi_2]\!]_{\mathsf{s,t}}(\mathsf{p}) \\
[\![\pi_1 - \pi_2]\!]_{\mathsf{s,t}}(\mathsf{p}) &= [\![\pi_1]\!]_{\mathsf{s,t}}(\mathsf{p}) \setminus [\![\pi_2]\!]_{\mathsf{s,t}}(\mathsf{p})
\end{aligned}
$$

$$
\begin{aligned}
[\![\mathrm{self}]\!]_{\mathsf{s,t}}(\mathsf{p}) &= \mathsf{p} \\
[\![\mathrm{child}]\!]_{\mathsf{s,t}}(\mathsf{p}) &= \{\mathsf{n}_2 \mid \exists \mathsf{n}_1 \in \mathsf{p}.\, \mathsf{t} \equiv \mathrm{ap}(\mathsf{c}, \mathsf{n}_{1\,\mathsf{a}_1:\mathsf{p}_1}[\, \mathsf{n}_{2\,\mathsf{a}_2:\mathsf{p}_2}[\mathsf{t}_2] \mid \mathsf{t}_1\,])\} \\
[\![\mathrm{desc.}]\!]_{\mathsf{s,t}}(\mathsf{p}) &= \{\mathsf{n}_2 \mid \exists \mathsf{n}_1 \in \mathsf{p}.\, \mathsf{t} \equiv \mathrm{ap}(\mathsf{c}_1, \mathsf{n}_{1\,\mathsf{a}_1:\mathsf{p}_1}[\, \mathrm{ap}(\mathsf{c}_2, \mathsf{n}_{2\,\mathsf{a}_2:\mathsf{p}_2}[\mathsf{t}_2])\,])\} \\
[\![\mathrm{parent}]\!]_{\mathsf{s,t}}(\mathsf{p}) &= \text{error if } \exists \mathsf{n}_2 \in \mathsf{p}.\, \mathsf{t} \equiv \mathsf{n}_{2\,\mathsf{a}_2:\mathsf{p}_2}[\mathsf{t}_2] \mid \mathsf{t}_1 \\
&\quad \text{else } \{\mathsf{n}_1 \mid \exists \mathsf{n}_2 \in \mathsf{p}.\, \mathsf{t} \equiv \mathrm{ap}(\mathsf{c}, \mathsf{n}_{1\,\mathsf{a}_1:\mathsf{p}_1}[\, \mathsf{n}_{2\,\mathsf{a}_2:\mathsf{p}_2}[\mathsf{t}_2] \mid \mathsf{t}_1\,])\} \\
[\![\mathrm{link}]\!]_{\mathsf{s,t}}(\mathsf{p}) &= \text{error if } \exists \mathsf{n}_1 \in \mathsf{p}, \mathsf{p}_1 \in \mathcal{P}_{\mathrm{fin}}(\mathcal{N}).\, \mathsf{t} \equiv \mathrm{ap}(\mathsf{c}, \mathsf{n}_{1\,\mathsf{a}_1:\mathsf{p}_1}[\mathsf{t}_1]) \wedge \exists \mathsf{n}_2 \in \mathsf{p}_1.\, \mathsf{n}_2 \notin \mathrm{locs}(\mathsf{t}) \\
&\quad \text{else } \{\mathsf{n}_2 \mid \exists \mathsf{n}_1 \in \mathsf{p}, \mathsf{p}_1 \in \mathcal{P}_{\mathrm{fin}}(\mathcal{N}).\, \mathsf{t} \equiv \mathrm{ap}(\mathsf{c}, \mathsf{n}_{1\,\mathsf{a}_1:\mathsf{p}_1}[\mathsf{t}_1]) \wedge \mathsf{n}_2 \in \mathsf{p}_1\}
\end{aligned}
$$

$$
\begin{aligned}
[\![E_{\mathcal{A}}]\!]_{\mathsf{s,t}}(\mathsf{p}) &= \{\mathsf{n} \in \mathsf{p} \mid \mathsf{t} \equiv \mathrm{ap}(\mathsf{c}, \mathsf{n}_{[\![E_{\mathcal{A}}]\!]\mathsf{s}:\mathsf{p}}[\mathsf{t}'])\} \\
[\![*]\!]_{\mathsf{s,t}}(\mathsf{p}) &= \mathsf{p}
\end{aligned}
$$

$$
[\![\pi]\!]_{\mathsf{s,t}}(\mathrm{error}) = [\![\pi_a]\!]_{\mathsf{s,t}}(\mathrm{error}) = [\![\pi_f]\!]_{\mathsf{s,t}}(\mathrm{error}) = \mathrm{error}
$$

Figure 8.1: PQL Query Semantics

As noted before, trying to access a missing part of the tree always results in an error. Hence, all the initial nodes in p must be present in the tree. Similarly, it is not possible to move up the parent axis of a top-level node, or follow a dangling pointer.

Example 8.10 (Queries)**.** Below are some simple examples of PQL queries. These make use of some standard notational shorthand: they assume the child axis where no axis is given (taking π_f to mean $(\mathrm{child} :: \pi_f)$), and use '..' and '.' as syntactic sugar for $(\mathrm{parent} :: *)$ and $(\mathrm{self} :: *)$ respectively.

(a) p/city — all 'city'-labelled child nodes of p.

(b) $p/(\mathsf{city} \cup \mathrm{link} :: \mathsf{city})$ — all 'city'-labelled child nodes or pointer targets of p.

(c) $p/(\mathsf{city} \cup \mathrm{link} :: \mathsf{city})/\mathsf{capital}$ — all '$\mathsf{capital}$'-labelled child nodes of (b).

(d) $p/(../*-.)$ — all the siblings of p. Note that, as a result of the locality condition, this returns an error for top-level nodes. This is unsurprising, since the siblings of such nodes clearly depend on the context.

8.4 Extended Update Language

This section presents a high-level update language for manipulating trees with pointers. The language has the same overall structure as the basic language BTU considered in Chapter 6, and satisfies the same locality condition (Defn. 6.10). The key differences are the presence of commands that act at multiple locations, commands that manipulate labels and pointers, and choice and loop control structures. The basic command types include assignment, lookup, update and new (just as in BTU), as well extra commands for moving trees. Of these, assignment, lookup and update have different forms for manipulating trees, labels and pointers, while new and move only act on trees. As in BTU, updates to trees can take place at either the tree level or the subforest level; in the case of the move command, this results in four possible behaviours (two at the source and two at the destination). To allow action at multiple locations, the target of an update is given by a pointer set, rather than a single node as for BTU.

Definition 8.11 (XTU commands)**.** The commands of the Extended Tree Update Language (XTU) are defined formally by following grammar:

$\mathbb{C} ::=$	$\mathbb{C}_{\text{up}}$	update commands
	$\mathbb{C}\ ;\ \mathbb{C}$	sequencing
	if B then $\mathbb{C}$ else $\mathbb{C}$	if-then-else
	while B do $\mathbb{C}$	while-do

The update commands $\mathbb{C}_{\text{up}}$ are given in Figure 8.2, where they are arranged by the type of the command and the part of the tree it affects. The boolean expressions B were defined in Defn. 8.7.

Before giving the operational semantics of the language, it is useful to first present an informal description of its behaviour. As for BTU, the locality condition means that update commands that try to access missing parts of the tree must produce a fault. Beyond that, the behaviour of the different command types is as follows:

	Trees	Subforests
Assign:	$x := E_{\mathcal{T}}$	
Lookup:	$x :=$ get-trees at p	$x :=$ get-subforests at p
Update:	dispose-trees at p	dispose-subforests at p
	append-trees $E_{\mathcal{T}}$ at p	append-subforests $E_{\mathcal{T}}$ at p
New:	$p' :=$ new-trees at p	$p' :=$ new-subforests at p
Move:	move-tree p_1 to-tree p_2	move-tree p_1 to-subforest p_2
	move-subforest p_1 to-tree p_2	move-subforest p_1 to-subforest p_2

	Pointers	Labels
Assign:	$p := E_{\mathcal{P}}$	$a := E_{\mathcal{A}}$
	$p := q$	
Lookup:	$p' :=$ get-links at p	$a :=$ get-labels at p
Update:	dispose-links at p	set-labels $E_{\mathcal{A}}$ at p
	append-links $E_{\mathcal{P}}$ at p	

Figure 8.2: XTU Update Commands

Assign: assigns the value of an expression to a variable of the appropriate type. There is an additional case $p := q$ for queries, which evaluates a query q on the current tree and assigns the resulting set of nodes to a pointer variable p.

Lookup: obtains tree, label or pointer values from the locations specified by a pointer set p and assigns them to a variable. Since searching a set of locations results in a set of values, there is the question of how to combine these into a single value that can be stored and used. A different solution is employed for each type: for trees and subforests, lookup concatenates the resulting subtrees, renaming nodes if necessary; for pointer sets, it returns the union of the results; and for labels, it picks a random label from the resulting set, using the default label $\star$ if the set is empty. These solutions are chosen for their simplicity. A more abstract approach is also possible.

Update: updates the tree, label or pointer values at the locations specified by a pointer set p. For trees and pointers, two types of update are possible: disposing the values at the locations, or appending to them a new value given by an expression. In the case of trees, this may involve renaming the nodes and internal pointers in the appended tree, similarly to Section 6.7. Label update involves replacing old labels with a new one.

New: creates new nodes at the locations specified by a pointer set p. The nodes have fresh identifiers, which are stored in a pointer set variable p', the default label $\star$, and no pointers or subtrees.

Move: moves a tree from a *single* source location to a *single* destination location, without renaming any node identifiers. This allows the preservation of external references and pointers, which cannot be achieved using the node-renaming tree update commands. The lack of renaming, however, means that both the source and target locations must consist of just one node, since it is not possible to make multiple copies of trees with unique identifiers. Furthermore, for the move to even be possible, the target location must not be a descendant of the source.

Definition 8.12 (XTU operational semantics)**.** The operational semantics of XTU is given in Figure 8.3. As in Defn. 6.3, this uses an evaluation relation $\rightsquigarrow$, defined on configuration triples $\mathbb{C}, \mathsf{s}, \mathsf{t}$, terminal states s, t and memory faults. Additionally, updates that act at multiple locations use partial computation states $\mathbb{C}, \mathsf{p}, \mathsf{s}, \mathsf{t}$, representing states where the command $\mathbb{C}$ has yet to act at the locations given by the pointer set p.

Note that the semantics for the partial computation states $\mathbb{C}, \mathsf{p}, \mathsf{s}, \mathsf{t}$ is given in a non-deterministic fashion that does not specify the order of execution. For example, the semantics of tree disposal describes the successive removal of subtrees of the form $\mathsf{n}_{\mathsf{a}:\mathsf{p}'}[\mathsf{t}']$, where the subtree locations n are picked in an arbitrary order from the argument p. This is perfectly fine, since for any successful execution, the order in which the nodes were chosen does not affect the final result. Some executions do diverge: it is not possible to dispose a location if any of its ancestors have already been disposed. The semantics, however, is only concerned with the existence of a terminating order, and since all the nodes in p must be in the tree, there is always at least one.

Assign :

$$\frac{[\![E_{\mathcal{T}}]\!]\mathsf{s} \equiv \mathsf{t}'}{x := E_{\mathcal{T}}, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathsf{s}[x \leftarrow \mathsf{t}'], \mathsf{t}} \qquad \frac{[\![E_{\mathcal{A}}]\!]\mathsf{s} = \mathsf{a}}{a := E_{\mathcal{A}}, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathsf{s}[a \leftarrow \mathsf{a}], \mathsf{t}}$$

$$\frac{[\![E_{\mathcal{P}}]\!]\mathsf{s} = \mathsf{p}'}{p := E_{\mathcal{P}}, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathsf{s}[p \leftarrow \mathsf{p}'], \mathsf{t}} \qquad \frac{q(\mathsf{s}, \mathsf{t}) = \mathsf{p}'}{p := q, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathsf{s}[p \leftarrow \mathsf{p}'], \mathsf{t}} \qquad \frac{q(\mathsf{s}, \mathsf{t}) = \text{error}}{p := q, \mathsf{s}, \mathsf{t} \rightsquigarrow \text{fault}}$$

Lookup
Update :
New

$\mathbb{C}$ below is any lookup, update or new command acting at locations p
the individual semantics for the commands is given on the next page

$$\frac{[\![p]\!]\mathsf{s} \not\subseteq \text{locs}(\mathsf{t})}{\mathbb{C}, \mathsf{s}, \mathsf{t} \rightsquigarrow \text{fault}} \qquad \frac{[\![p]\!]\mathsf{s} = \mathsf{p} \subseteq \text{locs}(\mathsf{t})}{\mathbb{C}, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathbb{C}, \mathsf{p}, \mathsf{s}, \mathsf{t}} \; \text{mod}(\mathbb{C}) = \emptyset$$

$$\frac{[\![p]\!]\mathsf{s} = \mathsf{p} \subseteq \text{locs}(\mathsf{t})}{\mathbb{C}, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathbb{C}, \mathsf{p}, \mathsf{s}[x \leftarrow 0], \mathsf{t}} \; x \in \text{mod}(\mathbb{C}) \qquad \frac{[\![p]\!]\mathsf{s} = \mathsf{p} \subseteq \text{locs}(\mathsf{t})}{\mathbb{C}, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathbb{C}, \mathsf{p}, \mathsf{s}[p \leftarrow \emptyset], \mathsf{t}} \; p \in \text{mod}(\mathbb{C})$$

$$\frac{[\![p]\!]\mathsf{s} = \mathsf{p} \subseteq \text{locs}(\mathsf{t})}{\mathbb{C}, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathbb{C}, \mathsf{p}, \mathsf{s}[a \leftarrow \star], \mathsf{t}} \; a \in \text{mod}(\mathbb{C}) \qquad \frac{}{\mathbb{C}, \emptyset, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathsf{s}, \mathsf{t}}$$

Move : for $\mathbb{C}$ = move-tree p_2 to-subforest p_1 ; other moves are similar

$$\frac{\mathsf{s}(p_1) = \{\mathsf{n}_1\} \quad \mathsf{s}(p_2) = \{\mathsf{n}_2\} \quad \mathsf{t} \equiv \text{ap}(\mathsf{c}_2, \mathsf{n}_{2\mathsf{a}_2:\mathsf{p}_2'}[\text{ap}(\mathsf{c}_1, \mathsf{n}_{1\mathsf{a}_1:\mathsf{p}_1'}[\mathsf{t}_1])])}{\mathbb{C}, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathsf{s}, \text{ap}(\mathsf{c}_2, \mathsf{n}_{2\mathsf{a}_2:\mathsf{p}_2'}[\text{ap}(\mathsf{c}_1, 0) \mid \mathsf{n}_{1\mathsf{a}_1:\mathsf{p}_1'}[\mathsf{t}_1]])}$$

$$\frac{\mathsf{s}(p_1) = \{\mathsf{n}_1\} \quad \mathsf{s}(p_2) = \{\mathsf{n}_2\} \quad \mathsf{t} \equiv \text{ap}(\mathsf{c}, \text{ap}(\mathsf{c}_1, \mathsf{n}_{1\mathsf{a}_1:\mathsf{p}_1'}[\mathsf{t}_1]) \mid \text{ap}(\mathsf{c}_2, \mathsf{n}_{2\mathsf{a}_2:\mathsf{p}_2'}[\mathsf{t}_2]))}{\mathbb{C}, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathsf{s}, \text{ap}(\mathsf{c}, \text{ap}(\mathsf{c}_2, \mathsf{n}_{2\mathsf{a}_2:\mathsf{p}_2'}[\mathsf{t}_2 \mid \mathsf{n}_{1\mathsf{a}_1:\mathsf{p}_1'}[\mathsf{t}_1]] \mid \text{ap}(\mathsf{c}_1, 0)))}$$

$$\frac{\text{otherwise}}{\mathbb{C}, \mathsf{s}, \mathsf{t} \rightsquigarrow \text{fault}}$$

Control :

$$\frac{\mathbb{C}_1, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathbb{C}', \mathsf{s}', \mathsf{t}'}{(\mathbb{C}_1 \; ; \mathbb{C}_2), \mathsf{s}, \mathsf{t} \rightsquigarrow (\mathbb{C}' \; ; \mathbb{C}_2), \mathsf{s}', \mathsf{t}'} \qquad \frac{\mathbb{C}_1, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathsf{s}', \mathsf{t}'}{(\mathbb{C}_1 \; ; \mathbb{C}_2), \mathsf{s}, \mathsf{t} \rightsquigarrow \mathbb{C}_2, \mathsf{s}', \mathsf{t}'}$$

$$\frac{\mathbb{C}_1, \mathsf{s}, \mathsf{t} \rightsquigarrow \text{fault}}{(\mathbb{C}_1 \; ; \mathbb{C}_2), \mathsf{s}, \mathsf{t} \rightsquigarrow \text{fault}} \qquad \frac{[\![B]\!]\mathsf{s} = \text{true}}{\text{if } B \text{ then } \mathbb{C}_1 \text{ else } \mathbb{C}_2, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathbb{C}_1, \mathsf{s}, \mathsf{t}}$$

$$\frac{[\![B]\!]\mathsf{s} = \text{false}}{\text{if } B \text{ then } \mathbb{C}_1 \text{ else } \mathbb{C}_2, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathbb{C}_2, \mathsf{s}, \mathsf{t}} \qquad \frac{[\![B]\!]\mathsf{s} = \text{false}}{\text{while } B \text{ do } \mathbb{C}, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathsf{s}, \mathsf{t}}$$

$$\frac{[\![B]\!]\mathsf{s} = \text{true}}{\text{while } B \text{ do } \mathbb{C}, \mathsf{s}, \mathsf{t} \rightsquigarrow (\mathbb{C} \; ; \text{while } B \text{ do } \mathbb{C}), \mathsf{s}, \mathsf{t}}$$

Figure 8.3: XTU Operational Semantics (1/2)

$$\text{dispose-trees at } p: \quad \frac{\mathsf{t} \equiv \mathrm{ap}(\mathsf{c}, \mathsf{n}_{\mathsf{a}:\mathsf{p}'}[\mathsf{t}'])}{\mathbb{C}, \{\mathsf{n}\} \uplus \mathsf{p}, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathbb{C}, \mathsf{p}, \mathsf{s}, \mathrm{ap}(\mathsf{c}, 0)}$$

$$\text{dispose-subforests at } p: \quad \frac{\mathsf{t} \equiv \mathrm{ap}(\mathsf{c}, \mathsf{n}_{\mathsf{a}:\mathsf{p}'}[\mathsf{t}'])}{\mathbb{C}, \{\mathsf{n}\} \uplus \mathsf{p}, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathbb{C}, \mathsf{p}, \mathsf{s}, \mathrm{ap}(\mathsf{c}, \mathsf{n}_{\mathsf{a}:\mathsf{p}'}[0])}$$

$$\text{dispose-links at } p: \quad \frac{\mathsf{t} \equiv \mathrm{ap}(\mathsf{c}, \mathsf{n}_{\mathsf{a}:\mathsf{p}'}[\mathsf{t}'])}{\mathbb{C}, \{\mathsf{n}\} \uplus \mathsf{p}, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathbb{C}, \mathsf{p}, \mathsf{s}, \mathrm{ap}(\mathsf{c}, \mathsf{n}_{\mathsf{a}:\emptyset}[\mathsf{t}'])}$$

$$\text{append-trees } E_{\mathcal{T}} \text{ at } p: \quad \frac{\mathsf{t} \equiv \mathrm{ap}(\mathsf{c}, \mathsf{n}_{\mathsf{a}:\mathsf{p}'}[\mathsf{t}']) \quad \mathsf{t}'' \simeq [\![E_{\mathcal{T}}]\!]\mathsf{s} \quad \mathsf{t}'' \# \mathsf{t}}{\mathbb{C}, \{\mathsf{n}\} \uplus \mathsf{p}, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathbb{C}, \mathsf{p}, \mathsf{s}, \mathrm{ap}(\mathsf{c}, \mathsf{n}_{\mathsf{a}:\mathsf{p}'}[\mathsf{t}' \mid \mathsf{t}''])}$$

$$\text{append-subforests } E_{\mathcal{T}} \text{ at } p: \quad \frac{\mathsf{t} \equiv \mathrm{ap}(\mathsf{c}, \mathsf{n}_{\mathsf{a}:\mathsf{p}'}[\mathsf{t}']) \quad \mathsf{t}'' \simeq [\![E_{\mathcal{T}}]\!]\mathsf{s} \quad \mathsf{t}'' \# \mathsf{t}}{\mathbb{C}, \{\mathsf{n}\} \uplus \mathsf{p}, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathbb{C}, \mathsf{p}, \mathsf{s}, \mathrm{ap}(\mathsf{c}, \mathsf{n}_{\mathsf{a}:\mathsf{p}'}[\mathsf{t}' \mid \mathsf{t}''])}$$

$$\text{append-links } E_{\mathcal{P}} \text{ at } p: \quad \frac{\mathsf{t} \equiv \mathrm{ap}(\mathsf{c}, \mathsf{n}_{\mathsf{a}:\mathsf{p}'}[\mathsf{t}'])}{\mathbb{C}, \{\mathsf{n}\} \uplus \mathsf{p}, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathbb{C}, \mathsf{p}, \mathsf{s}, \mathrm{ap}(\mathsf{c}, \mathsf{n}_{\mathsf{a}:\mathsf{p}' \cup [\![E_{\mathcal{P}}]\!]\mathsf{s}}[\mathsf{t}'])}$$

$$\text{set-labels } E_{\mathcal{A}} \text{ at } p: \quad \frac{\mathsf{t} \equiv \mathrm{ap}(\mathsf{c}, \mathsf{n}_{\mathsf{a}:\mathsf{p}'}[\mathsf{t}'])}{\mathbb{C}, \{\mathsf{n}\} \uplus \mathsf{p}, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathbb{C}, \mathsf{p}, \mathsf{s}, \mathrm{ap}(\mathsf{c}, \mathsf{n}_{[\![E_{\mathcal{A}}]\!]\mathsf{s}:\mathsf{p}'}[\mathsf{t}'])}$$

$$x := \text{get-trees at } p: \quad \frac{\mathsf{t} \equiv \mathrm{ap}(\mathsf{c}, \mathsf{n}_{\mathsf{a}:\mathsf{p}'}[\mathsf{t}']) \quad \mathsf{t}'' \simeq \mathsf{n}_{\mathsf{a}:\mathsf{p}'}[\mathsf{t}'] \quad \mathsf{t}'' \# \mathsf{s}(x)}{\mathbb{C}, \{\mathsf{n}\} \uplus \mathsf{p}, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathbb{C}, \mathsf{p}, \mathsf{s}[x \leftarrow \mathsf{s}(x) \mid \mathsf{t}''], \mathsf{t}}$$

$$x := \text{get-subforests at } p: \quad \frac{\mathsf{t} \equiv \mathrm{ap}(\mathsf{c}, \mathsf{n}_{\mathsf{a}:\mathsf{p}'}[\mathsf{t}']) \quad \mathsf{t}'' \simeq \mathsf{t}' \quad \mathsf{t}'' \# \mathsf{s}(x)}{\mathbb{C}, \{\mathsf{n}\} \uplus \mathsf{p}, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathbb{C}, \mathsf{p}, \mathsf{s}[x \leftarrow \mathsf{s}(x) \mid \mathsf{t}''], \mathsf{t}}$$

$$p' := \text{get-links at } p: \quad \frac{\mathsf{t} \equiv \mathrm{ap}(\mathsf{c}, \mathsf{n}_{\mathsf{a}:\mathsf{p}'}[\mathsf{t}'])}{\mathbb{C}, \{\mathsf{n}\} \uplus \mathsf{p}, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathbb{C}, \mathsf{p}, \mathsf{s}[p \leftarrow \mathsf{s}(p) \cup \mathsf{p}'], \mathsf{t}}$$

$$a := \text{get-labels at } p: \quad \frac{\mathsf{t} \equiv \mathrm{ap}(\mathsf{c}, \mathsf{n}_{\mathsf{a}:\mathsf{p}'}[\mathsf{t}'])}{\mathbb{C}, \{\mathsf{n}\} \uplus \mathsf{p}, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathbb{C}, \mathsf{p}, \mathsf{s}[a \leftarrow \mathsf{a}], \mathsf{t}}$$

$$p' := \text{new-trees at } p: \quad \frac{\mathsf{t} \equiv \mathrm{ap}(\mathsf{c}, \mathsf{n}_{\mathsf{a}:\mathsf{p}'}[\mathsf{t}']) \quad \mathsf{n}' \notin \mathrm{locs}(\mathsf{t}) \quad \mathsf{t}'' \equiv \mathsf{n}_{\mathsf{a}:\mathsf{p}'}[\mathsf{t}'] \mid \mathsf{n}'_{\star:\emptyset}[0]}{\mathbb{C}, \{\mathsf{n}\} \uplus \mathsf{p}, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathbb{C}, \mathsf{p}, \mathsf{s}[p' \leftarrow \mathsf{s}(p') \cup \mathsf{n}'], \mathrm{ap}(\mathsf{c}, \mathsf{t}'')}$$

$$p' := \text{new-subforests at } p: \quad \frac{\mathsf{t} \equiv \mathrm{ap}(\mathsf{c}, \mathsf{n}_{\mathsf{a}:\mathsf{p}'}[\mathsf{t}']) \quad \mathsf{n}' \notin \mathrm{locs}(\mathsf{t}) \quad \mathsf{t}'' \equiv \mathsf{n}_{\mathsf{a}:\mathsf{p}'}[\mathsf{t}' \mid \mathsf{n}'_{\star:\emptyset}[0]]}{\mathbb{C}, \{\mathsf{n}\} \uplus \mathsf{p}, \mathsf{s}, \mathsf{t} \rightsquigarrow \mathbb{C}, \mathsf{p}, \mathsf{s}[p' \leftarrow \mathsf{s}(p') \cup \mathsf{n}'], \mathrm{ap}(\mathsf{c}, \mathsf{t}'')}$$

Figure 8.3: XTU Operational Semantics (2/2)

Observation 8.13. The commands in XTU are all local, as per Defn. 6.10.

Example 8.14 (Collapse links)**.** The following are two simple example programs in XTU. Both programs collapse the links in a tree by copying the subtrees at the targets to the subtree of the source, renaming the nodes and any internal pointers. The first program collapses all the links at a location p; the second only collapses the links given by a variable p'. The diagram on the right shows the action of 'collapse-all-links at n_1' on the tree from Example 8.4.

collapse-all-links at $p \triangleq$

$p' := p/\text{link} :: *$;

$x :=$ get-trees at p' ;

append-subforests x at p ;

dispose-links at p

collapse-links p' at $p \triangleq$

$p'' := p/\text{link} :: * \cap p'$;

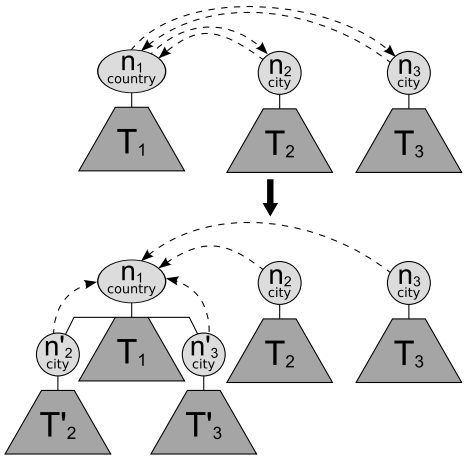
$x :=$ get-trees at p'' ;

append-subforests x at p ;

$p''' := p/\text{link} :: * - p'$;

dispose-links at p ;

append-links p''' at p



8.5 Context Logic Adaptation

The next step is the adaptation of Context Logic to the new tree data structure, incorporating pointers and labels, the different types of variables and expressions, and queries. This is a relatively straightforward generalisation of the CL_{tree} logic given in Defn. 5.24 and used in Chapter 6.

Before defining the logic, it is first necessary to extend the framework slightly to allow reasoning about individual node identifiers. The update language uses variables denoting sets of nodes, but does not refer directly to individual node identifiers. For reasoning about update, however, it is necessary to both mention and quantify over individual nodes. This is achieved by introducing an environment, separate from the storage model, which contains logical variables denoting locations. To allow comparisons between pointer sets and identifiers, it is also necessary to extend pointer set expressions to include these variables, as well as the basic set operations.

Definition 8.15 (Environment)**.** Given an infinite set $\mathrm{Var}_{\mathcal{N}} = \{m, n, \dots\}$ of location variables, an environment $\mathsf{e} \in \mathcal{E}$ is a total function $\mathsf{e} : \mathrm{Var}_{\mathcal{N}} \to \mathcal{N}$ returning the value of the variables.

Definition 8.16 (Extended pointer expressions)**.** Extended pointer expressions $\hat{E}_{\mathcal{P}}$ are given by the following grammar:

$$\hat{E}_{\mathcal{P}} ::= E_{\mathcal{P}} \mid \{n\} \mid \hat{E}_{\mathcal{P}} \cup \hat{E}_{\mathcal{P}} \mid \hat{E}_{\mathcal{P}} \cap \hat{E}_{\mathcal{P}}$$

with valuation on an environment e and store s written $[\![\hat{E}_{\mathcal{P}}]\!]\mathsf{es}$, and given by:

$$\begin{aligned}
[\![E_{\mathcal{P}}]\!]\mathsf{es} &= [\![E_{\mathcal{P}}]\!]\mathsf{s} \\
[\![\{n\}]\!]\mathsf{es} &= \{\mathsf{e}(n)\} \\
[\![\hat{E}_{\mathcal{P}} \cup \hat{F}_{\mathcal{P}}]\!]\mathsf{es} &= [\![\hat{E}_{\mathcal{P}}]\!]\mathsf{es} \cup [\![\hat{F}_{\mathcal{P}}]\!]\mathsf{es} \\
[\![\hat{E}_{\mathcal{P}} \cap \hat{F}_{\mathcal{P}}]\!]\mathsf{es} &= [\![\hat{E}_{\mathcal{P}}]\!]\mathsf{es} \cap [\![\hat{F}_{\mathcal{P}}]\!]\mathsf{es}
\end{aligned}$$

Just like $\mathrm{CL}_{\mathrm{tree}}$, the logic considered here is an extension of Context Logic with Zero, with additional formulæ introduced for reasoning about trees. The semantics comes from the new tree data structure, indexed this time by both the environment and the store.

Definition 8.17 ($\mathrm{CL}_{\mathrm{xtree}}$ formulæ)**.** The formulæ of Context Logic for Extended Trees ($\mathrm{CL}_{\mathrm{xtree}}$) consists of the same context and data assertions as $\mathrm{CL}_{\emptyset}$ with the addition of formulæ for expressions, query evaluation, node renaming, branching contexts, parallel composition contexts, and quantification over variables.

$P ::= p$	propositional variables
$\mid \boldsymbol{E_{\mathcal{T}}} \mid \boldsymbol{\hat{E}_{\mathcal{P}} \doteq q} \mid [\![\boldsymbol{P}]\!]$	**special formulæ**
$\mid 0 \mid K \cdot P \mid K \lhd P$	structural formulæ
$\mid P \Rightarrow P \mid \mathrm{false} \mid \exists \boldsymbol{x}.\boldsymbol{P} \mid \exists \boldsymbol{p}.\boldsymbol{P} \mid \exists \boldsymbol{a}.\boldsymbol{P} \mid \exists \boldsymbol{n}.\boldsymbol{P}$	**logical formulæ**
$K ::= k$	propositional variables
$\mid \boldsymbol{n_{E_{\mathcal{A}}:\hat{E}_{\mathcal{P}}}[K]} \mid \boldsymbol{(P \mid K)}$	**special formulæ**
$\mid I \mid P \rhd P$	structural formulæ
$\mid K \Rightarrow K \mid \mathrm{False} \mid \exists \boldsymbol{x}.\boldsymbol{K} \mid \exists \boldsymbol{p}.\boldsymbol{K} \mid \exists \boldsymbol{a}.\boldsymbol{K} \mid \exists \boldsymbol{n}.\boldsymbol{K}$	**logical formulæ**

Definition 8.18 ($\mathrm{CL}_{\mathrm{xtree}}$ semantics)**.** The semantics of $\mathrm{CL}_{\mathrm{xtree}}$ is an extension of the semantics of $\mathrm{CL}_{\emptyset}$ for the model $(\mathcal{C}, \mathcal{T}, \mathrm{ap}, \{-\}, \{0\})^{\mathcal{E}\times\mathcal{S}}$, where trees and tree contexts are indexed, as in Example 3.8, by both the environment and the store. Given an

$$
\begin{array}{ll}
\sigma, \mathsf{e}, \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} E_{\mathcal{T}} & \Leftrightarrow \mathsf{t} \equiv [\![E_{\mathcal{T}}]\!]\mathsf{s} \\
\sigma, \mathsf{e}, \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} \hat{E}_{\mathcal{P}} \doteq q & \Leftrightarrow q(\mathsf{s}, \mathsf{t}) = [\![\hat{E}_{\mathcal{P}}]\!]\mathsf{es} \\
\sigma, \mathsf{e}, \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} [\![P]\!] & \Leftrightarrow \exists \mathsf{t}' \in \mathcal{T}. (\mathsf{t} \simeq \mathsf{t}' \wedge \sigma, \mathsf{e}, \mathsf{s}, \mathsf{t}' \vDash_{\mathcal{T}} P) \\
\sigma, \mathsf{e}, \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} \exists x.P & \Leftrightarrow \exists \mathsf{t}' \in \mathcal{T}. \sigma, \mathsf{e}, [\mathsf{s}|x \mapsto \mathsf{t}'], \mathsf{t} \vDash_{\mathcal{T}} P \\
\sigma, \mathsf{e}, \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} \exists p.P & \Leftrightarrow \exists \mathsf{p} \in \mathcal{P}_{\text{fin}}(\mathcal{N}). \sigma, \mathsf{e}, [\mathsf{s}|p \mapsto \mathsf{p}], \mathsf{t} \vDash_{\mathcal{T}} P \\
\sigma, \mathsf{e}, \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} \exists a.P & \Leftrightarrow \exists \mathsf{a} \in \mathcal{A}. \sigma, \mathsf{e}, [\mathsf{s}|a \mapsto \mathsf{a}], \mathsf{t} \vDash_{\mathcal{T}} P \\
\sigma, \mathsf{e}, \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} \exists n.P & \Leftrightarrow \exists \mathsf{n} \in \mathcal{N}. \sigma, [\mathsf{e}|n \mapsto \mathsf{n}], \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} P \\
\\
\sigma, \mathsf{e}, \mathsf{s}, \mathsf{c} \vDash_{\mathcal{C}} n_{E_{\mathcal{A}}:\hat{E}_{\mathcal{P}}}[K] & \Leftrightarrow \exists \mathsf{c}' \in \mathcal{C}. \begin{pmatrix} \mathsf{c} \equiv \mathsf{e}(n)_{[\![E_{\mathcal{A}}]\!]\mathsf{s}:[\![\hat{E}_{\mathcal{P}}]\!]\mathsf{s}}[\mathsf{c}'] \wedge \\ \sigma, \mathsf{e}, \mathsf{s}, \mathsf{c}' \vDash_{\mathcal{C}} K \wedge \mathsf{e}(n) \notin \text{locs}(\mathsf{c}') \end{pmatrix} \\
\sigma, \mathsf{e}, \mathsf{s}, \mathsf{c} \vDash_{\mathcal{C}} P \mid K & \Leftrightarrow \exists \mathsf{t} \in \mathcal{T}, \mathsf{c}' \in \mathcal{C}. \begin{pmatrix} \mathsf{c} \equiv \mathsf{t} \mid \mathsf{c}' \wedge \sigma, \mathsf{e}, \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} P \\ \wedge \sigma, \mathsf{e}, \mathsf{s}, \mathsf{c}' \vDash_{\mathcal{C}} K \wedge \mathsf{t} \# \mathsf{c}' \end{pmatrix} \\
\sigma, \mathsf{e}, \mathsf{s}, \mathsf{c} \vDash_{\mathcal{C}} \exists x.K & \Leftrightarrow \exists \mathsf{t} \in \mathcal{T}. \sigma, \mathsf{e}, [\mathsf{s}|x \mapsto \mathsf{t}], \mathsf{c} \vDash_{\mathcal{C}} K \\
\sigma, \mathsf{e}, \mathsf{s}, \mathsf{c} \vDash_{\mathcal{C}} \exists p.K & \Leftrightarrow \exists \mathsf{p} \in \mathcal{P}_{\text{fin}}(\mathcal{N}). \sigma, \mathsf{e}, [\mathsf{s}|p \mapsto \mathsf{p}], \mathsf{c} \vDash_{\mathcal{C}} K \\
\sigma, \mathsf{e}, \mathsf{s}, \mathsf{c} \vDash_{\mathcal{C}} \exists a.K & \Leftrightarrow \exists \mathsf{a} \in \mathcal{A}. \sigma, \mathsf{e}, [\mathsf{s}|a \mapsto \mathsf{a}], \mathsf{c} \vDash_{\mathcal{C}} K \\
\sigma, \mathsf{e}, \mathsf{s}, \mathsf{c} \vDash_{\mathcal{C}} \exists n.K & \Leftrightarrow \exists \mathsf{n} \in \mathcal{N}. \sigma, [\mathsf{e}|n \mapsto \mathsf{n}], \mathsf{s}, \mathsf{c} \vDash_{\mathcal{C}} K
\end{array}
$$

Figure 8.4: CL_{xtree} Semantics

interpretation function $\sigma : (\mathcal{V}_{\mathcal{D}} \to \mathcal{P}(\mathcal{E} \times \mathcal{S} \times \mathcal{T})) \times (\mathcal{V}_{\mathcal{C}} \to \mathcal{P}(\mathcal{E} \times \mathcal{S} \times \mathcal{C}))$ mapping propositional variables to sets of trees and contexts indexed by environments and stores, the forcing semantics is given by two satisfaction relations $\sigma, \mathsf{e}, \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} P$ and $\sigma, \mathsf{e}, \mathsf{s}, \mathsf{c} \vDash_{\mathcal{C}} K$ for trees and contexts, defined as in Defn. 4.12 and extended as in Figure 8.4.

Of the new formulæ, all but query evaluation were also present in CL_{tree} in some form or other: variable quantification and node renaming are essentially unchanged; tree expression assertions correspond to tree variable assertions; and the context formulæ denoting the branching structure now also include expressions to represent node labels and pointers. The one genuinely new primitive is the query evaluation $\hat{E}_{\mathcal{P}} \doteq q$. This states that the query q returns the pointer set $\hat{E}_{\mathcal{P}}$ when evaluated on the current tree and store.

As in other models of Context Logic, it is possible to define various standard

derived formulæ, such as the existential adjoint duals ▶ and ◀ (Defn. 3.11) and the somewhere modality ◇ (Defn. 3.12). Additionally, it is useful to define a number of more specific derived formulæ for reasoning about expressions and variables. These are presented below in three sets: the first defines equality tests for the different types of expressions; the second introduces some useful formulæ for reasoning about variables; and the last presents some formulæ for reasoning about pointer expressions. The derivations are non-trivial, raising the possibility of adding some of the formulæ to the logic as basic assertions. Conversely, the derivations serve as a good illustration of the expressivity of Context Logic.

Definition 8.19 (Derived equality formulæ)**.** Equality for tree expressions, label expressions and extended pointer expressions can be defined as follows, using the validity connective from Defn. 3.15 just like in Section 5.5.

$$(E_{\mathcal{T}} = F_{\mathcal{T}}) \triangleq \mathrm{valid}_i(E_{\mathcal{T}} \Leftrightarrow F_{\mathcal{T}})$$
$$(E_{\mathcal{A}} = F_{\mathcal{A}}) \triangleq \mathrm{valid}_i(\forall n.\ n_{E_{\mathcal{A}}:\emptyset}[0] \Leftrightarrow n_{F_{\mathcal{A}}:\emptyset}[0])$$
$$(\hat{E}_{\mathcal{P}} = \hat{F}_{\mathcal{P}}) \triangleq \mathrm{valid}_i(\forall n.\ n_{\star:\hat{E}_{\mathcal{P}}}[0] \Leftrightarrow n_{\star:\hat{F}_{\mathcal{P}}}[0])$$

These have the following derived semantics:

$$\sigma, \mathsf{e}, \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} E_{\mathcal{T}} = F_{\mathcal{T}} \Leftrightarrow [\![E_{\mathcal{T}}]\!]\mathsf{s} \equiv [\![F_{\mathcal{T}}]\!]\mathsf{s}$$
$$\sigma, \mathsf{e}, \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} E_{\mathcal{A}} = F_{\mathcal{A}} \Leftrightarrow [\![E_{\mathcal{A}}]\!]\mathsf{s} = [\![F_{\mathcal{A}}]\!]\mathsf{s}$$
$$\sigma, \mathsf{e}, \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} \hat{E}_{\mathcal{P}} = \hat{F}_{\mathcal{P}} \Leftrightarrow [\![\hat{E}_{\mathcal{P}}]\!]\mathsf{es} = [\![\hat{F}_{\mathcal{P}}]\!]\mathsf{es}$$

Definition 8.20 (Derived variable formulæ)**.** The following are useful derived formulæ for dealing with variables:

$$n[K] \triangleq \exists a, p.\ n_{a:p}[K]$$
$$n \in \hat{E}_{\mathcal{P}} \triangleq (\hat{E}_{\mathcal{P}} \cap \{n\}) = \{n\}$$
$$n \in E_{\mathcal{T}} \triangleq \mathrm{valid}_i(E_{\mathcal{T}} \Rightarrow \Diamond n[\mathrm{true}])$$
$$x \in P \triangleq \mathrm{valid}_i(x \Rightarrow P)$$

The interpretations of these are straightforward: $n[K]$ describes a node n with arbitrary label and pointers; $n \in \hat{E}_{\mathcal{P}}$ and $n \in E_{\mathcal{T}}$ state that location n is in the pointer set $\hat{E}_{\mathcal{P}}$ or tree $E_{\mathcal{T}}$; and $x \in P$ states that tree x satisfies proposition P. The derived semantics is as follows:

$$\sigma, \mathsf{e}, \mathsf{s}, \mathsf{c} \vDash_{\mathcal{C}} n[K] \quad \Leftrightarrow \exists \mathsf{a}, \mathsf{p}, \mathsf{c}'.(\mathsf{c} \equiv \mathsf{e}(n)_{\mathsf{a}:\mathsf{p}}[\mathsf{c}'] \wedge \sigma, \mathsf{e}, \mathsf{s}, \mathsf{c}' \vDash_{\mathcal{C}} K \wedge \mathsf{e}(n) \notin \mathrm{locs}(\mathsf{c}'))$$
$$\sigma, \mathsf{e}, \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} n \in \hat{E}_{\mathcal{P}} \Leftrightarrow \mathsf{e}(n) \in [\![\hat{E}_{\mathcal{P}}]\!]\mathsf{es}$$
$$\sigma, \mathsf{e}, \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} n \in E_{\mathcal{T}} \Leftrightarrow \mathsf{e}(n) \in \mathrm{locs}([\![E_{\mathcal{T}}]\!]\mathsf{s})$$
$$\sigma, \mathsf{e}, \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} x \in P \quad \Leftrightarrow \sigma, \mathsf{e}, \mathsf{s}, \mathsf{s}(x) \vDash_{\mathcal{T}} P$$

Definition 8.21 (Derived expression formulæ)**.** The following are useful derived formulæ for dealing with extended pointer expressions:

$$\Diamond\hat{E}_{\mathcal{P}} \triangleq \forall n.n \in \hat{E}_{\mathcal{P}} \Rightarrow \Diamond n[\text{true}]$$
$$\hat{E}_{\mathcal{P}} = \hat{E}^1_{\mathcal{P}} \uplus \hat{E}^2_{\mathcal{P}} \triangleq (\hat{E}_{\mathcal{P}} = \hat{E}^1_{\mathcal{P}} \cup \hat{E}^2_{\mathcal{P}}) \wedge (\hat{E}^1_{\mathcal{P}} \cap \hat{E}^2_{\mathcal{P}} = \emptyset)$$
$$|\hat{E}^1_{\mathcal{P}}| = |\hat{E}^2_{\mathcal{P}}| \triangleq \exists x, y.\, x \in [\![y]\!] \wedge \forall n.(n \in \hat{E}^1_{\mathcal{P}} \Leftrightarrow n \in x \wedge n \in \hat{E}^2_{\mathcal{P}} \Leftrightarrow n \in y)$$

Again, the interpretations are simple: $\Diamond\hat{E}_{\mathcal{P}}$ says that all the locations in $\hat{E}_{\mathcal{P}}$ are in the tree, while $\hat{E}_{\mathcal{P}} = \hat{E}^1_{\mathcal{P}} \uplus \hat{E}^2_{\mathcal{P}}$ and $|\hat{E}^1_{\mathcal{P}}| = |\hat{E}^2_{\mathcal{P}}|$ should be self-explanatory. The second of these works by asserting the existence of two trees, x and y, that contain the same nodes as $\hat{E}^1_{\mathcal{P}}$ and $\hat{E}^2_{\mathcal{P}}$ respectively, and which are equivalent modulo renaming. The derived semantics of the formulæ is as follows:

$$\sigma, \mathsf{e}, \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} \Diamond\hat{E}_{\mathcal{P}} \qquad \Leftrightarrow \forall \mathsf{n}.\mathsf{n} \in [\![\hat{E}_{\mathcal{P}}]\!]\mathsf{es} \Rightarrow \mathsf{n} \in \mathrm{locs}(\mathsf{t})$$
$$\sigma, \mathsf{e}, \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} \hat{E}_{\mathcal{P}} = \hat{E}^1_{\mathcal{P}} \uplus \hat{E}^2_{\mathcal{P}} \Leftrightarrow [\![\hat{E}_{\mathcal{P}}]\!]\mathsf{es} = [\![\hat{E}^1_{\mathcal{P}}]\!]\mathsf{es} \uplus [\![\hat{E}^2_{\mathcal{P}}]\!]\mathsf{es}$$
$$\sigma, \mathsf{e}, \mathsf{s}, \mathsf{t} \vDash_{\mathcal{T}} |\hat{E}^1_{\mathcal{P}}| = |\hat{E}^2_{\mathcal{P}}| \qquad \Leftrightarrow |[\![\hat{E}^1_{\mathcal{P}}]\!]\mathsf{es}| = |[\![\hat{E}^2_{\mathcal{P}}]\!]\mathsf{es}|$$

Note that all the derived formulæ except $n[K]$ and $\Diamond\hat{E}_{\mathcal{P}}$ are pure (Defn. 3.16c): that is, they do not depend on the value of the current tree. The formula $\hat{E}_{\mathcal{P}} \doteq q$, meanwhile, is not pure, since it relies on the tree being large enough to execute the query. However, it is upward-closed (Defn. 3.16a): the locality conditions mean that querying a larger tree always returns the same result. This property, expressed by the equation $K \cdot (P \wedge (\hat{E}_{\mathcal{P}} \doteq q)) \Rightarrow (\hat{E}_{\mathcal{P}} \doteq q) \wedge K \cdot P$, is useful for simplifying formulæ.

8.6 Inductive Predicates

The update commands in XTU typically act at an arbitrary set of locations. Context Logic, however, can only be used to reason about a single location at a time. One way of addressing this, considered here, is to introduce a limited form of recursion into the logic, in the shape of inductive fold-like predicates that recurse over node sets. This turns out to be sufficient, and simpler than incorporating full recursion into the logic [Hut99]. Other potential solutions, discussed in Chapter 9, include trying to extend Context Logic to handle contexts with multiple holes, though this does not immediately solve the problem, since some of the update locations might be nested underneath others, and hence not be separable from them using contexts.

Definition 8.22 (Fold predicate)**.** The inductive predicate *fold* is defined below. It takes three arguments: a function P_f of type $\mathcal{N} \times \mathcal{P} \to \mathcal{P}$, a base case assertion P_0,

and a pointer expression $\hat{E}_{\mathcal{P}}$ denoting the set of locations over which the fold acts.

$$\begin{aligned}\mathrm{fold}(P_f, P_0, \hat{E}_{\mathcal{P}}) \triangleq\ & ((\hat{E}_{\mathcal{P}} = \emptyset) \wedge P_0) \vee \\ & (\exists n, p.\ (\hat{E}_{\mathcal{P}} = p \uplus \{n\}) \wedge P_f(n, \mathrm{fold}(P_f, P_0, p)))\end{aligned}$$

The fold takes the locations from $\hat{E}_{\mathcal{P}}$, one by one and in any order, and uses them to expand P_f recursively, with P_0 serving as the base case for when $\hat{E}_{\mathcal{P}}$ is empty. The definition is well-founded since the value of $\hat{E}_{\mathcal{P}}$ is always finite. When working with *fold*, it is often useful to use the following shorthand notation:

$$\mathrm{fold}_{n,Q}(P_f, P_0, \hat{E}_{\mathcal{P}}) \triangleq \mathrm{fold}(\lambda n, Q.\, P_f, P_0, \hat{E}_{\mathcal{P}})$$

where Q is a metavariable denoting a formula.

Example 8.23 (Fold)**.** Below are a few example uses of *fold*.

(a) $\mathrm{fold}_{n,Q}(n[Q], 0, p)$ — describes a tower of nodes $\mathsf{n}_{\mathsf{1a_1{:}p_1}}[\mathsf{n}_{\mathsf{2a_2{:}p_2}}[\ldots[0]]]$, with the node identifiers $\mathsf{n_i}$ taken from p. This can also be expressed without using fold: $\forall n.\, (n \in p \Leftrightarrow \Diamond n[\mathrm{true}]) \wedge \neg\Diamond(\neg 0 \mid \neg 0)$.

(b) $\mathrm{fold}_{n,Q}(((0 \rhd Q) \cdot n[0]), 0, p)$ — describes any tree with the same nodes as p. This too can be written more simply: $\forall n.\, (n \in p \Leftrightarrow \Diamond n[\mathrm{true}])$.

(c) $\mathrm{fold}_{n,Q}(((n[\mathrm{true}] \blacktriangleright Q) \cdot 0), P, p)$ — satisfied by a tree if it is possible to add to it subtrees with root nodes n, for each n in p, and obtain a tree satisfying the base assertion P. In other words, starting from a tree satisfying P, the formula describes the result of disposing the subtrees at the nodes in p. This concisely captures the behaviour of the dispose-tree command, and is indeed used as part of its specification. Unlike the other two examples, this cannot be expressed without using fold.

To describe the behaviour of other commands such as lookup and new, it is useful to introduce a more expressive fold predicate that can explicitly compute tree, label and pointer set values as the recursion unfolds. This second predicate, defined below, is a bit more subtle than the one above.

Definition 8.24 (Foldval)**.** The inductive predicate *foldval*, which computes values of type τ (where τ denotes either trees, labels or pointer sets), takes four arguments—a function $P_f : \tau \times \mathcal{N} \times (\tau \to \mathcal{P}) \to \mathcal{P}$, a base case assertion $P_0 : \tau \to \mathcal{P}$, a starting

value $E_0 : \tau$, and a pointer expression $\hat{E}_{\mathcal{P}}$:

$$\begin{aligned}\text{foldval}(P_f, P_0, E_0, \hat{E}_{\mathcal{P}}) \triangleq ((\hat{E}_{\mathcal{P}} = \emptyset) \wedge P_0(E_0)) \vee \\ (\exists n, p.\, (\hat{E}_{\mathcal{P}} = p \uplus \{n\}) \wedge P_f(E_0, n, \lambda x.\, \text{foldval}(P_f, P_0, x, p))\end{aligned}$$

Like fold, this uses the locations from $\hat{E}_{\mathcal{P}}$ to expand P_f recursively, but additionally passes values between calls, with a starting value E_0. As for fold, it is useful to define the following shorthand notation:

$$\text{foldval}_{x,n,Q}(P_f, P_0, E_0, \hat{E}_{\mathcal{P}}) \triangleq \text{foldval}(\lambda x, n, Q.P_f, \lambda x.P_0, E_0, \hat{E}_{\mathcal{P}})$$

where Q is a metavariable denoting a formula, and x is a variable of type τ.

Example 8.25 (Foldval)**.** The following is an example use of *foldval*:

$$\text{foldval}_{p_1,n,Q}(\exists a, p, p_0.\Diamond n_{a:p}[\text{true}] \wedge (p_1 = p \cup p_0) \wedge Q(p_0), p_1 = \emptyset, p', p)$$

This states that the variable p' contains the union of all the pointer sets at the locations in p, with the base value of $\emptyset$ for when p is empty. This cleanly captures the behaviour of the get-links command.

8.7 Program Logic

The Program Logic framework considered in this chapter is a simple adaptation of the one introduced in Chapter 6. Adapted for the new tree data structure, it uses CL_{xtree} rather than CL_{tree} as the underlying assertion and specification language. The interpretation of Hoare Triples (Defn. 6.8), the Frame Rule, and the other standard Hoare inference rules (Defn. 6.9) are all adapted in the obvious way. The inference rule adaptation is given in Figure 8.5. In the case of the two auxiliary variable rules, this involves extending the rules to all types of variable, including the node variables in the environment. The sets of modified and dependent variables are also treated as before: again it is possible to use the left-hand side of an assignment as its modified variable set, and all the free names in a command as its dependent variable set. Note that the environment variables are never modified nor free in any command.

The one remaining issue, then, involves the axiomatisation of the command language. It is here that one encounters a key difference between the current framework and the previous, simpler one: specifically, in how to provide local specifications for commands that act at more than one location in the tree.

$$\text{Consequence:}\quad \frac{P' \Rightarrow P \quad \{P\}\ \mathbb{C}\ \{Q\} \quad Q \Rightarrow Q'}{\{P'\}\ \mathbb{C}\ \{Q'\}}$$

$$\text{Auxiliary Variable Elimination:}\quad \frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{\exists \bullet.\, P\}\ \mathbb{C}\ \{\exists \bullet.\, Q\}}\ \bullet \notin \mathrm{dep}(\mathbb{C})$$

$$\text{Auxiliary Variable Renaming:}\quad \frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{P[\circ/\bullet]\}\ \mathbb{C}\ \{Q[\circ/\bullet]\}}\ \bullet, \circ \notin \mathrm{dep}(\mathbb{C}) \wedge \circ \notin \mathrm{free}(P, Q)$$

$$\text{Frame Rule:}\quad \frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{K \cdot P\}\ \mathbb{C}\ \{K \cdot Q\}}\ \mathrm{mod}(\mathbb{C}) \cap \mathrm{free}(K) = \emptyset$$

where $\bullet$ and $\circ$ are any variables of the same type

Figure 8.5: XTU Inference Rules

The problem is best illustrated using an example. Consider the move-tree p_1 to-subforest p_2 command. This acts at two subtrees: one at the source location and one at the destination. These locations can either be separate or nested, though the destination must not be contained within the source. One approach is to define the strict footprint of the command to include just these two (possibly overlapping) trees. Expressing two separate trees, however, is not possible in the current Context Logic framework, and the possibility of overlap means that it would not be enough simply to move to multi-holed contexts. An alternative approach is to weaken the notion of footprint to the smallest tree that contains both the source and destination. Describing this weaker footprint, however, still requires a case-by-case analysis of the relative shape of the two subtrees: whether they are nested or disjoint, and which one is at the top-level. These three possibilities can be expressed by the following three axioms:

$$\begin{aligned}
\{n_{1a_1:p_1}[(0 \rhd x) \cdot n_{2a_2:p_2}[y]]\}\ &\mathbb{C}\ \{n_{1a_1:p_1}[x \mid n_{2a_2:p_2}[y]]\} \\
\{n_{1a_1:p_1}[x] \mid ((0 \rhd z) \cdot n_{2a_2:p_2}[y])\}\ &\mathbb{C}\ \{n_{1a_1:p_1}[x \mid n_{2a_2:p_2}[y]] \mid z\} \\
\{(\Diamond n_{1a_1:p_1}[x] \wedge z) \mid n_{2a_2:p_2}[y]\}\ &\mathbb{C}\ \{(n_{1a_1:p_1}[x] \rhd z) \cdot n_{1a_1:p_1}[x \mid n_{2a_2:p_2}[y]]\}
\end{aligned}$$

These can then be combined into a single large axiom using a labelled disjunction and auxiliary variables. This approach, however, is clearly clumsy and cannot necessarily

be applied when trying to describe any of the other update commands, all of which act at an arbitrary number of locations.

This leads to a third alternative, which is to simply drop the idea of local specification, and instead specify commands by describing their action on arbitrary trees satisfying the necessary safety preconditions, giving the strongest possible postconditions at the end. The important observation here is that abandoning local specification has no effect on the locality of the commands, or the possibility of local reasoning: a non-local specification can still be extended using the Frame Rule to describe a larger tree. This fracture between local reasoning and local specification is novel. It is not evident in Separation Logic since the flat, set-like nature of the heap means that multiple locations can always be separated together using the $*$ connective.

Thus, the 'move-tree p_1 to-subforest p_2' command is specified on any tree, represented by a variable x, that satisfies the appropriate safety preconditions: namely $p_1{=}\{n_1\}$ and $p_2{=}\{n_2\}$, which state that p_1 and p_2 consist of single locations, and $(0 \rhd \Diamond n_{2a_2:p'_2}[\text{true}]) \cdot n_{1a_1:p'_1}[\text{true}]$, which states that both the source and destination locations are present and that the latter is not nested inside the former. The postcondition, meanwhile, describes the resulting tree after the move. This clearly includes the subtree $n_{2a_2:p'_2}[y \,|\, n_{1a_1:p'_1}[z]]$, for appropriate values of y and z. The context of this subtree, as well as the values of y and z, are described indirectly using the $\blacktriangleright$ adjoint: the required context is the only one to which it is possible to add back the subtrees $n_{1a_1:p'_1}[z]$ and $n_{2a_2:p'_2}[y]$ and recover the original tree x. Hence the final specification states:

$$\{x \wedge (p_1{=}\{n_1\}) \wedge (p_2{=}\{n_2\}) \wedge (0 \rhd \Diamond n_{2a_2:p'_2}[\text{true}]) \cdot n_{1a_1:p'_1}[\text{true}]\}$$
$$\text{move-tree } p_1 \text{ to-subforest } p_2$$
$$\{(n_{2a_2:p'_2}[y] \blacktriangleright ((n_{1a_1:p'_1}[z] \blacktriangleright x) \cdot 0)) \cdot n_{2a_2:p'_2}[y \mid n_{1a_1:p'_1}[z]]\}$$

In fact, it is possible to restrict this non-local specification to one describing the smallest subtree containing both the source and destination, by adding the side-condition $\neg(\neg I \cdot (\Diamond n_1 \wedge \Diamond n_2))$. However, this merely complicates the specification without introducing any obvious benefits.

The resulting specification, while not local, is succinct and manageable. The axioms of the other update commands all follow the same principle.

Definition 8.26 (XTU Axioms)**.** The axioms for the atomic update commands are given in Figure 8.6. The inference rules for the control structures are the standard Hoare ones below, with booleans B embedded into the logic in the obvious way:

$$\frac{\{P\}\ \mathbb{C}_1\ \{Q\} \quad \{Q\}\ \mathbb{C}_2\ \{R\}}{\{P\}\ \mathbb{C}_1\ ;\ \mathbb{C}_2\ \{R\}} \qquad \frac{\{B \wedge P\}\ \mathbb{C}_1\ \{Q\} \quad \{\neg B \wedge P\}\ \mathbb{C}_2\ \{Q\}}{\{P\}\ \text{if } B \text{ then } \mathbb{C}_1 \text{ else } \mathbb{C}_2\ \{Q\}}$$

$$\frac{\{B \wedge P\}\ \mathbb{C}\ \{P\}}{\{P\}\ \text{while } B \text{ do } \mathbb{C}\ \{\neg B \wedge P\}}$$

The axioms in Figure 8.6, while more complicated than those of BTU, still exhibit a high level of order and inter-similarity. The axioms for normal assignment are the standard small ones, defined on the empty tree 0. Query evaluation, in contrast, requires a non-local specification, since its successful execution depends on the presence of a large enough tree to run the query. As in the move axiom, the precondition describes an arbitrary tree x together with a safety condition, $p' \doteq q$, guaranteeing that the query executes successfully. The postcondition, meanwhile, simply updates the value of p, leaving the tree unchanged.

The remaining axioms all use the inductive fold predicates to reason about updates at multiple locations, but otherwise follow the same patterns as move and query. The preconditions are all identical: a variable x symbolises an arbitrary tree, while $\diamond p$ provides the necessary safety condition by stating that all the update locations are present. The postconditions, meanwhile, describe the effect of executing the appropriate command on x. In the case of update commands such as append and dispose, this is done using the *fold* predicate, which specifies each update as the step-by-step replacement of subtrees $n_{a:p}[y]$ by new trees appropriate to the command. For example, the axiom for dispose-trees, previously explained in Example 8.23, replaces these subtrees by 0. For the lookup commands, meanwhile, the original tree x is unchanged, while the lookup process is described using the *foldval* predicate, which specifies the step-by-step construction of a lookup value from the subtrees $n_{a:p}[y]$. Hence, the axiom for get-links, discussed in Example 8.25, describes the unioning of the pointer values p, with a base case value of $\emptyset$. Finally, the axioms for the new commands, which also use *foldval*, combine the approaches of update and lookup, to describe the addition of fresh nodes $m_{\star:\emptyset}[0]$ to the tree, while at the same time collecting the node identifiers m in a variable p'.

Assign

$\{0 \wedge (x' = E_{\mathcal{T}})\}$	$x := E_{\mathcal{T}}$	$\{0 \wedge (x = x')\}$
$\{0 \wedge (a' = E_{\mathcal{A}})\}$	$a := E_{\mathcal{A}}$	$\{0 \wedge (a = a')\}$
$\{0 \wedge (p' = E_{\mathcal{P}})\}$	$p := E_{\mathcal{P}}$	$\{0 \wedge (p = p')\}$
$\{x \wedge (p' \doteq q)\}$	$p := q$	$\{x \wedge (p = p')\}$

Update

$\{x \wedge \Diamond p\}$	dispose-trees at p	$\{\mathrm{fold}_{n,Q}((\exists a, p, y.\, (n_{a:p}[y] \blacktriangleright Q) \cdot 0), x, p)\}$
$\{x \wedge \Diamond p\}$	dispose-subforests at p	$\{\mathrm{fold}_{n,Q}((\exists a, p, y.\, (n_{a:p}[y] \blacktriangleright Q) \cdot n_{a:p}[0]), x, p)\}$
$\{x \wedge \Diamond p\}$	dispose-links at p	$\{\mathrm{fold}_{n,Q}((\exists a, p, y.\, (n_{a:p}[y] \blacktriangleright Q) \cdot n_{a:\emptyset}[y]), x, p)\}$
$\{x \wedge \Diamond p\}$	append-trees $E_{\mathcal{T}}$ at p	$\{\mathrm{fold}_{n,Q}((\exists a, p, y.\, (n_{a:p}[y] \blacktriangleright Q) \cdot (n_{a:p}[y \mid [\![E_{\mathcal{T}}]\!]])), x, p)\}$
$\{x \wedge \Diamond p\}$	append-subforests $E_{\mathcal{T}}$ at p	$\{\mathrm{fold}_{n,Q}((\exists a, p, y.\, (n_{a:p}[y] \blacktriangleright Q) \cdot n_{a:p}[y \mid [\![E_{\mathcal{T}}]\!]]), x, p)\}$
$\{x \wedge \Diamond p\}$	append-links $E_{\mathcal{P}}$ at p	$\{\mathrm{fold}_{n,Q}((\exists a, p, y.\, (n_{a:p}[y] \blacktriangleright Q) \cdot n_{a:p \cup E_{\mathcal{P}}}[y]), x, p)\}$
$\{x \wedge \Diamond p\}$	set-labels $E_{\mathcal{A}}$ at p	$\{\mathrm{fold}_{n,Q}((\exists a, p, y.\, (n_{a:p}[y] \blacktriangleright Q) \cdot n_{E_{\mathcal{A}}:p}[y]), x, p)\}$

Figure 8.6: XTU Command Axioms (1/2)

Lookup

$\{x \wedge \Diamond p\}$ $y :=$ get-trees at p $\{x \wedge \text{foldval}_{y_1,n,Q}(\exists a, p, y, y_0.\Diamond n_{a:p}[y] \wedge (y_1 \in [\![n_{a:p}[y]]\!] \mid y_0) \wedge Q(y_0), y_1 = 0, y, p)\}$

$\{x \wedge \Diamond p\}$ $y :=$ get-subforests at p $\{x \wedge \text{foldval}_{y_1,n,Q}(\exists a, p, y, y_0.\Diamond n_{a:p}[y] \wedge (y_1 \in [\![y]\!] | y_0) \wedge Q(y_0), y_1 = 0, y, p)\}$

$\{x \wedge \Diamond p\}$ $p' :=$ get-links at p $\{x \wedge \text{foldval}_{p_1,n,Q}(\exists a, p, y, p_0.\Diamond n_{a:p}[y] \wedge (p_1 = p \cup p_0) \wedge Q(p_0), p_1 = \emptyset, p', p)\}$

$\{x \wedge \Diamond p\}$ $a :=$ get-labels at p $\{x \wedge \text{foldval}_{a_1,n,Q}(\exists a, p, y, a_0.\Diamond n_{a:p}[y] \wedge (a_1 = a) \wedge Q(a_0), a_1 = \star, a, p)\}$

New

$\{x \wedge \Diamond p\}$ $p' :=$ new-trees at p $\{\text{foldval}_{p_1,n,Q}(\exists a, p, y, m, p_0.(p_1 = p_0 \uplus m) \wedge ((n_{a:p}[y] \blacktriangleright Q(p_0)) \cdot (n_{a:p}[y] \mid m_{\star:\emptyset}[0])), (p_1 = \emptyset) \wedge x, p', p)\}$

$\{x \wedge \Diamond p\}$ $p' :=$ new-subforests at p $\{\text{foldval}_{p_1,n,Q}(\exists a, p, y, m, p_0.(p_1 = p_0 \uplus m) \wedge ((n_{a:p}[y] \blacktriangleright Q(p_0)) \cdot (n_{a:p}[y \mid m_{\star:\emptyset}[0]])), (p_1 = \emptyset) \wedge x, p', p)\}$

Move* *only $\mathbb{C} =$ move-tree p_1 to-subforest p_2 shown; other move axioms similar

$$\{x \wedge (p_1 = \{n_1\}) \wedge (p_2 = \{n_2\}) \wedge ((0 \rhd \Diamond n_{2_{a_2:p_2'}}[\text{true}]) \cdot n_{1_{a_1:p_1'}}[\text{true}])\} \;\; \mathbb{C} \;\; \{(n_{2_{a_2:p_2'}}[y] \blacktriangleright ((n_{1_{a_1:p_1'}}[z] \blacktriangleright x) \cdot 0)) \cdot (n_{2_{a_2:p_2'}}[y \mid n_{1_{a_1:p_1'}}[z]])\}$$

Figure 8.6: XTU Command Axioms (2/2)

8.8 Weakest Preconditions

As in Chapter 6, a useful test of the logic involves giving and deriving the weakest precondition axioms of the various commands.

Lemma 8.27 (XTU Weakest Preconditions)**.** *The weakest precondition axioms for the atomic update commands are given in Figure 8.7. The weakest preconditions for the control structures are the standard Hoare ones: namely*

$$\begin{array}{ll} \mathrm{wp}(\mathbb{C}_1, \mathrm{wp}(\mathbb{C}_2, P)) & \textit{for } \mathbb{C}_1 \,;\, \mathbb{C}_2 \\ (B \wedge \mathrm{wp}(\mathbb{C}_1, P)) \vee (\neg B \wedge \mathrm{wp}(\mathbb{C}_2, P)) & \textit{for if } B \textit{ then } \mathbb{C}_1 \textit{ else } \mathbb{C}_2 \end{array}$$

and the weakest P_w satisfying $(\neg B \wedge P) \vee (B \wedge \mathrm{wp}(\mathbb{C}, P_w))$ for while B do $\mathbb{C}$.

The weakest preconditions of the assignment commands, including query evaluation, are straightforward, corresponding to the standard Hoare assignment axiom. The weakest preconditions of the other updates, meanwhile, exhibit a strong 'duality' with the non-local axioms, often resembling the latter 'in reverse'. This is not surprising: whereas the specification axioms start with an arbitrary tree x and describe the strongest postcondition, the weakest precondition axioms start with an arbitrary postcondition P and describe the most general precondition. Hence, the weakest preconditions for update commands acting at a location set p, such as append and dispose, use the fold predicate to specify that *whenever* it is possible to remove the subtrees $n_{a:p'}[y]$ from the current tree, for each $n \in p$, and replace them by the appropriate updated subtrees, *then* the resulting tree must satisfy the postcondition P. Note how this uses the universal adjoint $\triangleright$, as opposed to the existential adjoint $\blacktriangleright$ used in the specification axioms. This is important for commands such as append-trees, since the precondition must take into account all the possible node renamings of the appended tree. Finally, notice how the fold assertions all imply the necessary safety precondition $\Diamond p$.

The weakest preconditions for lookup and new are slightly more complicated. Unlike those for update, these require the explicit safety property $\Diamond p$. Furthermore, they must factor in the effect of variable assignment. Hence, the preconditions for lookup state that the postcondition P must hold for *all* the possible lookup values y' that might be assigned to y. This universal quantification is particularly important in the case of get-trees, since it is necessary to consider all the possible node-renamings of the resulting tree. Similarly, the preconditions for new state that the postcondition P

Assign

$\{P[E_{\mathcal{T}}/x]\}$	$x := E_{\mathcal{T}}$	$\{P\}$
$\{P[E_{\mathcal{A}}/a]\}$	$a := E_{\mathcal{A}}$	$\{P\}$
$\{P[E_{\mathcal{P}}/p]\}$	$p := E_{\mathcal{P}}$	$\{P\}$
$\{\exists p'.(p' \doteq q) \wedge P[p'/p]\}$	$p := q$	$\{P\}$

Update

$\{\mathrm{fold}_{n,Q}((\exists a, p, y.\, (0 \rhd Q) \cdot n_{a:p}[y]), P, p)\}$	dispose-trees at p	$\{P\}$
$\{\mathrm{fold}_{n,Q}((\exists a, p, y.\, (n_{a:p}[0] \rhd Q) \cdot n_{a:p}[y]), P, p)\}$	dispose-subforests at p	$\{P\}$
$\{\mathrm{fold}_{n,Q}((\exists a, p, y.\, (n_{a:\emptyset}[y] \rhd Q) \cdot n_{a:p}[y]), P, p)\}$	dispose-links at p	$\{P\}$
$\{\mathrm{fold}_{n,Q}((\exists a, p, y.\, ((n_{a:p}[y] \mid [\![E_{\mathcal{T}}]\!]) \rhd Q) \cdot n_{a:p}[y]), P, p)\}$	append-trees $E_{\mathcal{T}}$ at p	$\{P\}$
$\{\mathrm{fold}_{n,Q}((\exists a, p, y.\, (n_{a:p}[y \mid [\![E_{\mathcal{T}}]\!]] \rhd Q) \cdot n_{a:p}[y]), P, p)\}$	append-subforests $E_{\mathcal{T}}$ at p	$\{P\}$
$\{\mathrm{fold}_{n,Q}((\exists a, p, y.\, (n_{a:p \cup E_{\mathcal{P}}}[y] \rhd Q) \cdot n_{a:p}[y]), P, p)\}$	append-links $E_{\mathcal{P}}$ at p	$\{P\}$
$\{\mathrm{fold}_{n,Q}((\exists a, p, y.\, (n_{E_{\mathcal{A}}:p}[y] \rhd Q) \cdot n_{a:p}[y]), P, p)\}$	set-labels $E_{\mathcal{A}}$ at p	$\{P\}$

Figure 8.7: XTU Weakest Preconditions (1/2)

Lookup

$$\{\Diamond p \wedge \forall y'.(\text{foldval}_{y_1,n,Q}(\exists a,p,y,y_0.\Diamond n_{a:p}[y] \wedge (y_1 \in [\![n_{a:p}[y]]\!]|y_0) \wedge Q(y_0), y_1 = 0, y', p) \Rightarrow P[y'/y])\} \quad y := \text{get-trees at } p \quad \{P\}$$

$$\{\Diamond p \wedge \forall y'.(\text{foldval}_{y_1,n,Q}(\exists a,p,y,y_0.\Diamond n_{a:p}[y] \wedge (y_1 \in [\![y]\!]|y_0) \wedge Q(y_0), y_1 = 0, y', p) \Rightarrow P[y'/y])\} \quad y := \text{get-subforests at } p \quad \{P\}$$

$$\{\Diamond p \wedge \forall p''.(\text{foldval}_{p_1,n,Q}(\exists a,p,y,p_0.\Diamond n_{a:p}[y] \wedge (p_1 = p \cup p_0) \wedge Q(p_0), p_1 = \emptyset, p'', p) \Rightarrow P[p''/p'])\} \quad p' := \text{get-links at } p \quad \{P\}$$

$$\{\Diamond p \wedge \forall a'.(\text{foldval}_{a_1,n,Q}(\exists a,p,y,a_0.\Diamond n_{a:p}[y] \wedge (a_1 = a) \wedge Q(a_0), a_1 = \star, a', p) \Rightarrow P[a'/a])\} \quad a := \text{get-labels at } p \quad \{P\}$$

New

$$\left\{\Diamond p \wedge \forall p''.\left(|p''| = |p| \Rightarrow \text{foldval}_{p_1,n,Q}\left(\begin{array}{c}\exists a,p,y,m,p_0.(p_1 = p_0 \uplus m) \wedge ((n_{a:p}[y] \mid m_{\star:\emptyset}[0]) \rhd \\ Q(p_0))\cdot n_{a:p}[y]), (p_1 = \emptyset) \wedge P[p''/p'], p'', p\end{array}\right)\right)\right\} \quad p' := \text{new-trees at } p \quad \{P\}$$

$$\left\{\Diamond p \wedge \forall p''.\left(|p''| = |p| \Rightarrow \text{foldval}_{p_1,n,Q}\left(\begin{array}{c}\exists a,p,y,m,p_0.(p_1 = p_0 \uplus m) \wedge (n_{a:p}[y \mid m_{\star:\emptyset}[0]] \rhd \\ Q(p_0))\cdot n_{a:p}[y]), (p_1 = \emptyset) \wedge P[p''/p'], p'', p\end{array}\right)\right)\right\} \quad p' := \text{new-subforests at } p \quad \{P\}$$

*only $\mathbb{C}$ = move-tree p_1 to-subforest p_2 shown; other move preconditions similar **Move***

$$\{\exists n_1, n_2.(p_1 = \{n_1\}) \wedge (p_2 = \{n_2\}) \wedge \exists a_1, p_1', y, a_2, p_2', z.\,(0 \rhd ((n_{2_{a_2:p_2'}}[y \mid n_{1_{a_1:p_1'}}[z]] \rhd P) \cdot n_{2_{a_2:p_2'}}[y])) \cdot n_{1_{a_1:p_1'}}[z]\} \;\; \mathbb{C} \;\; \{P\}$$

Figure 8.7: XTU Weakest Preconditions (2/2)

must hold for any fresh set of node identifiers of the appropriate size which is chosen for the variable p'.

Finally, the weakest precondition for move, following the same principle, states that the source location n_1 and destination location n_2 are in the tree, with the latter not contained inside the former, and that whenever the tree at n_1 is moved to n_2, the resulting tree satisfies P.

As in Chapter 6, the derivations of the weakest precondition axioms from the individual command axioms are highly regular, and consist of applying the right context assertion using the Frame Rule, and then simplifying and eliminating auxiliary variables. For the non-local axioms, the right context assertion is simply the one which equates the initial tree x with the tree described in the weakest precondition, without adding any extra tree structure: that is $(I \wedge (x \rhd P_w))$, where P_w is the part of the weakest precondition describing the tree. In each case, the weakest precondition axiom follows directly by the Rule of Consequence.

Lemma 8.28 (Derivability of Weakest Preconditions)**.** *The weakest preconditions in Figure 8.7 are derivable from the command axioms in Figure 8.6.*

Proof. A generic derivation is given for each command type in Figure 8.9 at the end of the chapter. These derivations, written in the same style as the derivations in Lemma 6.18, use z to represent variables of arbitrary type, and $P_{\triangleright}$, $P_{\blacktriangleright}$, P_f and P_0 as notational shorthand for invariant parts of the appropriate axiom or weakest precondition. Some uses of the Rule of Consequence, which are outside the scope of the program logic and are not justified formally here, rely on induction on the fold structure, as well as some of the general properties of formulæ explored previously in the thesis.

8.9 Reasoning Example

Like in Chapter 6, this chapter's exposition concludes with an example of program reasoning. The following examples both reason about the collapse-all-links program given in Example 8.14. The first uses backward-reasoning to derive the weakest precondition of the program, while the second uses forward reasoning to provide a specification for the program in the style of the command axioms.

Backward Reasoning	**Forward Reasoning**
$\{P\}$	$\{\exists p''.\,(p'' \doteq p/\text{link} :: *) \wedge z\}$
dispose-links at p	$p' := p/\text{link} :: *$
$\{\text{dispose}_{\text{L}}(p \rhd P)\}$	$\{(p' \doteq p/\text{link} :: *) \wedge z\}$
append-subforests x at p	$x :=$ get-trees at p'
$\{\text{append}_{\text{SF}}(x, p \rhd \text{dispose}_{\text{L}}(p \rhd P))\}$	$\left\{\begin{array}{l}(p' \doteq p/\text{link} :: *) \wedge (x = \text{lookup}_{\text{T}}(p')) \\ \wedge\, z\end{array}\right\}$
$x :=$ get-trees at p'	append-subforests x at p
$\left\{\begin{array}{l}\Diamond p' \wedge \forall y.(y = \text{lookup}_{\text{T}}(p')) \Rightarrow \\ \text{append}_{\text{SF}}(y, p \rhd \text{dispose}_{\text{L}}(p \rhd P))\end{array}\right\}$	$\left\{\begin{array}{l}(p' \doteq p/\text{link} :: *) \wedge (x = \text{lookup}_{\text{T}}(p')) \\ \wedge\, \text{append}_{\text{SF}}(x, p \blacktriangleright z)\end{array}\right\}$
$p' := p/\text{link} :: *$	dispose-links at p
$\left\{\begin{array}{l}\exists p''.\,(p'' \doteq p/\text{link} :: *) \wedge \\ \Diamond p'' \wedge \forall y.(y = \text{lookup}_{\text{T}}(p'')) \Rightarrow \\ \text{append}_{\text{SF}}(y, p \rhd \text{dispose}_{\text{L}}(p \rhd P))\end{array}\right\}$	$\left\{\begin{array}{l}(x = \text{lookup}_{\text{T}}(p')) \\ \wedge\, \text{dispose}_{\text{L}}(p \blacktriangleright \text{append}_{\text{SF}}(x, p \blacktriangleright z))\end{array}\right\}$

Figure 8.8: XTU Program Reasoning Example

Example 8.29 (Weakest precondition)**.** The calculation of the weakest precondition of the collapse-all-links program of Example 8.14 is given in the left column of Figure 8.8. For clarity, the derivation uses the following syntactic sugar for the fold predicates used in the weakest preconditions of the constituent program commands:

$$x = \text{lookup}_{\text{T}}(p) \triangleq \text{foldval}_{y_1,n,Q}\left(\begin{array}{l}\exists a, p, y, y_0.\Diamond n_{a:p}[y] \wedge Q(y_0) \wedge \\ \quad (y_1 \in [\![n_{a:p}[yy]]\!] \mid y_0), y_1 = 0, x, p\end{array}\right)$$

$$\text{append}_{\text{SF}}(E_{\mathcal{T}}, p \rhd P) \triangleq \text{fold}_{n,Q}((\exists a, p, y.\,(n_{a:p}[y \mid [\![E_{\mathcal{T}}]\!]] \rhd Q) \cdot n_{a:p}[y]), P, p)$$

$$\text{dispose}_{\text{L}}(p \rhd P) \triangleq \text{fold}_{n,Q}((\exists a, p, y.\,(n_{a:\emptyset}[y] \rhd Q) \cdot n_{a:p}[y]), P, p)$$

The weakest precondition of the program then follows immediately by applying the weakest precondition axioms of the individual commands sequentially, starting at the end of the program. The last condition simplifies further to:

$$\exists p'', y.\,(p'' \doteq p/\text{link} :: *) \wedge (y = \text{lookup}_{\text{T}}(p'')) \wedge \text{append}_{\text{SF}}(y, p \rhd \text{dispose}_{\text{L}}(p \rhd P))$$

since appending trees that are equivalent modulo renaming gives the same results. This provides a natural statement of the weakest precondition, which in turn includes

the safety precondition $\exists p''.(p'' \doteq p/\mathrm{link} :: *)$ stating that all the nodes in p, as well as any links from them, must be in the tree.

Example 8.30 (Specification)**.** The derivation of a specification for the collapse-all-links program is given in the right column of Figure 8.8. Like in Example 8.29, this uses syntactic sugar to represent the fold predicates in the postconditions of the specification axioms of the constituent commands:

$$\mathrm{append}_{\mathrm{SF}}(E_{\mathcal{T}}, p \blacktriangleright P) \triangleq \mathrm{fold}_{n,Q}((\exists a, p, y.\, (n_{a:p}[y] \blacktriangleright Q) \cdot n_{a:p}[y \mid [\![E_{\mathcal{T}}]\!]]), P, p)$$
$$\mathrm{dispose}_{\mathrm{L}}(p \blacktriangleright P) \triangleq \mathrm{fold}_{n,Q}((\exists a, p, y.\, (n_{a:p}[y] \blacktriangleright Q) \cdot n_{a:\emptyset}[y]), P, p)$$

with $x = \mathrm{lookup}_{\mathrm{T}}(p)$ defined as in Example 8.29. Figure 8.8 shows the result of reasoning forwards from the command axioms, applying the Frame Rule to derive the necessary safety preconditions, and using the Rule of Consequence to simplify the postconditions at each step. This results in a concise specification of the program, in exactly the same style as the individual command axioms.

8.10 Assessment

This chapter demonstrates how to reason about a complex update language using Context Logic. As such, it is an important first step towards reasoning about update to real semistructured data. Additionally, it raises many interesting observations along the way, both about language design (local queries, atomic actions at multiple locations, incorporating pointers) and program reasoning (the use of fold predicates, non-local specifications). Nevertheless, the solution presented here is not completely satisfactory: the resulting reasoning is very systematic but still rather complicated. A long-term assessment of the various choices made, and a look at whether the reasoning can be further simplified, is therefore necessary. Furthermore, the work raises a deeper question, regarding the rôle (in particular, the necessity and/or possibility) of local specifications in a local reasoning framework, that also need to be addressed.

The most obvious thing to do at this point would be to take a few steps back and return to a simpler setting for a while. One option would be to try first to solve the problem of locally specifying the move command: this involves update at multiple locations, but not at arbitrary sets of locations as for queries, and can be expressed in a simple language (such as the one given in Chapter 6). Another option is to explore other approaches to complex tree update language design. For example, one could

try to reason about a DOM-like language, which traverses ordered lists of locations and manipulates them one at a time. These options, as well as other possibilities for future work, are discussed in the following chapter.

Assignments of an expression E to a variable z

$$\dfrac{\dfrac{\dfrac{\{0 \wedge (z' = E)\}\ z := E\ \{0 \wedge (z = z')\}}{\{(0 \rhd P[z'/z]) \cdot (0 \wedge (z' = E))\}\ z := E\ \{(0 \rhd P[z'/z]) \cdot (0 \wedge (0 \wedge (z = z'))\}}\ \textsc{Cons}}{\{(z' = E) \wedge P[z'/z]\}\ z := E\ \{(z = z') \wedge P[z'/z]\}}\ \textsc{Cons/Elim}}{\{P[E/z]\}\ z := E\ \{P\}}\ \textsc{Frame}$$

Query evaluation

$$\dfrac{\dfrac{\dfrac{\{x \wedge (p' \doteq q)\}\ p := q\ \{x \wedge (p = p')\}}{\{(I \wedge (x \rhd P[p'/p])) \cdot (x \wedge (p' \doteq q)\}\ p := q\ \{(I \wedge (x \rhd P[p'/p])) \cdot (x \wedge (p = p'))\}}\ \textsc{Cons}}{\{x \wedge (p' \doteq q) \wedge P[p'/p]\}\ p := q\ \{(p = p') \wedge P[p'/p]\}}\ \textsc{Cons/Elim}}{\{\exists p'.\, (p' \doteq q) \wedge P[p'/p]\}\ p := q\ \{P\}}\ \textsc{Frame}$$

Updates at locations p

$$\dfrac{\dfrac{\dfrac{\{x \wedge \Diamond p\}\ \mathbb{C}(p)\ \{\mathrm{fold}(P_{\blacktriangleright}, x, p)\}}{\{(I \wedge (x \rhd \mathrm{fold}(P_{\rhd}, P, p))) \cdot (x \wedge \Diamond p)\}\ \mathbb{C}(p)\ \{(I \wedge (x \rhd \mathrm{fold}(P_{\rhd}, P, p))) \cdot (\mathrm{fold}(P_{\blacktriangleright}, x, p))\}}\ \textsc{Cons}}{\{x \wedge \Diamond p \wedge \mathrm{fold}(P_{\rhd}, P, p)\}\ \mathbb{C}(p)\ \{\mathrm{fold}(P_{\blacktriangleright}, (\mathrm{fold}(P_{\rhd}, P, p)), p)\}}\ \textsc{Cons/Elim}}{\{\mathrm{fold}(P_{\rhd}, P, p)\}\ \mathbb{C}(p)\ \{P\}}\ \textsc{Frame}$$

Figure 8.9: Generic Derivations of the Weakest Preconditions (1/3)

Lookups assigning to variable z values at locations p

$$\dfrac{\dfrac{\dfrac{\{x \wedge \Diamond p\}\ z := \mathbb{C}(p)\ \{x \wedge \mathrm{foldval}(P_f, P_0, z, p)\}}{\left\{\begin{array}{l}(I \wedge (x \rhd \forall y.\,(\mathrm{foldval}(P_f, P_0, y, p) \Rightarrow P[y/z])))\\ \quad \cdot\,(x \wedge \Diamond p)\end{array}\right\}\ z := \mathbb{C}(p)\ \left\{\begin{array}{l}(I \wedge (x \rhd \forall y.\,(\mathrm{foldval}(P_f, P_0, y, p) \Rightarrow P[y/z])))\\ \quad \cdot\,(x \wedge \mathrm{foldval}(P_f, P_0, z, p))\end{array}\right\}}\ \textsc{Cons}}{\{x \wedge \Diamond p \wedge \forall y.\,(\mathrm{foldval}(P_f, P_0, y, p) \Rightarrow P[y/z])\}\ z := \mathbb{C}(p)\ \{\mathrm{foldval}(P_f, P_0, z, p) \wedge \forall y.\,(\mathrm{foldval}(P_f, P_0, y, p) \Rightarrow P[y/z])\}}\ \textsc{Cons/Elim}}{\{\Diamond p \wedge \forall y.\,(\mathrm{foldval}(P_f, P_0, y, p) \Rightarrow P[y/z])\}\ z := \mathbb{C}(p)\ \{P\}}\ \textsc{Frame}$$

New commands assigning to variable p' fresh nodes at locations p

$$\dfrac{\dfrac{\dfrac{\dfrac{\{x \wedge \Diamond p\}\ p' := \mathbb{C}(p)\ \{\mathrm{foldval}_{p_1}(P_{\blacktriangleright}, (p_1 = \emptyset) \wedge x, p', p)\}}{\left\{\begin{array}{l}(I \wedge (x \rhd (\forall p''.(|p''| = |p|) \Rightarrow \mathrm{foldval}_{p_1}(P_{\rhd},\\ \quad (p_1 = \emptyset) \wedge P[p''/p'], p'', p)))\\ \quad \cdot\,(x \wedge \Diamond p))\end{array}\right\}\ p' := \mathbb{C}(p)\ \left\{\begin{array}{l}(I \wedge (x \rhd (\forall p''.(|p''| = |p|) \Rightarrow \mathrm{foldval}_{p_1}(P_{\rhd},\\ \quad (p_1 = \emptyset) \wedge P[p''/p'], p'', p))))\\ \quad \cdot\,(\mathrm{foldval}_{p_1}(P_{\blacktriangleright}, (p_1 = \emptyset) \wedge x, p', p))\end{array}\right\}}\ \textsc{Cons}}{\left\{\begin{array}{l}x \wedge \Diamond p \wedge \forall p''.(|p''| = |p|) \Rightarrow\\ \quad \mathrm{foldval}_{p_1}(P_{\rhd}, (p_1 = \emptyset) \wedge P[p''/p'], p'', p)\end{array}\right\}\ p' := \mathbb{C}(p)\ \left\{\begin{array}{l}\mathrm{foldval}_{p_1}(P_{\blacktriangleright}, (p_1 = \emptyset) \wedge \forall p''.(|p''| = |p|) \Rightarrow\\ \quad \mathrm{foldval}_{p_1}(P_{\rhd}, (p_1 = \emptyset) \wedge P[p''/p'], p'', p), p', p)\end{array}\right\}}\ \textsc{Cons}}{\left\{\begin{array}{l}x \wedge \Diamond p \wedge \forall p''.(|p''| = |p|) \Rightarrow\\ \quad \mathrm{foldval}_{p_1}(P_{\rhd}, (p_1 = \emptyset) \wedge P[p''/p'], p'', p)\end{array}\right\}\ p' := \mathbb{C}(p)\ \left\{\begin{array}{l}\mathrm{foldval}_{p_1}(P_{\blacktriangleright}, (p_1 = \emptyset) \wedge\\ \quad \mathrm{foldval}_{p_1}(P_{\rhd}, (p_1 = \emptyset) \wedge P, p', p), p', p)\end{array}\right\}}\ \textsc{Cons/Elim}}{\{\Diamond p \wedge \forall p''.(|p''| = |p|) \Rightarrow \mathrm{foldval}_{p_1}(P_{\rhd}, (p_1 = \emptyset) \wedge P[p''/p'], p'', p)\}\ p' := \mathbb{C}(p)\ \{P\}}\ \textsc{Frame}$$

Figure 8.9: Generic Derivations of the Weakest Preconditions (2/3)

Move: move-tree p_1 to-subforest p_2 shown, other move derivations similar

$$\left\{\begin{array}{l} x \wedge (p_1{=}\{n_1\}) \wedge (p_2{=}\{n_2\}) \wedge \\ ((0 \rhd \Diamond n_{2_{a_2:p_2'}}[\text{true}]) \cdot n_{1_{a_1:p_1'}}[\text{true}]) \end{array}\right\} \mathbb{C} \left\{(n_{2_{a_2:p_2'}}[y] \blacktriangleright ((n_{1_{a_1:p_1'}}[z] \blacktriangleright x) \cdot 0)) \cdot (n_{2_{a_2:p_2'}}[y \mid n_{1_{a_1:p_1'}}[z]])\right\}$$

FRAME

$$\left\{\begin{array}{l} (I \wedge (x \rhd ((0 \rhd ((n_{2_{a_2:p_2'}}[y \mid n_{1_{a_1:p_1'}}[z]] \rhd P) \cdot n_{2_{a_2:p_2'}}[y])) \\ \cdot n_{1_{a_1:p_1'}}[z]))) \cdot (x \wedge (p_1{=}\{n_1\}) \wedge (p_2{=}\{n_2\}) \wedge \\ \qquad ((0 \rhd \Diamond n_{2_{a_2:p_2'}}[\text{true}]) \cdot n_{1_{a_1:p_1'}}[\text{true}])) \end{array}\right\} \mathbb{C} \left\{\begin{array}{l} (I \wedge (x \rhd ((0 \rhd ((n_{2_{a_2:p_2'}}[y \mid n_{1_{a_1:p_1'}}[z]] \rhd P) \cdot n_{2_{a_2:p_2'}}[y])) \\ \cdot n_{1_{a_1:p_1'}}[z]))) \cdot \\ ((n_{2_{a_2:p_2'}}[y] \blacktriangleright ((n_{1_{a_1:p_1'}}[z] \blacktriangleright x) \cdot 0)) \cdot (n_{2_{a_2:p_2'}}[y \mid n_{1_{a_1:p_1'}}[z]])) \end{array}\right\}$$

CONS

$$\left\{\begin{array}{l} x \wedge (p_1{=}\{n_1\}) \wedge (p_2{=}\{n_2\}) \wedge \\ ((0 \rhd \Diamond n_{2_{a_2:p_2'}}[\text{true}]) \cdot n_{1_{a_1:p_1'}}[\text{true}]) \wedge \\ (0 \rhd ((n_{2_{a_2:p_2'}}[y \mid n_{1_{a_1:p_1'}}[z]] \rhd P) \cdot n_{2_{a_2:p_2'}}[y])) \cdot n_{1_{a_1:p_1'}}[z] \end{array}\right\} \mathbb{C} \left\{\begin{array}{l} (n_{2_{a_2:p_2'}}[y] \blacktriangleright ((n_{1_{a_1:p_1'}}[z] \blacktriangleright \\ ((0 \rhd ((n_{2_{a_2:p_2'}}[y \mid n_{1_{a_1:p_1'}}[z]] \rhd P) \cdot n_{2_{a_2:p_2'}}[y])) \cdot n_{1_{a_1:p_1'}}[z])) \\ \cdot 0)) \cdot (n_{2_{a_2:p_2'}}[y \mid n_{1_{a_1:p_1'}}[z]]) \end{array}\right\}$$

CONS

$$\left\{\begin{array}{l} x \wedge (p_1{=}\{n_1\}) \wedge (p_2{=}\{n_2\}) \wedge \\ (0 \rhd ((n_{2_{a_2:p_2'}}[y \mid n_{1_{a_1:p_1'}}[z]] \rhd P) \cdot n_{2_{a_2:p_2'}}[y])) \cdot n_{1_{a_1:p_1'}}[z] \end{array}\right\} \mathbb{C} \left\{\begin{array}{l} (n_{2_{a_2:p_2'}}[y] \blacktriangleright ((n_{2_{a_2:p_2'}}[y \mid n_{1_{a_1:p_1'}}[z]] \rhd P) \\ \cdot n_{2_{a_2:p_2'}}[y]) \cdot (n_{2_{a_2:p_2'}}[y \mid n_{1_{a_1:p_1'}}[z]])) \end{array}\right\}$$

CONS/ELIM

$$\left\{\begin{array}{l} \exists n_1, n_2.(p_1{=}\{n_1\}) \wedge (p_2{=}\{n_2\}) \wedge \exists a_1, p_1', y, a_2, p_2', z. \\ (0 \rhd ((n_{2_{a_2:p_2'}}[y \mid n_{1_{a_1:p_1'}}[z]] \rhd P) \cdot n_{2_{a_2:p_2'}}[y])) \cdot n_{1_{a_1:p_1'}}[z] \end{array}\right\} \mathbb{C} \{P\}$$

Figure 8.9: Generic Derivations of the Weakest Preconditions (3/3)

Chapter 9

Conclusion

The thesis concludes with a short summary of the main achievements of the work presented here, followed by a discussion of a number of possible directions for future work.

9.1 Achievements

The achievements of the thesis can be broadly split into three main points.

The first consists of the development of a novel spatial logic called Context Logic. The logic is based on the simple idea of context application and hierarchical (as opposed to flat) resource splitting, but is general enough to naturally describe a wide range of inductive data structures. As such, it can be used as an expressive description language for structures such as sequences, trees and terms, in the style of Separation Logic and the Ambient Logic. In the thesis, the underlying theory of Context Logic was explored (Chapter 3), various extensions were investigated (Chapter 4), and a number of example applications were given (Chapter 5).

The second achievement involved presenting a framework for local Hoare reasoning about data update using Context Logic. This framework was directly based on the local reasoning framework of Separation Logic, but extended the key ideas of locality and the Frame Rule to hierarchical data structures. What really set it apart, however, was the expressivity of Context Logic, which enabled a form of reasoning about data update that was not possible using previous spatial logics such as the Ambient Logic. The thesis presented three examples of this type of local reasoning: for tree update (Chapter 6), heap update (Section 7.1) and term rewriting (Section 7.2).

It also raised, in Chapter 8, an interesting point concerning the link between local specifications and local reasoning.

Finally, the third main achievement involved the successful application in Chapter 8 of program reasoning to a more realistic tree update language. The language presented there included pointers, path queries and atomic updates at multiple locations. The work was thus an important first step towards showing the feasibility of reasoning about update to semistructured data.

9.2 Future Work

While the work presented in this thesis forms a fairly self-contained story, it also opens up a wide range of possible future research. Below are a number of interesting avenues that are worth pursuing.

Expressivity and decidability. Two immediate practical questions raised by the thesis concern the expressivity and decidability properties of Context Logic; the former are useful for comparing Context Logic to other logics, the latter for assessing the practicality of automated reasoning tools. Some work has already been done on the expressivity of Context Logic for sequences and for trees, showing that quantifier-free Context Logic for these structures is parametrically more expressive than a simple BI-like logic for sequences, and the Ambient Logic for trees [CGZ07a]. The source of this expressivity lies in the presence of the $\triangleright$ adjoint of context application, and the result provides a strong justification for using Context Logic to reason about tree update. More work on expressivity remains to be done, such as looking for adjunct elimination results for Context Logic of the type that exist for Separation Logic and Ambient Logic. No work, meanwhile, has yet been done on decidability. Using an argument similar to [CYO01], it is possible to show that the Context Logic models for trees and terms with quantification are undecidable. Finding rich decidable fragments and practical decision procedures is vital, however, if one wishes to implement automatic reasoning techniques based on Context Logic similar to the ones already present for Separation Logic.

Multi-holed contexts. Another avenue of research involves looking at possible extensions or restrictions to Context Logic, similar to the ones considered in Chapter 4,

to see how it can better model structured data. While the version of Context Logic chosen for this thesis was the simplest one necessary to reason about tree update, observations like the ones raised in Chapter 8 about footprints suggest that different extensions might be useful for different applications. One obvious extension is to look for natural models of multi-holed contexts that are easy to apply in practice. This seems particularly attractive, as it could simplify the task of reasoning about update at multiple locations, while at the same time reducing the two structures of Context Logic (contexts and data) to one: data can be viewed simply as zero-holed contexts. The main difficulty, however, involves formulating context composition in a clean and usable fashion, since the presence of multiple holes makes it necessary to both specify the holes in which the composed contexts are placed, and identify the holes in the resulting contexts.

Footprints. In Chapter 8, we discussed the problem of defining the memory footprint of tree update commands, and of providing local specifications for the commands that only mention this footprint. In fact, giving a formal definition of footprints is an interesting problem even in the simpler setting of Separation Logic. For example, consider the following simple heap update program, pointed out by Yang:

$$\mathbb{C} \triangleq x := \text{new}() \text{ ; dispose } x \text{ ; if } x = y \text{ then } z := 0 \text{ else } z := 1$$

This assigns the variable z a value that depends on the non-deterministically assigned address of the cell x, which is dynamically allocated and then immediately disposed. The program is local (since all its constituent commands are), and requires no specific heap resource to execute. Hence, in one sense, its footprint is empty. However, the behaviour of the program clearly depends on whether a cell at y is already present in the heap during allocation, since in that case it is known that x cannot equal y and that consequently z always gets assigned the value 1. Hence, in another sense, the cell at y is also part of the footprint. Indeed, to locally specify the program, one must describe its behaviour on both these 'footprints':

$$\begin{array}{rcl} \{\text{emp}\} & \mathbb{C} & \{\text{emp} \wedge (z = 0 \vee z = 1)\} \\ \{y \mapsto y'\} & \mathbb{C} & \{y \mapsto y' \wedge (z = 1)\} \end{array}$$

For Context Logic, the situation is still more complicated, since the hierarchical nature of data also means that multiple footprints cannot be trivially combined into one.

Thus, for example, a tree update command that moves the subtree at a location n_1 to a location n_2 clearly only affects the subtrees at these locations. To combine these subtrees into a single footprint, however, involves describing the smallest tree containing both n_1 and n_2, which might also contain a significant amount of other unaffected data. An alternative approach which is worth investigating is to consider the footprint to be a *set* containing the two subtrees, thus moving the reasoning framework from dealing with data to dealing with sets of data. This is made more challenging by the fact that the two subtrees may overlap.

Integrating high-level and low-level reasoning. Another interesting problem involves trying to integrate the high-level reasoning in Context Logic with low-level reasoning about data update in Separation Logic. This raises yet more questions concerning footprints: for example, consider a command dispose n acting on linear sequences (see Section 5.3) that removes the subelement n in a sequence if it is present, and faults otherwise. This command can easily be axiomatised locally in Context Logic, mentioning just the subelement n:

$$\{\mathsf{n}\}\ \text{dispose}\ \mathsf{n}\ \{0\}$$

At the low-level, it is possible to naturally represent sequences as doubly-linked lists: for example, the sequence $\mathsf{m} * \mathsf{n}$ can be represented by the heap $(\mathsf{m} \mapsto \text{nil}, \mathsf{n}) * (\mathsf{n} \mapsto \mathsf{m}, \text{nil})$ containing two cells with backwards and forwards pointers. The dispose n command, meanwhile, can be represented using the following low-level program, which updates the pointers at the previous and subsequent cells (if they exist) before disposing of n:

$$\begin{array}{l}\text{if } [\mathsf{n}.\text{left}] \neq \text{nil then } [\mathsf{n}.\text{left}].\text{right} := \mathsf{n}.\text{right} \ ; \\ \text{if } [\mathsf{n}.\text{right}] \neq \text{nil then } [\mathsf{n}.\text{right}].\text{left} := \mathsf{n}.\text{left} \ ; \\ \text{dispose } \mathsf{n}\end{array}$$

The footprint of this implementation, however, is clearly more than just the cell at n: it now also includes the previous and subsequent cells in the sequence. Thus, a local specification of the command might look something like this:

$$\{(\mathsf{n}_{-1} \mapsto \mathsf{n}_{-2}, \mathsf{n}) * (\mathsf{n} \mapsto \mathsf{n}_{-1}, \mathsf{n}_1) * (\mathsf{n}_1 \mapsto \mathsf{n}, \mathsf{n}_2)\}\ \text{dispose}\ \mathsf{n}\ \{(\mathsf{n}_{-1} \mapsto \mathsf{n}_{-2}, \mathsf{n}_1) * (\mathsf{n}_1 \mapsto \mathsf{n}_{-1}, \mathsf{n}_2)\}$$

with additional specifications for the cases when either of the pointers is nil. This low-level/high-level footprint disparity comes from the difference at the low-level

between the representation of full sequences (when the back and forwards pointers are nil) and subsequences (when they are not), a distinction which does not exist at the high level.

Tree update. A particularly wide-ranging area of future work concerns tree update in general. In Chapter 8, we successfully applied Context Logic to a semi-realistic tree update language with queries and pointers. However, as discussed in the assessment at the end of that chapter, there still remains much work to be done in this area, both in terms of high-level tree update language design and in deciding how best to reason about such languages. Furthermore, it would also be interesting to take a more in-depth look at low-level tree update languages such as the DOM interface, to see how Context Logic can help in specifying these formally and reasoning about them. For example, the DOM interface is currently specified in English, and would benefit greatly from a concise (and compositional) formal specification. However, adapting Context Logic to DOM is not trivial, since DOM treats trees in a different way to what we have seen so far: for one, it makes a firm distinction between trees (which are referred to by their root node) and forests (which contain a list of such node identifiers), and furthermore allows a collection of trees and tree fragments to inhabit the same state space. Extending Context Logic to describe this three-level hierarchy would be an interesting challenge, and a first step towards a better understanding of both Context Logic and DOM.

Concurrency and distribution. An important part of future work on tree update involves looking at concurrency and distribution. While a significant amount of research has already been done on concurrent heap update using Separation Logic, concurrent in-place update for trees has barely been explored despite its obvious applicability to web services and native XML databases. Potential work in this area includes developing and reasoning about high-level concurrent tree update languages, as well as looking at concurrent extensions to the DOM interface. Similarly, distribution, a natural property of data on the web, is not explicitly dealt with in the current framework. Most current research into distribution is based on process calculi and uses the global notion of bisimulation. It would therefore be particularly interesting to see how distribution can coexist with local reasoning.

General data update. Finally, the strong similarities observed between the different forms of update reasoning explored in this thesis suggest that a general theory of local data update is possible. A first step towards this is to adapt Yang's method of local Hoare reasoning for arbitrary heap update commands [Yan01b] to our models of data update. A more ambitious aim is to try to develop a general notion of local action which is independent of any specific model, and applies to arbitrary data update, including higher-order imperative update such as in ML.

Unknown unknowns. The paragraphs above describe some of the main 'known unknowns'; of course, the are also many 'unknown unknowns' [Rum02] which will surely be encountered along the way.

Notation Index

Bibliography

[ABS99] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: from relations to semistructured data and XML*. Morgan Kaufmann, 1999.

[ACe07] ACeDB database. Online at `http://www.acedb.org/`, accessed 2007.

[BBFV05a] M. Benedikt, A. Bonifati, S. Flesca, and A. Vyas. Adding updates to XQuery: Semantics, optimization, and static analysis. In *Proceedings of XIME-P*, 2005.

[BBFV05b] M. Benedikt, A. Bonifati, S. Flesca, and A. Vyas. Verification of tree updates for optimization. In *Proceedings of CAV*, volume 3576 of *LNCS*, pages 379–393. Springer-Verlag, 2005.

[BBTS05] B. Biering, L. Birkedal, and N. Torp-Smith. BI-hyperdoctrines and higher order separation logic. In *Proceedings of ESOP*, volume 3444 of *LNCS*, pages 233–247. Springer-Verlag, 2005.

[BCC04] M. Bugliesi, G. Castagna, and S. Crafa. Access control for mobile agents: the calculus of boxed ambients. *ACM Transactions on Programming Languages and Systems*, 26(1):57–124, 2004.

[BCDO05] J. Berdine, B. Cook, D. Distefano, and P.W. O'Hearn. Symbolic execution with separation logic. In *Proceedings of APLAS*, volume 3780 of *LNCS*, pages 52–68. Springer-Verlag, 2005.

[BCDO06] J. Berdine, B. Cook, D. Distefano, and P.W. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *Proceedings of CAV*, volume 4144 of *LNCS*, pages 386–400. Springer-Verlag, 2006.

[BCF03] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *Proceedings of ICFP*, pages 51–63. ACM Press, 2003.

[BCO04a] J. Berdine, C. Calcagno, and P. O'Hearn. A decidable fragment of separation logic. In *Proceedings of FSTTCS*, volume 3328 of *LNCS*, pages 97–109. Springer-Verlag, 2004.

[BCO04b] R. Bornat, C. Calcagno, and P.W. O'Hearn. Local reasoning, separation and aliasing. In *Proceedings of SPACE*, 2004.

[BCO06] J. Berdine, C. Calcagno, and P.W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Proceedings of FMCO*, volume 4111 of *LNCS*, pages 115–137. Springer-Verlag, 2006.

[BCOP05] R. Bornat, C. Calcagno, P.W. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *Proceedings of POPL*, pages 259–270. ACM Press, 2005.

[BCY06] R. Bornat, C. Calcagno, and H. Yang. Variables as resource in separation logic. In *Proceedings of MFPS*, volume 155 of *ENTCS*, pages 247–276. Elsevier, 2006.

[BdV01] P. Blackburn, M. de Rijke, and Y. Venema. *Modal Logic*. Cambridge University Press, 2001.

[BG03] N. Biri and D. Galmiche. A separation logic for resource distribution. In *Proceedings of FSTTCS*, volume 2914 of *LNCS*, pages 23–37. Springer-Verlag, 2003.

[Bir05] N. Biri. *Logiques spatiales de ressources, modèles d'arbres et applications*. PhD thesis, Université Henri Poincaré, 2005.

[Bro04] S.D. Brookes. A semantics for concurrent separation logic. In *Proceedings of CONCUR*, volume 3170 of *LNCS*, pages 16–34. Springer-Verlag, 2004.

[BTSR04] L. Birkedal, N. Torp-Smith, and J.C. Reynolds. Local reasoning about a copying garbage collector. In *Proceedings of POPL*, pages 220–231. ACM Press, 2004.

[Bun97] P. Buneman. Semistructured data. In *Proceedings of PODS*, pages 117–121. ACM Press, 1997.

[Bur72] R. Burstall. Some techniques for proving correctness of programs which alter data structures. In *Proceedings of MIW*, volume 7 of *Machine Intelligence*, pages 23–50, 1972.

[Car01] L. Cardelli. Describing semistructured data. *SIGMOD Record*, 30(4):80–85, December 2001.

[CC02] L. Caires and L. Cardelli. A spatial logic for concurrency (part ii). In *Proceedings of ICCT*, volume 2421 of *LNCS*, pages 209–225. Springer-Verlag, 2002.

[CC03] L. Caires and L. Cardelli. A spatial logic for concurrency (part i). *Information and Computation*, 186(2):194–235, 2003.

[CCG03] C. Calcagno, C. Cardelli, and A. Gordon. Deciding validity in a spatial logic for trees. In *Proceedings of TLDI*. ACM Press, 2003.

[CDOY06] C. Calcagno, D. Distefano, P.W. O'Hearn, and H. Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In *Proceedings of SAS*, volume 4134 of *LNCS*, pages 182–203. Springer-Verlag, 2006.

[CG00a] L. Cardelli and A.D. Gordon. Anytime, anywhere: modal logics for mobile ambients. In *Proceedings of POPL*, pages 365–377. ACM Press, 2000.

[CG00b] L. Cardelli and A.D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, June 2000.

[CG01] L. Cardelli and A.D. Gordon. Logical properties of name restriction. In *Proceedings of CLTA*, volume 2044 of *LNCS*, pages 46–60. Springer-Verlag, 2001.

[CG03] L. Cardelli and A.D. Gordon. Equational properties of mobile ambients. *Mathematical Structures in Computer Science*, 13(3):371–408, June 2003.

[CG04] L. Cardelli and A.D. Gordon. TQL: a query language for semistructured data based on the ambient logic. *Mathematical Structures in Computer Science*, 14(3):285–327, June 2004.

[CG07] L. Cardelli and A.D. Gordon. Ambient logic. To appear in Mathematical Structures in Computer Science, 2007.

[CGG00] L. Cardelli, G. Ghelli, and A.D. Gordon. Types for the ambient calculus. *Information and Computation*, 177(2):160–194, September 200.

[CGG02] L. Cardelli, P. Gardner, and G. Ghelli. A spatial logic for querying graphs. In *Proceedings of ICALP*, volume 2380 of *LNCS*, pages 597–610. Springer-Verlag, 2002.

[CGG03a] L. Cardelli, P. Gardner, and G. Ghelli. Manipulating trees with hidden labels. In *Proceedings of FOSSACS*, volume 2620 of *LNCS*, pages 216–232. Springer-Verlag, 2003.

[CGG03b] L. Cardelli, P. Gardner, and G. Ghelli. Querying trees with pointers. Unpublished notes, 2003.

[CGH05] C. Calcagno, P. Gardner, and M. Hague. From separation logic to first-order logic. In *Proceedings of FOSSACS*, volume 3441 of *LNCS*, pages 395–409. Springer-Verlag, 2005.

[CGMH+94] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *Proceedings of the 100th Anniversary Meeting, Information Processing Society of Japan*, pages 7–18, 1994.

[CGZ04] C. Calcagno, P. Gardner, and U. Zarfaty. A context logic for tree update. In *Proceedings of Workshop on Logics for Resources, Processes and Programs (LRPP)*, 2004.

[CGZ05] C. Calcagno, P. Gardner, and U. Zarfaty. Context logic & tree update. In *Proceedings of POPL*, pages 271–282. ACM Press, 2005.

[CGZ07a] C. Calcagno, P. Gardner, and U. Zarfaty. Context logic as modal logic: Completeness and parametric inexpressivity. In *Proceedings of POPL*. ACM Press, 2007.

[CGZ07b] C. Calcagno, P. Gardner, and U. Zarfaty. Local reasoning about data update. To appear in Gordon Plotkin Festschrift, 2007.

[CM98] L. Caires and L. Monteiro. Verifiable and executable logic specifications of concurrent objects in $\mathcal{L}_\pi$. In *Proceedings of ESOP*, volume 1381 of *LNCS*, pages 42–56. Springer-Verlag, 1998.

[CMS05a] G. Conforti, D. Macedonio, and V. Sassone. Bigraphical logics for XML. In *Proceedings of SEBD*, pages 392–399, 2005.

[CMS05b] G. Conforti, D. Macedonio, and V. Sassone. Spatial logics for bigraphs. In *Proceedings of ICALP*, volume 3580 of *LNCS*, pages 766–778. Springer-Verlag, 2005.

[Con05] G. Conforti. *Spatial Logics for Semi-Structured Resources*. PhD thesis, Università degli Studi di Pisa, 2005.

[CPR06] B. Cook, A. Podelski, and A. Rybalchenko. Terminator: Beyond safety. In *Proceedings of CAV*, volume 4144 of *LNCS*, pages 415–418. Springer-Verlag, 2006.

[CRZ03] A.B. Chaudhri, A. Rashid, and R. Zicari. *XML Data Management: Native XML and XML-enabled Database Systems*. Addison-Wesley, 2003.

[CYO01] C. Calcagno, H. Yang, and P. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In *Proceedings of FSTTCS*, volume 2245 of *LNCS*, pages 108–119. Springer-Verlag, 2001.

[Dal01] S. Dal Zilio. Fixed points in the ambient logic. In *Proceedings of FICS*, 2001.

[DGG04] A. Dawar, P. Gardner, and G. Ghelli. Adjunct elimination through games in static ambient logic. In *Proceedings of FSTTCS*, volume 3328 of *LNCS*, pages 211–223. Springer-Verlag, 2004.

[DGG07] A. Dawar, P. Gardner, and G. Ghelli. Expressiveness and complexity of graph logic. To appear in Information and Computation, 2007.

[DLM04] S. Dal Zilio, D. Lugiez, and C. Meyssonnier. A logic you can count on. In *Proceedings of POPL*, pages 135–146. ACM Press, 2004.

[DOM04] DOM: Document Object Model. W3C recommendation, April 2004. Available at `http://www.w3.org/DOM/DOMTR`.

[DOY06] D. Distefano, P.W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *Proceedings of TACAS*, volume 3920 of *LNCS*, pages 287–302. Springer-Verlag, 2006.

[FFS07] X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *Proceedings of ESOP*, 2007.

[Flo67] R.W. Floyd. Assigning meaning to programs. In *Proceedings of Symposia in Applied Mathematics*, volume 19, pages 19–32. American Mathematical Society, 1967.

[Gar05] P. Gardner. A note on context logic, 2005. Notes accompanying a lecture course for the APPSEM summer school.

[GBC06] A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *Proceedings of SAS*, volume 4134 of *LNCS*, pages 240–260. Springer-Verlag, 2006.

[Gir87] J-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, January 1987.

[GR91] C.F. Goldfarb and Y. Rubinsky. *The SGML handbook*. OUP, 1991.

[GRS06] G. Ghelli, C. Ré, and J. Siméon. XQuery!: An XML query language with side effects. In *Proceedings of DATAX*, volume 4254 of *LNCS*, pages 178–191. Springer-Verlag, 2006.

[HLS02] D. Hirschkoff, E. Lozes, and D. Sangiorgi. Separability, expressiveness, and decidability in the ambient logic. In *Proceedings of LICS*, pages 423–432. IEEE Computer Society, 2002.

[Hoa69] C.A. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[Hoa71] C.A. Hoare. Proof of a program: Find. *Communications of the ACM*, 14(1):39–45, January 1971.

[HP02] H. Hosoya and B.C. Pierce. Regular expression pattern matching for XML. *Journal of Functional Programming*, 13(6):961–1004, 2002. Short version appeared in POPL 2001, pp. 67-80.

[HP03] H. Hosoya and B.C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, May 2003. Short version appeared in WebDB 2000, pp. 226-244.

[Hut99] G. Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372, July 1999.

[IO01] S. Isthiaq and P.W. O'Hearn. BI as an assertion language for mutable data structures. In *Proceedings of POPL*, pages 14–26. ACM Press, 2001.

[JM03] O.H. Jensen and R. Milner. Bigraphs and transitions. In *Proceedings of POPL*, pages 38–49. ACM Press, 2003.

[JM04] O.H. Jensen and R. Milner. Bigraphs and mobile processes (revised). Technical report, University of Cambridge, Computer Laboratory, 2004.

[JOW06] C. Jones, P. O'Hearn, and J. Woodcock. Verified software: A grand challenge. *Computer*, 39(4):93–95, April 2006.

[Koz82] D. Kozen. Results on the propositional μ-calculus. In *Proceedings of ICALP*, volume 140 of *LNCS*, pages 348–359. Springer-Verlag, 1982.

[Leh01] P. Lehti. Design and implementation of a data manipulation processor for an xml query processor, 2001. Technical Report, Technische Universitat Darmstadt. Report KOM-D-149.

[Loz04a] É. Lozes. Adjuncts elimination in the static ambient logic. In *Proceedings of EXPRESS*, volume 96 of *ENTCS*, pages 51–72. Elsevier, 2004.

[Loz04b] É. Lozes. Comparing the expressive power of separation logic and classical logic. Short Presentation, 2004.

[Loz05] É. Lozes. Elimination of spatial connectives in static spatial logics. *Theoretical Computer Science*, 330(3):475–499, February 2005.

[LS03] F. Levi and D. Sangiorgi. Mobile safe ambients. *ACM Transactions on Programming Languages and Systems*, 25(1):1–69, 2003.

[Mat03] MathML: Mathematical Markup Language. W3C recommendation, version 2.0 (second edition), October 2003. Available at `http://www.w3.org/TR/MathML2/`.

[MH69] J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.

[Mil01] R. Milner. Bigraphical reactive systems. In *Proceedings of CONCUR*, volume 2154 of *LNCS*, pages 16–35. Springer-Verlag, 2001.

[Mil04] R. Milner. Bigraphs for petri nets. *Lectures on Concurrency and Petri Nets: Advances in Petri Nets*, 3098:686–701, 2004.

[Mil06] R. Milner. Pure bigraphs: structure and dynamics. *Information and Computation*, 204(1):60–122, January 2006.

[O'H04] P.W. O'Hearn. Resources, concurrency and local reasoning. In *Proceedings of CONCUR*, volume 3170 of *LNCS*, pages 49–67. Springer-Verlag, 2004.

[OOX07] OpenXML Developer. Online at `http://openxmldeveloper.org/`, accessed 2007.

[OP99] P.W. O'Hearn and D. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, June 1999.

[ORY01] P.W. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of CSL*, volume 2142 of *LNCS*, pages 1–19. Springer-Verlag, 2001.

[PB05] M. Parkinson and G. Bierman. Separation logic and abstraction. In *Proceedings of POPL*, pages 247–258. ACM Press, 2005.

[POY04] D. Pym, P.W. O'Hearn, and H. Yang. Possible worlds and resources: the semantics of bi. *Theoretical Computer Science*, 315(1):257–305, May 2004.

[PV02] I. Phillips and M. Vigliotti. On reduction semantics for the push and pull ambient calculus. In *Proceedings of IFIP*, pages 550–562. Kluwer, 2002.

[Rey00] J. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Proceedings of 1999 Oxford-Microsoft Symposium in Honour of Sir Tony Hoare*, pages 303–321. Palgrave Macmillan, 2000.

[Rey02] J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings of LICS*, pages 55–74. IEEE Computer Society, 2002. Invited Paper.

[Rey03] J. Reynolds. A short course in separation logic, Fall 2003. BRICS minicourse.

[Rey05] J. Reynolds. Precise, intuitionistic, and supported assertions in separation logic, 2005. Invited talk at MFPS XXI.

[Rum02] D. Rumsfeld. Defense department briefing, February 12, 2002.

[Rys02] M. Rys. Proposal for an XML data modification language (version 3), May 2002. Microsoft Corporation, Redmond WA.

[SHS04] G. Sur, J. Hammer, and J. Simeon. An XQuery-based language for processing updates in XML. In *Proceedings of PLAN-X*, 2004.

[SVG03] SVG: Scalable Vector Graphics. W3C recommendation, version 1.1, January 2003. Available at `http://www.w3.org/TR/SVG/`.

[TER07] TERMINATOR: Automatic proof tools for termination & liveness. Online at `http://research.microsoft.com/TERMINATOR/`, accessed 2007.

[TIHW01] I. Tatarinov, Z.G. Ives, A.Y. Halevy, and D.S. Weld. Updating XML. In *Proceedings of COMAD*, pages 413–424. ACM Press, 2001.

[TZH04] D. Teller, P. Zimmer, and D. Hirschkoff. Using ambients to control resources. *International Journal of Information Security*, 2(3):126–144, August 2004.

[VP06] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. Submitted, 2006.

[WWW07] W3C: the World Wide Web consortium. Online at `http://www.w3.org/`, accessed 2007.

[XHT02] XHTML: the eXtensible HyperText Markup Language. W3C recommendation, version 1.0 (second edition), August 2002. Available at `http://www.w3.org/TR/xhtml1/`.

[XML01] XML Schema. W3C recommendation, May 2001. Available at `http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/`.

[XML06] XML: eXtensible Markup Language. W3C recommendation, version 1.0 (fourth edition), September 2006. Available at `http://www.w3.org/TR/REC-xml/`.

[XPA99] XPATH: XML Path Language. W3C recommendation, version 1.0, November 1999. Available at `http://www.w3.org/TR/xpath/`.

[XQu06] XQuery Update Facility. W3C Working Draft 11, July 2006. Available at `http://www.w3.org/TR/xqupdate/`.

[XQu07] XQuery: an XML Query language. W3C recommendation, version 1.0, January 2007. Available at `http://www.w3.org/TR/xquery/`.

[XSL99] XSLT: XSL Transformations. W3C recommendation, version 1.0, November 1999. Available at `http://www.w3.org/TR/xslt`.

[XUp00] XUpdate - XML Update Language. XML:DB initiative, working draft, September 2000. Available at `http://xmldb-org.sourceforge.net/xupdate/`.

[Yan01a] H. Yang. An example of local reasoning in bi pointer logic: the schorr-waite graph marking algorithm. In *Proceedings of SPACE*, 2001.

[Yan01b] H. Yang. *Local programming for stateful programs.* PhD thesis, University of Illinois at Urbana-Champaign, 2001.

[YO02] H. Yang and P.W. O'Hearn. A semantic basis for local reasoning. In *Proceedings of FOSSACS*, volume 2303 of *LNCS*, pages 402–416. Springer-Verlag, 2002.

[ZG06] U. Zarfaty and P. Gardner. Local reasoning about tree update. In *Proceedings of MFPS*, volume 158 of *ENTCS*, pages 399–424. Elsevier, 2006.